METHODOLOGY FOR MODELING HIGH PERFORMANCE DISTRIBUTED AND PARALLEL SYSTEMS

by
Rakesh Kushwaha

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

October 1993

APPROVAL PAGE

Methodology for Modeling High-Performance Distributed and Parallel Systems

Rakesh Kushwaha

8/24/93

Dr. Erol Gelenbe, Dissertation Advisor (Date)
Professor of Electrical Engineering and Computer Science, Duke University

8/27/93

Dr. Peter A. Ng, Committee Member (Date)
Chairperson and Professor of Computer and Information Science, NJIT

8/24/93

Dr. C.N. Manikopoulos, Committee Member (Date)
Associate Professor of Electrical and Computer Engineering, NJIT

Aug. 27, 93

Dr. David Nassimi, Committee Member (Date)
Associate Professor of Computer and Information Science, NJIT

8/27/93

Dr. Bruce Parker, Committee Member (Date)
Assistant Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Rakesh Kushwaha

Degree: Doctor of Philosophy in Computer Science

Date: October 1993

Date of Birth:

Place of Birth:

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1993

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1989

- Bachelor of Engineering in Mechanical Engineering,
  Delhi University, New Delhi, India, 1986

Major: Computer Science

Presentations and Publications:

R. Kushwaha, "Methodology for predicting Performance of Distributed and Parallel systems," *Performance Evaluation*, vol. 18, 1993.

E. Gelenbe and R. Kushwaha, "Incremental Dynamic Load Balancing in Distributed Systems," To appear in *International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS'94*, Jan 1994.

R. Kushwaha, "Optimal file placement and Performance Analysis of a Multicomputer System," Tech. Rep. CIS-93-07, Department of Computer and Information Sciences, NJIT, July 1993.

R. Kushwaha, *Design and Development of Distributed File System in 4.3 BSD UNIX*, Masters thesis, NJIT, May 1989.

dedicated to
my Mother
Prabha Kushwaha
.... source of inspiration

ACKNOWLEDGMENT

I am grateful to many individuals for their valuable contribution that made this dissertation possible. First on my list is my thesis advisor, Professor Erol Gelenbe. Erol has broadened my perception of not only research in Computer Science, but of life in general through his amazing incisiveness, knowledge and boundless enthusiasm for intellectual pursuit. His advice and criticism have always helped me not to lose sight of real objectives. He taught me what research and writing papers was all about.

Secondly I would like to thank Professor Peter A. Ng, committee member and my supervisor, who provided the initial incentive to pursue the journey towards Ph.D. and thereafter, constantly supported and funded this research. Thanks to Professors David Nassimi, Bruce Parker and Dino Manikopoulos for serving as members of the committee. I am in particular indebted to Bruce for providing pointers to the abundance of literature in the area of Distributed Systems. Special thanks to David for his support, encouragement and guidance throughout this dissertation.

I am grateful to the faculty, staff and students of the Computer and Information Science Department of New Jersey Institute of Technology. Thanks to Michael Tress, Fadi Deek and Leon Jololian for their moral support; Felicia and Brian for technical support; special thanks to Carol, Barbara, Rosemarie and Michelle for their friendship and for making my stay in the department so much fun.

I relied on the help and friendship of many individuals to support my studies. In particular, I thank Samir Chopra for reading different versions of the manuscript, helpful discussions and numerous late-night philosophizing sessions; James Whitescarver of CCCC for thought-provoking discussions and providing invaluable suggestions and insights concerning an object-oriented model; Sal Johar for always beaming me with positive thoughts and Eileen Michie for reading the final version of the manuscript.

I am grateful to many friends that have made my stay in New Jersey memorable. Among them are - Pamela Cham, Christine Hubert, Fortune Mhlanga, Michail

Papamichail, Xiowen Mang, Hans-Peter Meske, Alex Stoyenko, Lonnie Welch, Wilhelm Rossak, Vassilka Kirova, Volkan Atalay, Myriam Mukhtari, Michael Halper, Ajaz Rana, Steve Chiang, Davendra Vamathevan, Jenlong Moh and all the visitors at 432 Maple Ave., Linden.

I wish to express my appreciation to the faculty and staff of the Electrical Engineering Department at the Duke University. In particular I thank Professor Kishore Trevedi and Ms. Margrid Krueger for allowing unrestricted access to their laboratory and making my stay at the Duke University comfortable.

I would also like to thank the entire Johar family, whose love, concern and continuous encouragement supported me during the strenuous period of my Ph.D. I cannot omit admiration and thanks to my mother, brother, and my relatives, in particular Dr. Usha Sharma, for all the things they did for me during my academic endeavours.

TABLE OF CONTENTS

Chapter Page

1 INTRODUCTION 1

1.1 Overview of Distributed and Parallel Systems 2

1.2 Importance of Performance Prediction and Modeling 6

1.3 Motivation for a New Model 8

1.4 Contents of the Thesis 11

2 CURRENT MODELS AND PREDICTION METHODOLOGIES 14

2.1 Design Issues 14

2.2 Performance Models for Distributed and Parallel Systems 16

2.2.1 Data-Allocation Models 17

2.2.2 Task-Allocation Parallel Processing Models 19

2.2.3 Transaction processing System Models with Load Balancing .. 22

2.3 Current Prediction Methods 25

2.3.1 Simulation Methods 25

2.3.2 Analytical Methods 28

2.3.2.1 Queueing Methods 29

2.3.2.2 Graphical Methods 32

2.3.2.3 Other Methods 34

3 THE PERFORMANCE OF A FILE-SERVER MODEL 37

3.1 Introduction 37

3.2 The System Model 39

Chapter Page

3.2.1 System Description 39

3.2.2 The Queueing Network Model 44

3.3 Analysis 46

3.4 Results and Validation 52

3.4.1 Numerical Example 52

3.4.2 Validation 54

3.5 Approximate Model For Cache-Size Analysis 58

3.6 Bottleneck Identification 65

3.7 Conclusion 71

4 OPTIMAL FILE PLACEMENT STRATEGY 72

4.1 Introduction 72

4.2 The Extended Model 74

4.3 Average Node Response Time Based File Allocation Algorithm 76

4.3.1 Single Chain Case 77

4.3.1.1 File Allocation as a Routing Problem 79

4.3.1.2 Objective Function 81

4.3.2 Extension to Multiple Chain 82

4.3.2.1 Algorithm 84

4.3.3 Algorithm Convergence and Complexity 86

4.4 Performance Analysis 87

4.5 Numerical Examples 89

Chapter Page

4.5.1 Example 1 90
4.5.2 Example 2 90
4.6 Measurements and Validation 95
4.7 Conclusion 98

5 ON-LINE ADAPTIVE ALGORITHM FOR PROCESS MIGRATION 101

5.1 Introduction 101
5.2 Overview of Adaptive Algorithms 102
5.3 The Distributed System Environment 104
5.3.1 Assumptions concerning system operation 105
5.4 Simple and effective load-balancing policies 106
5.5 Adaptive Algorithm Design 108
5.5.1 Algorithm design 110
5.6 Performance metrics and experimental comparison of policies 114
5.6.1 Experimental results 115
5.6.2 Response-Time Comparisons with Different Load Values 122
5.7 Conclusions 126

6 SUMMARY AND CONCLUSION 128

6.1 Summary of the Methodology 128
6.2 Application Considerations 130
6.3 Comparisons with Other Methods 132
6.4 Suggestions for Future Research 134
6.4.1 Distributed Object-Oriented Model 134
6.4.2 Other Issues 136

Chapter Page

APPENDIX A 138

APPENDIX B 140

APPENDIX C 142

REFERENCES 145

LIST OF TABLES

Table		Page
1	Comparison of analytical and simulation values	58
2	Marginal probabilities for network compared with simulation results	59
3	Initial file-allocation pattern # 1 of Example 1	91
4	Initial file-allocation pattern # 2 of Example 1	91
5	Initial file-allocation pattern # 3 of Example 1	91
6	Optimal file-allocation pattern of Example 1	92
7	Node utilization compared with measurements from simulation	98
8	Comparison of average node response time and utilization	99
9	Comparative efficiency and performance for four load balancing policies	117
10	Comparative efficiency and performance measures.	119

LIST OF FIGURES

Figure		Page
1	Generic distributed and parallel system architecture	6
2	Flow of information in the distributed systems	44
3	The queueing network model	45
4	Network and server utilization increases as user think time decreases	53
5	Average node-response time of server and network node	55
6	The average system-response time as a function of think time	55
7	Comparison of analytical and simulation values	59
8	Marginal Probabilities for network compared with simulation results	60
9	Approximate model; each client is modeled as M/G/1 queue	61
10	Effect of cache size on probabilities	64
11	Effect of cache size on performance	65
12	Effect of cache size on the average system response time	66
13	Multi-client multi-server model	67
14	Asymptotic behavior of multiple resource system	69
15	Performance of multiple resource system	70
16	Average node utilization as a function of think time	93
17	Average node response time as a function of think time	94
18	Average throughput of a node as a function of think time	94
19	Average utilization of a node due to different job classes	95
20	Comparison of the instantaneous load for the AD-TWO and NP	120
21	Comparison of the instantaneous load for the AD-TWO and RD	120
22	Average process-response time as a function of execution time	121
23	Average process-response time as a function of execution time	123

Figure Page

24 Overhead as a function of process execution time 124
25 Average process-response time as a function of process-creation rate 125
26 Overhead as a function of the process-creation rate at each node . . . 126

ABSTRACT

Methodology for Modeling High Performance Distributed and Parallel Systems

by

Rakesh Kushwaha

Performance modeling of distributed and parallel systems is of considerable importance to the high performance computing community. To achieve high performance, proper task or process assignment and data or file allocation among processing sites is essential. This dissertation describes an elegant approach to model distributed and parallel systems, which combines the optimal static solutions for data allocation with dynamic policies for task assignment. A performance-efficient system model is developed using analytical tools and techniques.

The system model is accomplished in three steps. First, the basic client-server model which allows only data transfer is evaluated. A prediction and evaluation method is developed to examine the system behavior and estimate performance measures. The method is based on known product form queueing networks. The next step extends the model so that each site of the system behaves as both client and server. A data-allocation strategy is designed at this stage which optimally assigns the data to the processing sites. The strategy is based on flow deviation technique in queueing models. The third stage considers process-migration policies. A novel on-line adaptive load-balancing algorithm is proposed which dynamically migrates processes and transfers data among different sites to minimize the job execution cost.

The gradient-descent rule is used to optimize the cost function, which expresses the cost of process execution at different processing sites.

The accuracy of the prediction method and the effectiveness of the analytical techniques is established by the simulations. The modeling procedure described here is general and applicable to any message-passing distributed and parallel system. The proposed techniques and tools can be easily utilized in other related areas such as networking and operating systems. This work contributes significantly towards the design of distributed and parallel systems where performance is critical.

CHAPTER 1

INTRODUCTION

Computer science is one of the fastest growing scientific fields. At present, it is on the verge of a paradigm shift, from traditional von-Neumann architectures which allow only sequential processing, to *parallel and distributed* architectures, where non-sequential and parallel processing is also possible. Researchers have given particular consideration to system architectures with multiple parallel processing units (PUs). These units are capable of concurrent and asynchronous operations, which can be centrally controlled or the control itself can be distributed. Such architectures manifest the recent advances in hardware technology and the development of effective communication networks. Their range varies from a few processing units to thousands of PU's placed in one small portable box on one side, and the same number of PU's geographically separated and interconnected by a wide area network on the other. There are virtually limitless possibilities for connecting such PU's. Software developers and system designers are faced with the challenge of exploiting the potential benefits provided by these architectures. One of the concerns for such software applications and system designs is to improve performance. The size and design complexity makes the development of such systems time consuming and expensive. We need mechanisms to model these systems and analyze their behavior at the design stage.

This thesis describes a systematic approach to model distributed and parallel systems. Analytical tools are developed to aid the modeling process and techniques are proposed for improving the performance of these systems.

This introductory chapter provides a description of distributed and parallel systems and illustrates the importance of modeling and performance prediction. We discuss different approaches for designing distributed system models and suggest what improvements we plan to achieve with our modeling philosophy. The last section presents the organization of this thesis.

1.1 Overview of Distributed and Parallel Systems

The term distributed and parallel systems is a very general term. Just using the term 'parallel system' does not reveal what kind of system we are dealing with and what exactly is performed in parallel. Other terms such as multiprocessors, multicomputer, multiprocessing systems, distributed processing systems or distributed computing system are commonly used. Kleinrock [1] and Enslow [2], in their studies, cleared some of the ambiguity caused by these terms. They identified four physical components of the system that can be distributed: hardware, data, processes, and control. Since distribution leads to concurrency, the term 'parallel' also finds its place in describing such systems. Some authors [3, 4] consider a system that has any one of the above components distributed to be a distributed system. The concept of distributing these components can be applied to almost any level of the computer system

hierarchy – from the design of a circuit in a VLSI chip all the way to the design of an intercontinental computer network.

Distributed and parallel systems can be *homogeneous* or *heterogeneous*. All processing units in terms of data storage and CPU functionality are similar in homogeneous systems. Design and development of such systems is simpler because of non-discrepancies among processing units. Heterogeneous systems, on the other hand, consist of processing units which need not be similar. Dissimilarities might be in processing speed, data management, job-arrival rate or hardware. Such system design and implementation requires an extra effort. The control can be centralized or distributed in either of the two classes.

Flynn [5] classified distributed and parallel systems from the perspective of control. These are classified as *SIMD* (single instruction and multiple data) and *MIMD* (multiple instructions and multiple data) systems [6, 7]. In a SIMD system, a number of processors simultaneously execute the same instruction. A single control unit (CU) fetches and decodes the instructions. The instruction is executed either in the CU itself (e.g., a jump or any non-parallel instruction) or it is broadcast to processing elements (PE's). These PE's operate synchronously but their local memories have different contents. There exists a bidirectional bus interconnection between the main memory (usually subdivided into a number of memory modules) and the local memories. A system of this type is also called an 'Array Processor' because of the array of processors formed by the PE's. Examples of SIMD systems

include MASPAR [8] and CM-2 [9]. In a MIMD system, a set of processors can simultaneously execute different instruction sequences on different data sets. Some examples are the nCube [10], Intel's iPSC [11], and the Alliant FX/8 [12]. Recently, Thinking Machine Corporation released CM-5, which provides capabilities of both SIMD and MIMD systems.

Geographical location and interconnections among the processing units form the basis for classifying distributed systems as *loosely* or *tightly* coupled. In a tightly coupled system the processing units are interconnected by high speed buses. The communication delays are very small and less significant. Processors can be easily synchronized and may or may not use a single global clock. MASPAR and nCube are two tightly coupled commercial systems. Loosely coupled systems consist of geographically separated processing units interconnected by a wide area network (WAN) or a local area network (LAN). Wide area networks have been available for several years. To date, most wide area computer networks use packet switching technology. LAN's, intended to provide wide bandwidth over a limited distance, have developed rapidly in recent years. Such networks make use of relatively cheap methods of interconnection such as co-axial cable or twisted pairs. Unfortunately there is no common international standard for LAN's and no single technology yet dominates the market.

Another possible alternative for non Von Neumann architecture which aims at high speed computing is data-flow architecture [13, 14, 15]. A typical data flow computer consists of a number of processing units, interconnected by data buses or cross

bar switches, where the focus is on the flow of data rather than on the instructions which process the data. In such models, the data is active and flows asynchronously through the program, activating each instruction when all the required input data has arrived. This is in direct contrast to the Von Neumann model in which data passively resides in storage while instructions are executed one at a time according to a defined sequence. The data-flow systems are usually programmed using a data-flow language [16], which is a subset of the class of functional languages. Such systems have a large overhead involved in explicitly specifying all instruction sequencing.

In this dissertation, more general distributed and parallel system architectures are considered. Such systems are composed of a set of autonomous (and possibly heterogeneous) processors, each of which is fully functional in a stand-alone fashion. Figure 1 depicts a conceptual distributed and parallel system architecture under consideration. The system consists of J nodes which are interconnected through a communication network. Each node consists of a processor(s) and a storage unit. The size and sophistication of each individual processor can range from those of a DMA controller to those of a general-purpose mainframe computer. The storage units can also vary considerably in terms of capacity and technology used.

Our objective is to distribute all four physical components - hardware, data, process, and control - to achieve high performance. Such systems may be loosely or tightly coupled. Any number of processes can concurrently execute different programs using different or the same sets of data. In general, no particular operation selection

constraints are attached. Data, control, and process distribution is transparent to the user. These kinds of systems are typically used for transaction-oriented processing such as accessing and updating distributed data bases. One of the major advantage of applying distributed design concepts at this level is the resource-sharing capability, which results in reducing both cost, by not having to replicate the complete functionality of a system at each site, and system response time, by having idle processors 'assist' overloaded ones.

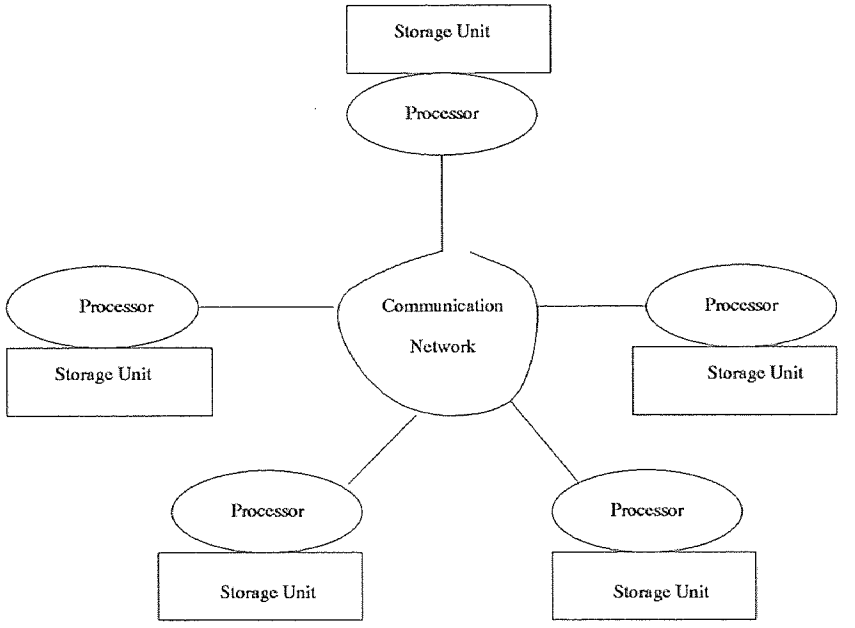

Figure 1: Generic distributed and parallel system architecture

1.2 Importance of Performance Prediction and Modeling

From the brief discussion presented above, we can infer that there are several design alternatives for developing distributed and parallel systems. Some of the important

design considerations are: configuration, size, and technology of the interconnection network; distribution of system functionality among its processing units; allocation of parallel entities to processors; synchronization of execution of tasks; and storage of various data sets. The designer of a distributed system is presented with the dilemma of properly resolving these issues to meet certain objectives. These objective may include performance, reliability, availability, user transparency or fault tolerance. It is usually not until the final stages of system development that it is possible to determine whether the original objectives have been met. Thus, an incorrect choice in deciding on any one of the numerous issues can result in a very costly and time consuming redesign and redevelopment effort.

Given the above considerations, the importance of being able to predict the eventual performance of a system during primary stages of development so that any design flaws can be detected and corrected without great expense is apparent. With proper modeling and performance prediction tools, a designer should be able to gradually 'pilot' the design into meeting all of the required objectives.

Over the years, the field of performance modeling and analysis has made valuable contributions towards system and application design [17], operating systems [18] and networks [19]. Several modeling and prediction methodologies, tools, and techniques [20] have been proposed by various modelers and analysts. The experience and maturity of the field has much to offer towards the efficient design of complex distributed and parallel systems.

1.3 Motivation for a New Distributed and Parallel System Model

Various performance models for distributed and parallel systems have been proposed and have made a significant contribution to the design and implementation of distributed and parallel systems. Scheduling algorithms and load balancing strategies designed for these models are particularly attractive to the development community. The future will see more implementations of performance models since the predictions and approximations made from the models are now increasingly accurate. In addition, the demand to achieve high performance from distributed architectures is steadily increasing.

This section discusses the evolution of distributed and parallel system models. Systems developed on the basis of these models, distributed operating systems in particular, are included for reference. In Chapter 2 we categorize the existing models and explain them in detail.

The *client-server* model [21] is one of the earliest models of a distributed system. In the client-server model, the server manages recoverable data objects and defines operations that are exported to clients. Clients invoke these operations to manipulate the data managed by servers. Operations are invoked by using a remote procedure call (RPC) [22] interface. Argus [23], Helix [24] and conventional databases with multiple accessing sites are implemented based on this model. The control for such systems usually resides in the central server. *Extended client-server* models include cache and increased processing capability at the client to increase availability

and performance. The data is distributed among the server and the clients. The clients and the server can work independently and also collectively to share resources. The control of the complete system can also be distributed among different processing units. SUN NFS [25] exemplifies the efficacy of such a model.

Other variations to the client-server model where data and control is distributed have also been proposed. Data may or may not be replicated in these models. Locus [26] and Sallow, [27] for example, are based on a model which allows data replication. Dunix [28] and Alpine [29] are systems based on the notion of non-replication. Such models generally use some resource sharing strategy to improve their performance.

The advent of efficient process migration techniques led the pioneers of modeling to design system models that dealt with *process migration* and *task allocation.* DEMOS/MP [30], Charlotte [31] and Mach [32] are a few examples of systems based on process migration models. The models developed to date which allow process migration fall into two major categories [33]: process-oriented models - based on the concept of a process, and object oriented models - based on the concept of an object. There is no logical difference between these models. Processes in the former model can be mapped into objects in the latter. Moreover, these two models use similar structuring and synchronization mechanisms and similar operations. However, these models differ at the level of mechanism. Thus, these categories of systems are defined based on the mechanisms by which they implement the notions of functional entity

and synchronization. A comprehensive survey which categorizes all the distributed systems according to the model on which they are based is presented in [34].

The task allocation models use probabilistic or deterministic strategies to assign tasks among distributed hardware, so that the total time to process all tasks is minimized [35, 36]. Models which allow task migration from heavily loaded sites to lightly loaded sites also exist [37]. Lately, models which consider adaptive process distributing strategies have been published [38]. These models assume that all the processes (tasks) are entirely computational or data and process are one functional unit that can be migrated (allocated) to any site. Process migration or task allocation to any arbitrary processing unit is possible if exact data is replicated on all sites. However, data replication on very large distributed systems is not performance efficient, since the algorithms to keep the data consistent are computationally expensive. We need models where both processes and data move independently among the processing units to get the job done. Such models should efficiently distribute control, data, processes, and processing logic to exploit maximum parallelism.

To achieve high performance, system modeling should be performance oriented. MOS (Multicomputer Operating Systems) [39] is one of the few systems that attempt to integrate load balancing philosophies into a distributed system for performance purposes. The system is a distributed implementation of UNIX providing suboptimal load balancing by means of process migration. The primary objective of the system design is to provide site autonomy, decentralized control, and location and

access transparency; while performance is the secondary objective. To design high performance systems, we need models where performance is the key issue. Moreover, these models should also allow designers to incorporate other objectives easily.

In this thesis, we analyze and discuss the development of a performance oriented model for distributed and parallel architectures. Efficient strategies are designed to optimally utilize available resources. Load balancing algorithms are integrated to judiciously distribute both processes and data. A performance prediction methodology is proposed to estimate performance and predict system behavior for such models. We use computer simulations to validate our assertions.

1.4 Contents of the Thesis

In this first chapter, we have presented a general overview of distributed and parallel systems, demonstrated the need for performance prediction and motivated the development of a new performance model. Chapter 2 will elaborate on the different design issues that are relevant to distributed and parallel system design. We also describe current performance models in detail and discuss existing prediction methods.

In Chapter 3 we will present our basic client-server model and outline our performance prediction methodology. The system model is constructed and mapped to a probabilistic queueing network model which is used to predict its behavior. The methodology is applied to identify bottlenecks and to establish proper balance between clients and servers. The model distributes data and provides the frame work

to construct the model where process and control can also be distributed. The proposed performance prediction methodology is general and is applicable to all message passing systems with distributed memory.

Chapter 4 extends the basic client-server model such that each processing site behaves as both a client and a server. The control is distributed along with data among processing sites. To improve performance we propose a load balancing algorithm which optimally allocates data among processing sites. The model is analyzed using the methodology proposed in Chapter 3 and is simulated on the nCube. We compare the analytical results with the measurements obtained from the nCube simulations and discuss their implications.

Chapter 5 completes the model by incorporating techniques to distribute processes in addition to data and control. A novel adaptive load balancing algorithm is designed to migrate jobs and transfer data. It attempts to minimize the cost of job execution. It is distributed, operates on each site, and uses *gradient descent* on a cost function to make decisions. Under the same operating conditions several strategies which use the adaptive algorithm are compared with the ones which do not make use of the algorithm. The performance and efficiency measures are obtained from simulations on the nCube. Results show that the algorithm is effective and has low overhead.

The final chapter concludes the thesis by reviewing the whole modeling approach, identifying its main research contributions, discussing how to effectively ap-

ply performance predicting method and solution procedures for system modeling, and providing guidelines for conducting further research.

CHAPTER 2

CURRENT MODELS AND PREDICTION METHODOLOGIES

Before describing our modeling methodology and prediction method, we identify the design issues relevant to the performance modeling of distributed and parallel systems. We categorize the existing performance oriented distributed and parallel system models and discuss their modeling approach. After introducing the models, we will review current performance prediction methodologies and emphasize their advantages and shortcomings.

2.1 Design Issues

It is clear from the discussion in the previous chapter that designing a distributed system is a complex job. In order to make proper design choices, a system architect must identify those issues which are important for achieving the specified objectives (having the system meet certain requirements). In the following section, we will briefly discuss some of the design issues critical for achieving high system performance.

Perhaps the most important performance-oriented design issue is the *communications mechanism* used for interchanging data and control information among processing elements and storage units of a system. Each individual property of such a communication mechanism must be carefully selected to maximize the system component utilizations and to minimize the communication delays. These properties

include the physical transport medium, the topology of processing elements and data storage modules, data interchange protocols, addressing and routing mechanisms, message sizes, *etc.* The selection of the aforementioned properties is usually constrained by the desired speed and size of the system, the geographical distribution of its components, fault-tolerance and system availability requirements, and bounds on the development and maintenance costs. In most cases, it is not possible for a given choice of the communications mechanism to be optimal in meeting high and balanced system utilization and maintaining fast response time. Thus, a designer has to make a compromise between effectively using available system resources and providing prompt service to the user community.

The proper selection of the *file servicing policy* and the *synchronization mechanism* are also very important for achieving high performance. In architectures which are geared toward specialized applications, wherein the nature of user demands for data is known in advance, it is usually advantageous to design static policies that strategically allocate data or files in the storage. For the programs that can be represented as parallel computations or tasks, efficient task distribution algorithms are desired. A centralized synchronization mechanism can be used for special purpose system as it eases the design and implementation of the system. A general purpose system in an environment of dynamically changing and unpredictable user demands would be better off using decentralized control. Adaptive and heuristic policies for data and process allocation are best suited for such dynamic systems.

Hybrid policies which combine optimal static strategies with the heuristic dynamic policies can also be designed to optimize resource utilization and improve response time. This is what we will demonstrate and test in this dissertation. We first design a simple static strategy for file allocation and later incorporate dynamic policies for process migration and data transfer.

The *cache* plays a significant role in designing high performance distributed systems. Caches at individual processing nodes have been known to significantly improve job execution time, especially in systems having high communication and synchronization overhead [40, 41, 42, 43, 44]. Efficient policies for updating caches have been proposed [45]. It has been shown that caching improves performance while still tolerating failures and provides the same level of coherence, availability, and reliability that the file service would have without caching [46]. Caching also allows overlapping of computation to reduce process-idle time [47].

We consider caching in our client server model and provide cache size analysis in chapter 3.

2.2 Performance Models for Distributed and Parallel Systems

From the perspective of our modeling methodology, we categorize existing models of distributed and parallel system in three categories as listed below:

1. Data (file) allocation models;

2. Task (process) allocation parallel processing models; and

3. Transaction processing system models with load balancing.

The major reason for placing models in three categories above is to emphasize the importance of our design methodology which focuses on optimal data (file) allocation and on load balancing for task (process) allocation. In other words, our objective is to design and model systems to achieve high performance by efficiently distributing both data and processes. Thus our modeling framework reference to all the models in the above three categories.

2.2.1 Data Allocation Models

The earliest attempts to improve performance in distributed systems were made through efficient data (file) allocation. The problem of allocating files at different storage sites was first treated intuitively or by trial and error. Later, Ramamoorty and Chandy [48] and Arora and Gallo [49] proposed analytical solutions to the problem. They concluded that the optimal file allocation strategy was to allocate the more frequently used files to faster devices to the extent possible. However, their model ignored the waiting time in the request queues. Chen [50, 51] performed a more realistic analysis by considering queueing delays and analyzed three types of file allocation problems. The first was to allocate files minimizing the mean overall system response time without considering the storage cost. The second minimized the total storage cost and satisfied one mean overall system response time requirement. The third minimized the total storage cost and satisfied the individual response time

requirements for each file. The first two problems are discussed in [50] and the last in [51].

Dowdy and Foster [52] compared and analyzed several distributed system models for the file assignment problem. They reviewed different models in a uniform manner. They concluded that different performance objectives such as cost, throughput, response time, and access time govern the modeling.

With the advancement in the networking area, wherein multiple machines can be connected by high speed networks, came the design of distributed file systems. Various file system models have been proposed, each providing a different level of functionality and service. File access mechanisms, locality and naming are the issues of interest. Models in the area of extended file systems, which expand across multiple machines, are broadly classified in two categories: *remote file systems* and *distributed file systems*. Both models have several similarities. For instance, both allow files to be placed anywhere in the community, and allow for files to be accessed transparently. The major difference between the two modeling methodologies is in their naming approaches. Distributed file systems provide a single global naming space, whereas remote file access provides a collection of several individual name spaces and a mechanism to connect them together in an arbitrary fashion. The remote file systems are best exemplified by the Newcastle Connection system [53] developed at the University of Newcastle. LOCUS [26] is an example that belongs to the second category. A comprehensive study of all the distributed file systems exists in [34].

Siegel in [54] studied different file systems in the context of performance. The tradeoffs between performance and safety are emphasized. The study concluded that file systems which provide users with various options such as the number of replicated files or disk write modes yield dramatic performance gains.

Apart from distributed file systems and file allocation models, the problem of allocating files is also investigated in distributed computer communication networks. Levin proposed several models for allocating programs and data files in a computer network [55, 56]. Mahamoud studied the problem by simultaneously considering the allocation of files to network nodes and channel capacities to network links [57]. Lanning and Leonard [58] proposed an adaptive algorithm for file placement in computer communications networks, where file storage and file availability are optimized with the possibility of duplicating files for a known maximum number of file copies.

2.2.2 Task Allocation Parallel Processing Models

In recent years we have witnessed a growing interest in the development of multiprocessing computing systems [59]. These systems are composed of multiple processors interconnected to each other and sharing the use of memory, input-output peripherals and other resources. To exploit the parallelism in such systems, various parallel applications have been designed. These applications are represented as concurrent parallel tasks, which co-operate to achieve the desired goal. The tasks are assigned to multiple processors so that the total time required to process the application is min-

imized. Such parallel processing systems are usually tightly coupled and processing units co-operate to achieve one objective. A Large number of parallel applications are also modeled as *fork-join* jobs. In such models, all the tasks of a program arrive simultaneously to the system, they are optimally allocated to PUs and the job is assumed to be finished when the last task completes.

All possible task allocation models can be separated into two general categories: *dynamic* or *static*. For models with the dynamic task allocation policy, each currently enabled task (*i.e.* task which is ready to be executed) competes, on an equal basis, with the other enabled tasks for the processing and communication resources of a distributed system. The required system resources are dynamically assigned to a task (by task scheduling and resource management components of the architecture) at the time when it becomes ready for execution. The assignments of resources to tasks can be made purely probabilistically, without considering the distribution of workload currently present in the system [36, 60], or purely adaptively, as a deterministic function of the current system state [61].

In static models, each task of a given program is a *priori* allocated to a prespecified subset of system's resources [62, 35, 63]. The allocation of resources is performed before commencing the execution of a program. The main objective of most static task allocation strategies is to balance, as much as possible, the expected workload represented by a program among different system components, while exploiting all the potential parallelism available in the program. Upon becoming enabled, each

task of a program utilizes only those resources that are statically assigned to it according to the particular policy adopted by the system in question. An optimal static probabilistic task assignment technique is proposed in [64]. Other static task allocation models include [65, 66, 67, 68].

Towesly *et.al.* [36] developed models for a shared memory multiprocessor that execute fork-join parallel programs. They analyzed models for two processor sharing policies, called *task-scheduling* processor sharing and *job-scheduling* processor sharing. The first policy schedules tasks independently of each other and allows parallel execution of an individual program, whereas the second policy schedules each job as a unit and thereby does not allow parallel execution of an individual program. They concluded that the task-scheduling policy exhibits better performance than the job scheduling policy for most system parameter values. The processors can be dynamically allocated to different tasks.

Setia *et.al.* [69] modeled a system as a distributed fork-join queueing system to evaluate the tradeoff between maximizing parallel execution and minimizing synchronization and communication overheads. They considered a small class of policies that represent typical aspects of processor allocation problem and approximated the expected job response time. Their results show that the solution to the allocation problem is an adaptive policy that spreads the tasks of a job across all processors when the load is light and continuously restricts the degree of parallel execution with increasing load. They demonstrated how quickly the benefits of parallel processing

are negated by its associated overheads across diverse multiprocessor environments.

Nelson *et al.* [70] modeled a centralized parallel processing system with job splitting, using a bulk arrival queueing system. In their model, jobs wait in a central queue, which is accessible by all the processors, and are split into independent tasks that can be executed on separate processors. They studied the effects of parallelism and overheads due to job splitting.

2.2.3 Transaction Processing System Models with Load balancing

Task assignment in a parallel or distributed system uses information available when a job is to be executed, so as to judiciously distribute the tasks among processing units in order to maximize performance. Optimal task assignment is well known to be computationally intractable, *i.e.*, it is NP-hard, so that it cannot be solved exactly when the number of PUs and the number of tasks is large. Therefore it must be implemented by the use of relatively fast heuristics. It is of greatest use when full information about the execution characteristics of parallel or distributed jobs is available before execution, and can only be carried out using sub-optimal strategies.

On the other hand, load balancing is the static or dynamic allocation of tasks (or processes) and data (or files) to processing units (PUs) so that work is equally shared and performance improved. The role of dynamic load balancing is complementary to that of task assignment since it uses on-line information which becomes available at process creation time. It can also be an alternative to task assignment if

relevant information about jobs, such as the number or nature of their parallel tasks or processes, becomes available only during execution.

System architectures which incorporate load balancing strategies are more general. Any tightly or loosely coupled processing units in a homogeneous or heterogeneous environment, interconnected by high speed networks, crossbar switches or any other devices, can be modeled. The models are flexible and are capable of modeling a wide range of distributed and parallel systems. The aim is to distribute load and improve performance. Both static and adaptive load balancing algorithms have been proposed for such distributed system models. If the decision of transferring a process or data is based on the current state of the system the algorithm is dynamic, otherwise it is static. Systems with static control [71, 72, 64, 73, 74] are easier to analyze and model than dynamic ones [75, 76, 77, 78, 79, 80, 81]. It may be argued that the usefulness of static control is limited, however, they are very effective for system sizing, *i.e.*, resource allocation, bottleneck identification and sensitivity studies. Static models are known best to optimize loads according to long-term traffic trends, while dynamic models are designed to react to sudden traffic changes. Load balancing in distributed systems, especially dynamic load balancing, needs information exchange which leads to communication overheads. The objective is to minimize the overhead and maximize the system throughput.

Silva and Gerla in [37] proposed distributed system models to evaluate static load balancing policies, where jobs are migrated from heavily loaded sites to lightly

loaded sites. More detailed distributed system models are considered in [72, 64]. In [82] it is said that "in a network of nodes, there is a very high probability that at least one node is idle while jobs are queued at some other nodes," which motivates the interest in the design of adaptive models for job allocation. Several adaptive load sharing strategies for job allocation have been proposed. Comparative and comprehensive studies include [83] and [77], which point to the potential benefits of adaptive load sharing and compare different policies, concluding that very simple adaptive load sharing policies which collect a small amount of system state information and use this information in simple ways, can yield dramatic performance improvement.

Mirchandaney *et. al.* [79] studied the performance characteristics of simple load sharing algorithms for a heterogeneous distributed system model. They assume that non-negligible delays are encountered in transferring jobs from one node to the another and in gathering remote state information. They analyzed the effects of these delays on the performance of two algorithms called *forward* and *reverse* and formulated models for each of the algorithms operating in heterogeneous systems. They conducted many interesting tests on the models, e.g., the effects of varying thresholds, the impact of changing probe limits, and determining the optimal response times over a large range of loads and delays.

In general, load balancing in transaction processing systems is formulated as a mathematical programming [64] or network flow problem [37], and solved by optimizing some performance index such as overall response time [84] or overall delay [85].

The direct numerical methods are best applicable to systems where jobs belong to one class and the local network can be modeled by a single queue. Load balancing in more general distributed systems with multiple classes, site constraints and a general interconnecting network is formulated as a nonlinear, multicommodity flow routing problem.

Load balancing in distributed systems has also been analyzed by modeling the system as a set of N parallel queues which represent the resources and a central dispatcher which distributes load among queues [77, 86, 87, 88, 84]. These studies include the analysis of static and dynamic policies. In a central server model [89] the processing sites itself can be modeled as a network of queues.

2.3 Current Prediction Methods

Currently available performance prediction methods for distributed systems fall into two general categories. Methods of the first category employ *simulation* tools to construct and run a system model. The second category consists of approximate *analytic* techniques. These methods employ either standard queueing techniques or graphical methods for behavior analysis and estimating various performance measures.

2.3.1 Simulation Methods

Simulation models are often preferred over analytical ones because of their flexibility and ability to express fine details. Analytical techniques are generally applicable to a

narrow range of system architectures and specific types of program structures. Simulations, on the other hand, are capable of handling complex architectures and all program structures. Model enhancement is simple and new features such as communication patterns and load balancing policies can be easily incorporated. Simulations also provide an environment to analyze the transition system behavior.

General-purpose simulation packages, such as IBM's RESQ [90] or UCLA's SARA [91] which have built in facilities for gathering and analyzing performance statistics are now available. Different packages can vary considerably in their modeling primitives, model definition languages, and performance measurements facilities. Thus, a model usually has to be re-designed and re-implemented in order to run in a different simulation environment. It is up to the modeler to determine what level of detail to implement in a model and how to properly abstract the pertinent characteristics of the actual system being evaluated. In deciding on the latter issues, one must consider what performance measures are being sought and what accuracy level is required.

In some cases, general-purpose packages may be intolerably slow in simulating very detailed models or may not be equipped to provide all of the desired performance measures. For these reasons several research groups have developed their own special-purpose hardware and software simulators to model specific distributed systems [92]. A given special-purpose simulator can usually represent only a particular system architecture, although some are parameterized to allow the modeling of different

configurations of the same basic design.

Trace driven simulators are popular in studying system dynamics. Zhou [93] conducted a trace-driven simulation study of homogeneous distributed systems. Seven load balancing algorithms were simulated and their performance was compared. Job traces from an actual machine were used instead of probability distributions to describe the arrival times and resource demands of the jobs. Darema-Rogers [94] generated execution traces of a prototype parallel program in a centralized environment, which were used to derive estimates of its performance in a true parallel environment. A method was proposed to scale up the results for larger programs.

On-line comparison of different policies and system functionalities is also possible with simulations. Depending on the system characteristics and application behavior, novel efficiency and performance metrics can be easily designed. Wong and Morris [83] simulated distributed systems and defined a performance metric called the Q-factor which summarizes both overall efficiency and fairness of an algorithm. The computed factor allows algorithms to be ranked by performance. Recently, Kremien and Kramer [38] also compared adaptive algorithms in distributed system using simulations. They considered the delay characteristics in the distributed environment and estimated various performance and efficiency measures.

Simulations are extensively used to validate the results obtained from analytical methods. It is common to make assumptions when modeling systems using analytical methods. The accuracy of the results and validity of assumptions are

usually established via simulations. One example is the common assumption of exponential service demand in queueing network models; realistically these times are not exponentially distributed as assumed. To demonstrate the efficacy of the mathematical model, systems are simulated without the exponential assumption and the results are compared against the analytical ones.

2.3.2 Analytical Methods

Simulations suffer from being expensive, time consuming and slow. Simulations involve not only a significant cost of developing a model, but also a large amount of expensive CPU time for every run of the simulation. Also, even if a single parameter is changed in a model, a complete, new set of simulation runs is required to determine the new performance statistics. Furthermore, the type of computing environment necessary to support most general-purpose simulation packages is usually very sophisticated and expensive. Special-purpose simulators are generally more efficient, but their development is very costly and each is able to model only a specific system architecture.

Considering the above properties of simulation, one can appreciate the importance of analytical methods. They are concise, efficient and expressed through a language (mathematics) which is precise and well comprehended. We categorize analytical prediction methods as queueing and graphical methods. Queueing methods include queueing networks and Markov decision theoretic techniques. Petrinets and

graph models fall in the second category. Prediction methodologies which combine modeling techniques from the two categories to estimate various performance measures also exists. We also discuss a few methods which do not fall in either of the two classes.

2.3.2.1 Queueing Methods

Of all the analytical methods, queueing methods are the most common performance prediction methods because of their simple applicability and ease of computation. Queueing theoretic techniques are suitable to model message passing distributed systems and packet switching networks because the jobs or packets can be easily represented as customers in queueing environments. The mathematical foundations of queueing models allow easy computation of various measures.

Researchers have used closed, open or mixed queueing networks [17, 95] with one or several job classes for modeling, analysis and synthesis of distributed systems. Gelenbe and Mitrani [17] used closed-form analytical solutions and approximate methods to model multiprogrammed computers. Gerla and Silva [72] considered multiple class mixed queueing network models to represent distributed systems. In their model, interactive jobs run on local sites and the batch jobs can run on any site. The decision of running a job on a site is independent of the state of the distributed system. Jobs in their model follow a particular chain; the interactive jobs follow open chains and batch jobs follow closed ones. The response time (delay) of each chain is calculated

using nominal throughput. The normalization constant is computed for mixed chains.

Akyildiz [96] modeled the synchronization of processes in distributed systems using product form queueing networks [97]. Processes communicate with each other via buffers using SEND and WAIT operations. A process which cannot execute a synchronization operation successfully goes into the blocking state. The analysis of the distributed system is conducted hierarchically using two models - a global model and a process communication model. The distributed system is modeled (global model) as a closed queueing network with finite station capacities. The blocking probabilities and the blocking times of processes are computed from the process communication model and are used as input parameters for the global model which can then be solved by appropriate existing product form network methods. The method provides exact results for two station cases and approximate results for multiple station cases.

Mirchandaney *et al.* [79] formulated queueing theoretic models for heterogeneous systems under the assumption that the job arrival process at each node is Poisson and the service times and job transfer times are exponentially distributed. The models are solved using the Matrix-Geometric solution technique. The results of the models are compared with the M/M/1, random assignment and the M/M/K models. Nelson *et al.* [70] developed a performance model for a parallel processing system. In their model, jobs wait in a central queue and are split into independent jobs with exponential execution times which synchronize at the end. Whenever a processor becomes idle, it executes the next waiting task from the queue. The job

response time consists of three components: queueing delay, service time, and synchronization delay. An expression for the mean job response time is obtained for this centralized parallel processing system. Centralized and distributed parallel processing systems (with and without job splitting) are considered and their performances are compared. Their methodology is useful for comparing different system models.

Heidelberger and Trivedi [98, 99] used analytic queueing models to predict performance for programs with internal concurrency. An approximate method to model the programs is developed in which a parent spawns off two or more statistically identical asynchronous children tasks. They also considered the case in which the parent task, after spawning off the children tasks, waits for their completion before continuing.

Gelenbe [100] used a queueing network model to analyze the performance of the Connection Machine. Special emphasis on estimating the effect of its interprocessor communication is made. A model of network architecture, including the NEWS and ROUTER networks, is used to compute the slow-down induced by message exchange between processors. Locality of the message exchanges is modeled by sending probabilities, which depend on whether a message is sent by a processor to another processor placed on the same NEWS network, on the same ROUTER, or at a remote location which is only accessible via the ROUTER network. The performance degradation of the Connection Machine as a function of the communication and architecture parameters is derived.

Markov and semi-markov theories have also been used to estimate the performance of distributed systems. Such methods are less attractive then queueing networks because the analysis becomes complex for large systems. Shin *et al.* [101] formulated the problem of controlling resources in a distributed system using Markov decision theory. The control variables considered were general and were related to system configuration, repair, diagnostics, files, and data. A reward function is optimized in search for optimal control strategy. Two algorithms for resource control in distributed systems are derived for time-invariant and periodic environments.

2.3.2.2 Graphical Methods

Graphical methods are very popular in modeling the inter-dependencies and communications among different processing units of distributed system. Parallel programs for multiprocessing systems, as discussed in section 2.2.2, can be represented as a set of tasks which can be optimally allocated to processing units. The interdependencies among the tasks are represented through graph models. Graph models in conjunction with queueing models have been used to predict the performance of parallel programs in distributed systems. For representing program behavior, graph-based techniques are used, while queueing networks are utilized for modeling system architectures. The solutions of both techniques are combined to estimate the performance of a distributed system in executing some selected applications.

Mak and Lundstrom [35] describe a method for predicting performance of a

class of parallel computations running on distributed systems. A parallel computation is modeled as a task system with precedence relationships expressed as a series-parallel directed acyclic graph. Resources in the concurrent system are modeled as service centers in a queueing network model. Using these two models as inputs, the method outputs predictions of expected execution time of the parallel computation and the concurrent system utilization.

A hybrid methodology proposed by Mohan [102] combines analytical models with Monte Carlo simulations. Parallel programs are adequately represented by task graphs (also known as precedence graphs). Task execution time is obtained through the resolution of a queueing network which considers contention for shared resources by the tasks which execute concurrently. The performance results of the execution of several tasks is integrated through Monte Carlo simulation which considers the precedence relationships described in the task graph.

Menasce and Barroso [63] presented a methodology to obtain the execution time of a parallel program composed of several concurrent tasks. The precedence relationship between the tasks is described by a task graph. The execution time of each task is shown to be given by the sum of two terms: a fixed one and a nondeterministic term which is a function of the contention for shared memory. The execution time of a task depends on the shared memory access delay, that is in turn dependent on the pattern of memory access delay, which depends on the pattern of memory requests, the network contention, and on memory modules contention. An algorithm is pre-

sented to show how to deal with these dependencies in order to obtain the execution time of a given parallel application.

Thomasian and Bay [62] propose a two-level analytic model to obtain the execution time of systems composed of parallel tasks. The first level model is based on the resolution of a Markov chain whose states are the feasible combinations, according to task dependencies, of concurrent tasks. The transition rates are obtained by solving queueing network models for each system state. Kapelnikov *et al.* [103] proposed a methodology for estimating the execution time of programs running on distributed multicomputer systems. Their approach is also based on the use of queueing network models, Markov processes, and graph models of computation.

Petrinets have also emerged as an important performance prediction method for a wide range of applications and computer systems wherever the mathematical modeling of dynamic systems is essential [104]. Sheih *et al* [105] addressed some fault tolerant issues pertaining to hierarchical distributed systems. They investigated centralized and distributed fault-tolerant schemes, using Stochastic Petrinet (SPN), by considering the individual levels in a hierarchical system independently. Cases were pointed out where one strategy performed better than the other. They also studied the effect of integration on the fault-tolerant strategies of various levels of a hierarchy.

2.3.2.3 Other Methods

Methods other than queueing and graphical methods have also been used to model and estimate performance of distributed systems. One of the distributed memory models based on integer programming was developed by Chen and Akoka [106] for the optimization of distributed information systems. They considered issues such as the distribution of processing power, the allocation of programs and databases, and the assignment of communication line capacities. Their model considers the return flow of information, as well as the dependencies between programs and databases. An algorithm based on the bounded branch and bound integer programming technique, was developed to obtain the optimal solution model. The model focused more on distribution and allocation of work among resources rather than predicting performance.

Norton and Pifister [107] proposed a methodology for the prediction of the performance of IBM RP3 type multiprocessors. This methodology takes into consideration some characteristics of the application code and does not explicitly consider either the mapping of concurrent tasks into the processors or any kind of synchronization delay. Their model is based on an iterative algorithm which computes the delays due to access to shared memory for a given processing rate. This memory access delay is used to compute a new value for the processing rate. The iterations continue until these two values converge for a given tolerance.

Gilio [108] proposed a virtual processor model for parallel programming on distributed systems. The notions of featherweight processes and featherweight communication are introduced. The programming model is a higher level of abstraction to

make parallel systems programmable and convenient to use. The level of abstraction was proposed to allow application programs to be written independently of the actual size of the system, not through virtual shared memory but by virtualized processors. The implications of the implementation of the virtual processor model on the system architecture, the operating system, and the compilers are discussed.

CHAPTER 3

THE PERFORMANCE OF A FILE-SERVER MODEL

3.1 Introduction

It is challenging to distribute all physical components, data, process, hardware and control in a distributed computing environment. Our modeling methodology aims at distributing all of them to improve the performance and we advocate step-by-step development of system models. As a first step, queueing network elements are used to model the operation of physical components of a distributed system. The next step considers designing static policies to optimize loads according to long-term traffic trends. As the last step we design and evaluate dynamic algorithms for sudden traffic changes and temporary perturbations.

The model, at this preliminary stage, considers distributing only data (files) among different storage units and operates on a simple principle to execute jobs. The model provides a underlying architecture and frame-work for designing the target system, where data, processes and control could be efficiently distributed . We also describe an efficient and accurate performance prediction method in this chapter, and use it to evaluate the performance of the model.

We represent a distributed and parallel system by the interactions among the following entities: clients, servers, network and information units. *Clients* are any processing units in the system which invoke operations and request access to the

information units. *Servers* are the nodes which service the requests made by the clients and manipulate the information units. *Information units* could be any data unit: file, messages, user-requests, or executing processes.

To predict the performance the information units are mathematically represented as customers of different classes in a queueing network model while servers, clients and the network are modeled as service centers. Performance measures such as the the average utilization of the service centers, their throughput, and the average response time of the system are estimated using the probabilistic queueing model. The *average response time* is defined as the time interval which starts when a client requests some information and ends when the request is serviced. It is the average time a request spends in the system. Short response time is a characteristic of good performance. The response time of a system not only depends on the transmission properties of the interprocess communication primitives, their implementation, and supporting protocol, but also on the manner in which the information units are located, managed and accessed. Cache size and client-server ratio also affect the average response time of the system.

The system we refer to operates as follows: when a user at a client makes a request, the request 'enters' the system and proceeds to receive service at different service centers. During this time the user waits for a response. After some time interval, when the request is satisfied, the user enters a 'think time' and then generates a new request.

We begin, in next section, with a detailed model description. A complete analysis of the system is given in Section 3.3, illustrating how the prediction method may be used to evaluate the performance of a distributed and parallel system. A numerical example is considered in Section 3.4, in which the results obtained from the analytical solution are validated against the simulated ones. In Section 3.5 we develop an approximate method to compute optimal cache size, and in Section 3.6 we describe a method to identify and isolate the bottleneck node. Section 3.7 summerizes this chapter.

3.2 The System Model

3.2.1 System Description

We consider a distributed system with c processors, which we call clients ($c_i, 1 \leq i \leq c$), and m servers ($s_j, 1 \leq j \leq m$), connected via a network (t). The clients request access and usage to a set of files; these files reside at a set of servers. The file is the information unit distributed in the system. Any task or process is generated at some client and require data (one or more files) for execution. Each file resides normally in some server storage; the node where a file resides is referred to as its 'host node'. File copies can be transferred to clients on demand. A file is in one of two states: *free* or *busy*. A file is busy if a copy of it has been transferred to some client; otherwise it is free.

A server receives a file request from a client, processes the request, and transfers a file copy to the client, if the file is free. This action marks the file as busy. Each client has a temporary local memory, the cache, which holds the file-copies obtained by the client. A file in the client's cache is in one of two states: *active* or *non-active*. A file is active if it is currently used by a process executing at the client and is non-active if it still resides in that cache without being used.

Servers and clients communicate via messages. Messages can be of the following three types: *file-request-message, file-active-message* and *confirm-message*, we denote them by M_1, M_2 and M_3 respectively. The messages circulate among different nodes depending upon the availability of the requested file. Formally, any information unit in the system at any instant of time can be defined as: $r_{x,y,z}(x = M_1, M_2, M_3, F; y, z = c_i, s_j, t)$. Here, the subscript x denotes the class of the information unit (file (F) or any message type), y is the source (service center) from which the information unit is coming from, and z is the destination node (service center). When a file is requested by some process at a client, then

1. For a cache miss (*i.e.* when the file-copy is not located in the client's memory), the client (c_i) sends a file-request-message ($r_{M_1,c_i,t}$) to the network (t). The network (t) in response forwards the file-request-message (r_{M_1,t,s_j}) to the appropriate server (s_j). If the file is free, the server (s_j) sends a copy of the file to the client, ($r_{F,s_j,t}, r_{F,t,c_i}$). After receiving a copy of the file, the client sends the confirm-message ($r_{M_3,c_i,t}$), to the network which passes the message (r_{M_3,t,s_j})

to the host server. The sequence of message can be expressed as the following subchain:

$$r_{M_1,c_i,t} -> r_{M_1,t,s_j} -> r_{F,s_j,t} -> r_{F,t,c_i} -> r_{M_3,c_i,t} -> r_{M_3,t,s_j}$$

2. If the requested file is busy at the server (s_j), then the copy of the file resides at some other client $(c_k, k \neq i)$. The server sends a file-request-message $(r_{M_1,s_j,t}, r_{M_1,t,c_k})$ to that client (c_k). If the requested file is in a non-active state at the client c_k, the client transfers the copy of the file to the host server $(r_{F,c_k,t}, r_{F,t,s_j})$. The server (s_j) updates the file and then transfers the copy of the file to the client (c_i). Upon receiving the file the latter responds to the server with confirm-message ($r_{M_3,c_i,t}$ and r_{M_3,t,s_j}). The sequence of messages can be described by the following sub-chain:

$$r_{M_1,c_i,t} -> r_{M_1,t,s_j} -> r_{M_1,s_j,t} -> r_{M_1,t,c_k} -> r_{F,c_k,t} -> r_{F,t,s_j} -> r_{F,s_j,t} -> r_{F,t,c_i} -> r_{M_3,c_i,t} -> r_{M_3,t,s_j}$$

3. If the file is active at the client (c_k), the client issues a file-active-message $(r_{M_2,c_k,t})$ for the server who in turn passes the message to the client who initially requested the file. The sub-chain of messages is:

$$r_{M_1,c_i,t} -> r_{M_1,t,s_j} -> r_{M_1,s_j,t} -> r_{M_1,t,c_k} -> r_{M_2,c_k,t} -> r_{M_2,t,s_j} -> r_{M_2,s_j,t} -> r_{M_2,t,c_i}$$

The client in the last case does not have to reply back with a confirm-message since the host server is aware of the file being active.

The cache at each client is local memory implemented as a Least Recently Used (LRU) stack. The *least recently used* file is at the bottom of the stack, and the most recently used is at the top. If the local memory is full, space for the new incoming file-copy is made by removing one or more files from the memory. The least recently used files are transferred to the respective host servers. If the client sends one or more files ($r_{F,c_i,t}$), the following sub-chain should be added to the two sub-chains in case 1 and 2,

$$r_{F,c_i,t} -> r_{F,t,s_j}$$

All nodes of the system - servers, clients and the network - handle files (r_F) and messages (r_M)[1]. A server handles the file-request-message (r_{M_1}) on FCFS basis for the different files and in the time-stamp (TS) order for the same file. When a file is busy at a server, the server forwards the file-request to the client which holds the file-copy. Any subsequent requests for the same file must wait at the host node until the file copy reaches the requesting node. In the meantime, the node services the other requests in FCFS order.

For every file, either no file-copy exists, or one exists and the host node has the address of the client who has it. We avoid having more than one extra copy of a file. This is a pragmatic choice which is close to what would be expected in practical systems.

File placement and movement are transparent to the users. At this stage of

[1] r_M denotes messages in general, without class distinction and without particular source and destination node.

modeling we do not allow dynamic creation and deletion of the files. All the processes and tasks are executed locally on clients. To locate a file in the system a simple but efficient approach of static maps [109] is used. In static mapping, part of the file name is used to identify the server. The simplest approach is to have a number as a suffix (or a prefix) for each file name. The number maps to a particular server and is stored in a table on the client's local memory.

A file request may follow any one of the subchains described above. The chance of a request following a particular subchain depends on three conditions:

1. The state of the file at the server (free or busy);

2. The state of the file copy at the client (active or non-active); and

3. The availability of memory space at the client.

Let p be the probability that a file is in busy state at the server, τ be the probability that a file is in non-active state and π be the probability that the cache is full. The flow of the information units in the system can be represented as shown in Figure 2.

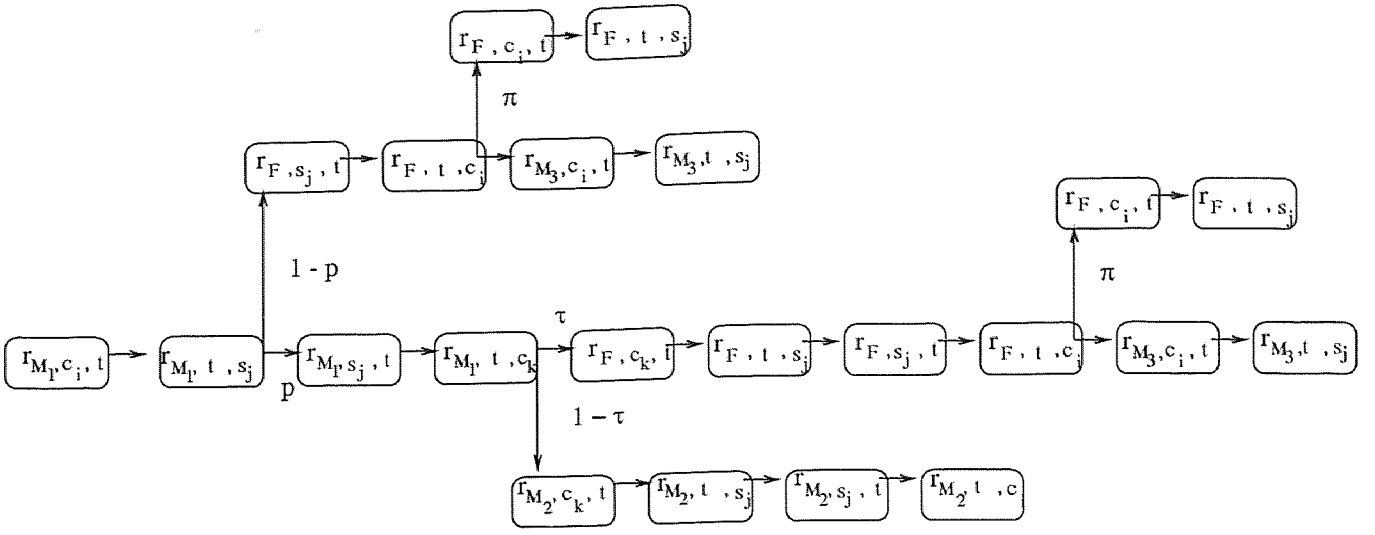

Figure 2: Flow of information in the distributed systems

3.2.2 The Queueing Network Model

The distributed system as described above is modeled as a network of queues formed by a collection of servers, clients and network. (Figure 3).

The physical nodes are represented as service centers in the queueing network model. Each information unit (user-requests, files, messages, etc.) is modeled as a customer of a particular class. Customers entering the system change classes as they circulate among service centers and follow one of the chains shown in Figure 2. The level of modeling is analogous to the multi-programming and multi-processing description of computer systems as proposed by Gelenbe and Mitrani [17], in which computer resources are modeled as service centers and computing jobs are modeled as customers. Modeling of distributed and parallel systems at this level is appropriate

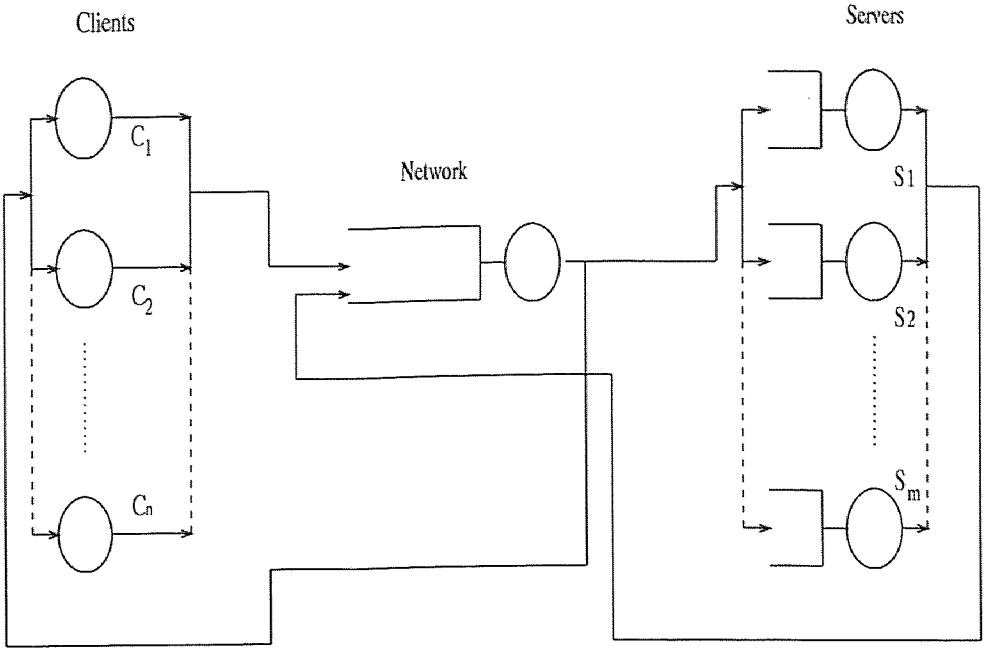

Figure 3: The queueing network model

because performance issues can be studied without worrying too much about the details of the system.

Based on the queueing discipline and the service demands the network, clients and servers are represented by different types of service centers in the queueing model [17]:

1. *Network: Single processor-shared server*

   For the network, customers of all classes (messages and files) constitute an arbitrary number of packets. All packets are served on FCFS basis. All customers are served in parallel and share the one common processor.

2. *Servers: Multiple-server Center*

All servers are modeled as one service center, where jobs wait in a common queue to be served. If no more than k jobs request service from a k-server service center, all jobs will receive service immediately without queueing. If there are more than k jobs, k of them can be serviced at one time, and others have to wait in the queue.

3. *Clients: Infinite-server*

   This service center models the group of clients, where the number of clients is always greater than or equal to the number of jobs. Thus, no job visiting this service center will have any queueing delay.

The queueing network as described above satisfy product form requirements [97, 95, 110]. The numerical values for the specific measures of system performance, such as node utilizations, throughputs, average response times, etc., can be extracted from the solution for the stationary distribution state of the network queueing model [111, 112]. We discuss the computation of performance measures in the next section.

3.3 ANALYSIS

Let S be the state of the system and N be the total number of customers in the system. If N = $(n_1 + n_2 + n_3)$, where n_1, n_2 and n_3 are the number of customers at servers, the network and clients respectively, then the steady state distribution can be obtained by summing p(S) over all states S which yields n. Let $1/\mu_{ir}$ be the average service time of class r customer at the network and client node, and $1/\mu_{in_i}$ be the mean

service time of the exponential distribution at the server node, when n_i customers are present. Assuming the system is closed with respect to all customer classes, and the server, the client and the network nodes are service centers as described in the queueing model, the equilibrium probabilities at steady state are given by [110]:

$$p(N) = (1/G)p_1(n_1)p_2(n_2)p_3(n_3) \tag{1}$$

In this equation marginal probabilities $p_i(n_i)$ are defined as follows:

for the server node

$$p_1(n_i) = \left(\frac{\sum_r e_{ir}}{\mu_{in_i}}\right)^{n_i} \tag{2}$$

for the network node

$$p_2(n_i) = \left(\sum_r e_{ir}\mu_{ir}\right)^{n_i} \tag{3}$$

for the client node

$$p_3(n_i) = \frac{1}{n_i!}\left(\sum_r \frac{e_{ir}}{\mu_{ir}}\right)^{n_i} \tag{4}$$

The service time distributions are arbitrary Coxian for the client and the network nodes, and assumed to be exponential for the server node. The quantity e_{ir} is proportional to the total arrival rate of class r jobs into node i and is interpreted as the relative arrival rate of class r customers to the service center i [17].

The existence of the steady state distribution for a closed network depends on

the solution of the following set of flow equations:

$$e_{ir} = \sum_{j,s} e_{js} P_{js,ir} \tag{5}$$

(1/G) in the equation (1) is a normalizing constant which must be calculated over all possible states [113, 114]:

$$G = \sum_{n_1+\cdots+n_N=K} \left(\prod_{i=1}^{N} p_i(n_i)^{n_i} \right) \tag{6}$$

The flow equations (5) are easily obtained from Figure 3 as follows:

$$e_{c,r_{M_1}} = e_{t,r_{M_3}} + e_{s,r_{M_2}} + e_{t,r_{M_3}} + e_{t,r_F}$$

$$e_{c,r_{M_3}} = e_{t,r_{M_F}} \qquad e_{t,r_{M_3}} = e_{c,r_{M_3}} \qquad e_{s,r_{M_3}} = e_{t,r_{M_3}}$$

$$e_{c,r_{M_3}} = e_{t,r_{M_F}} \qquad e_{t,r_{M_3}} = e_{c,r_{M_3}} \qquad e_{s,r_{M_3}} = e_{t,r_{M_3}}$$

$$e_{c,r_F} = \pi e_{t,r_F} \qquad e_{t,r_F} = e_{c,r_F} \qquad e_{s,r_F} = e_{t,r_F}$$

$$e_{s,r_{M_1}} = p e_{t,r_{M_1}} \qquad e_{t,r_{M_1}} = e_{s,r_{M_1}} \qquad e_{c,r_{M_2}} = (1-\tau) e_{t,r_{M_1}}$$

$$e_{t,r_{M_2}} = e_{c,r_{M_2}} \qquad e_{s,r_{M_2}} = e_{t,r_{M_2}} \qquad e_{t,r_{M_2}} = e_{s,r_{M_2}}$$

$$e_{c,r_F} = \tau e_{t,r_{M_1}} \qquad e_{t,r_F} = e_{c,r_F} \qquad e_{s,r_F} = e_{t,r_F}$$

$$e_{t,r_F} = e_{s,r_F} \qquad e_{c,r_{M_3}} = e_{t,r_{M_F}} \qquad e_{t,r_{M_3}} = e_{c,r_{M_3}}$$

$$e_{s,r_{M_3}} = e_{t,r_{M_3}} \qquad e_{c,r_F} = \pi e_{t,r_F} \qquad e_{t,r_F} = e_{c,r_F}$$

$$e_{s,r_F} = e_{t,r_F}$$

One solution can be obtained by setting $e_{c,r_{M_1}} = 1$, which gives

$$e_{c,r_{M_1}} = 1 \qquad e_{t,r_{M_1}} = 1 \qquad e_{s,r_F} = (1-p) \qquad e_{t,r_F} = (1-p)$$

$$e_{c,r_{M_3}} = (1-p) \qquad e_{t,r_{M_3}} = (1-p) \qquad e_{s,r_{M_3}} = (1-p) \qquad e_{c,r_F} = \pi(1-p)$$

$$e_{t,r_F} = \pi(1-p) \qquad e_{s,r_F} = \pi(1-p) \qquad e_{s,r_{M_1}} = p \qquad e_{t,r_{M_1}} = p$$

$$e_{c,r_{M_2}} = (1-\tau)p \qquad e_{t,r_{M_2}} = (1-\tau)p \qquad e_{s,r_{M_2}} = (1-\tau)p \qquad e_{t,r_{M_2}} = (1-\tau)p$$

$$e_{c,r_F} = \tau p \qquad e_{t,r_F} = \tau p \qquad e_{s,r_F} = \tau p \qquad e_{t,r_F} = \tau p$$

$$e_{c,r_{M_3}} = \tau p \qquad e_{t,r_{M_3}} = \tau p \qquad e_{s,r_{M_3}} = \tau p \qquad e_{c,r_F} = \pi\tau p$$

$$e_{t,r_F} = \pi\tau p \qquad e_{s,r_F} = \pi\tau p$$

Now the marginal probabilities of network, server and client nodes can be computed from the above flow equation as follows:

1. For network node, from (2), $p_1(n_1) = \left(\frac{\sum e_{t,rm}}{\mu_a} + \frac{\sum e_{t,r_F}}{\mu_b}\right)^{n_1}$ where

   $1/\mu_a$ is the average time to transmit one message (1 packet), and $1/\mu_b$ is the average time to transmit one file (multiple packets). All jobs, messages (r_M) and files (r_F) are treated as packets. The number of packets depends on the file-size in the case of a file (r_F). Summations $\sum e_{t,r_M}$ and $\sum e_{t,r_F}$ are the relative arrival rates of messages and files respectively at the network and are computed from the flow equations.

$$\sum e_{t,r_m} = 2 + 2p - p\tau \qquad \sum e_{t,r_F} = (1-p) + \pi(1-p+\tau p) + 2\tau p \qquad (7)$$

2. For clients, from (3), $p_2(n_2) = \frac{1}{n_2}\left(\frac{1}{\alpha} + \frac{\sum e_{c,rm}}{\gamma} + \frac{\sum e_{c,r_F}}{\beta_1}\right)^{n_2}$ where

   $1/\alpha$ is the average think time, and $1/\beta 1$ is the average time to send or receive

a file. $1/\gamma$ is the average time taken to service a message (r_M) by both a client and a server. These times includes the network access time. From the flow equations:

$$\sum e_{c,r_m} = (1-p) + (1-\tau)p + \tau p \qquad \sum e_{c,r_F} = \pi(1-p) + \tau p + \pi\tau p \quad (8)$$

3. For server node, from (4), $p_3(n_3) = (\sum e_{s,r_M} + \sum e_{s,r_F})^{n_3} \frac{1}{\mu_{n_3}}$ where $1/\mu_{n_3}$ is the average time needed to service a customer and depends on the number of customers (n_3). From the flow equations:

$$\sum e_{s,r_M} = 1 + p \qquad \sum e_{s,r_F} = (1+\pi)(1-p+\tau p) \quad (9)$$

The normalizing constant is given by:

$$G = G_3(N) = \sum\sum\prod (p_1(n_1)p_2(n_2)p_3(n_3)) \quad (10)$$

since N = n_1 + n_2 + n_3 (closed system)

$$G = \sum_{n_2=0}^{n-n_1} \sum_{n_1=0}^{n} \prod (p_1(n_1)p_2(n_2)p_3(N-n_1-n_2)) \quad (11)$$

The throughputs, utilization factors and average response times at node i are

calculated in terms of G and e_{ir} [17].

$$\lambda_i = e_i \frac{G_3(N-1)}{G_3(N)} \tag{12}$$

$$U_{ir} = \frac{\lambda_{ir}}{\mu_{ir}} \quad ; \quad U_i = \sum_r U_{ir} \tag{13}$$

$$W_{ir} = K_r/\lambda_{ir} \; - \; 1/\mu_{ir} \tag{14}$$

K_r are the number of clients of class r (in a heavily loaded system, when the clients are busy all the time, jobs can be identified with clients) [17]. The response time W_{ir}, for a class r job is defined as the interval between the job leaving its terminal (the user presses "carriage return") and returning to it (the keyboard unlocks).

All the servers are considered identical in their behavior, with equal probability to service a request. Thus, if q is the probability of locating a file at any server and there are three servers, the probability of finding a file at one server is q/3. Moreover, if T is the number of total files in the system, there are T/3 number of files assigned to each server.

To calculate steady state probabilities p, π, and τ, we assume that only one file is active at each client. For c clients, each one of them having F number of files in its cache, the probabilities can be approximated as follows:

1. The probability that a file is busy at a server: p = (c*F)/T

   where the average number of files at a client's cache:

   F = cache-size / average-file-size.

2. The probability that the cache is full: $\pi = 1 - \sum_{k=c*F-F}^{c*F-1} (\frac{T}{c}) p^k (1-p)^{T-k}$

3. The probability that a file is not active at a client:

   τ = 1 - probability file active at client = 1 - 1/F

3.4 Results and Validation

3.4.1 Numerical Example

The prediction method described above is general and can be used to estimate performance of a system with any number of servers, clients and files. To demonstrate the efficacy of the method we compute numeric values of the performance measures of a distributed system with three clients and three servers. We consider a total of 600 files in the system, 200 assigned to each server. A small number of files are chosen to introduce more conflict among different file requests. The average file size is assumed to be 3000 bytes. The network transmits packets of 1000 octets (bytes) at a rate of 10 Mbps. The disk transfer rate at a server and the cache transfer rate at a client are 1 Mb/s and 40 Mb/s respectively.

We are interested in the behavior of the system under heavy loads. The performance of the servers, of the network and of the complete system is measured by increasing the rate at which the user submits file requests, i.e., decreasing think time.

As the clients are loaded more heavily by the user's file requests, the utilization of a server and that of the network increases at a constant rate, as shown in Figure 4.

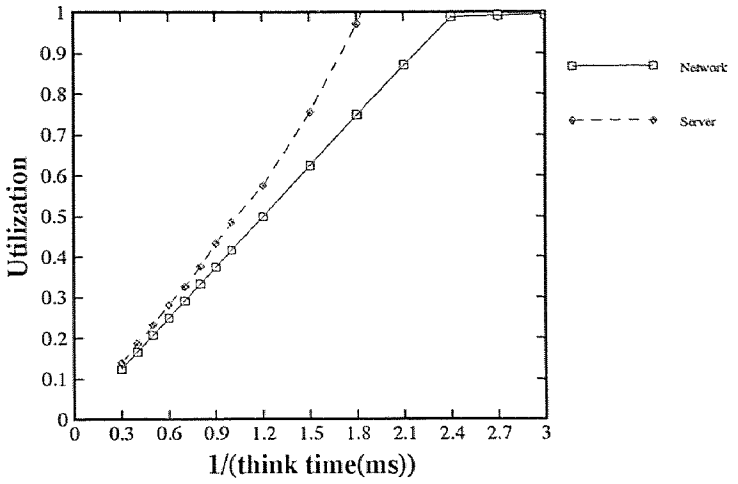

Figure 4: Network and server utilization increases as user think time decreases

From the curves we make specific, although approximate, predictions about the network and server performance. In Figure 5 we show the affect of think time on the *average node response time*, which is the average over all requests made to a node.

With a think time of 2 milli-seconds, we have network utilization of about 15%. Decreasing users' think time by 60% raises the network utilization to 55%. In other words, 60% more file requests will raise the throughput by nearly 200%, with only 20% degradation in the average node response time (Figure 5).

However, decreasing think time by a further 60% increases the network utilization to 80%, thus raising the throughput by only another 45%, but the average node response time increases by nearly 83%. Hence, for three servers and three clients, to achieve good performance the network utilization should be restricted to 60%. As

the client or server population increases so does the utilization and response time for the network.

Further observation from Figures 4 and 5 shows that even though the network is much faster than an individual server (the response time of network is much less than that of an individual server), the utilization of the two is almost the same. This is due to inherent parallelism in serving the file requests by the servers.

The average system response time, which is defined as the interval between the file request arriving at a client and returning back to it, increases as the rate of file requests arriving at the client increases. The results of the model with two and three servers are shown in Figure 6. Decreasing the user think time by 60% for the model with two servers increases the response time by 2.5%. A further 60% reduction in think time results in 46% degradation in the system average response time. Any further decrease in think time will increase the response time asymptotically.

Increasing the number of servers improves the response time of the system. The system performance increases by nearly 14% when the number of servers increases from two to three. Each additional server speeds up the request processing. However, the network capacity limits the speed-up achieved by adding more servers.

3.4.2 Validation

The performance prediction method described in the last section has been used to estimate the behavior of a distributed system. This section presents the results of

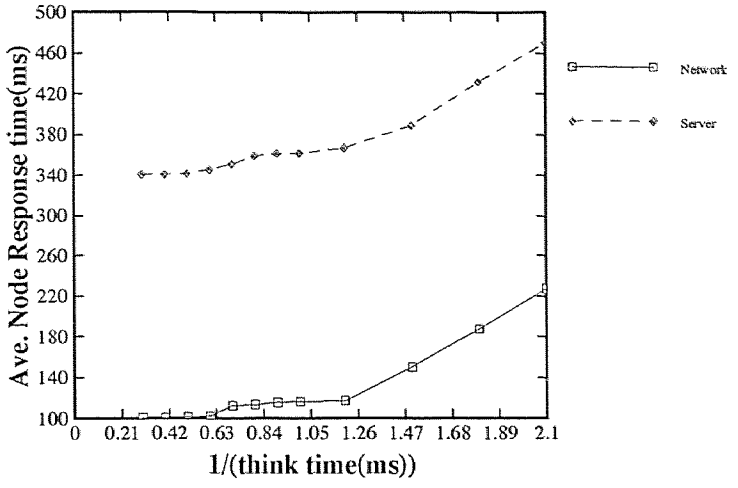

Figure 5: Average node response time of server and network node as a function of think time

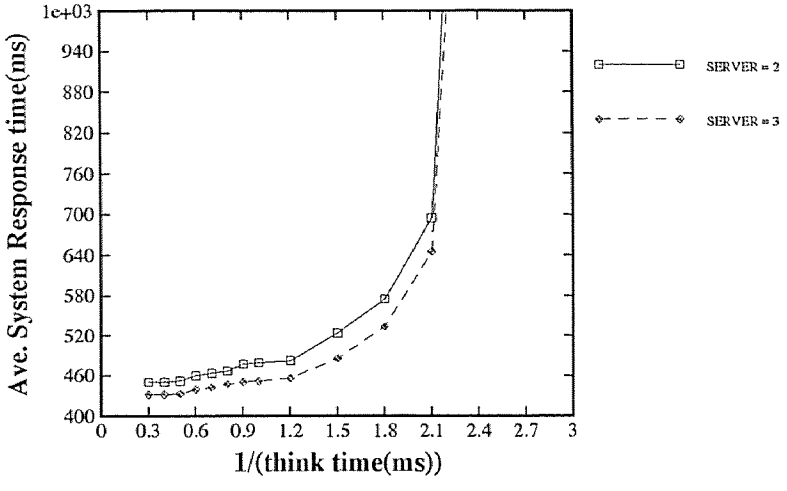

Figure 6: The average system response time as a function of think time

the evaluation of the prediction method in terms of accuracy of its prediction. The accuracy of the method is established by comparing the estimates of the prediction method to the statistics collected from detailed simulations.

During each simulation run, queue-lengths, marginal probabilities (probability that n customers are at a node), utilizations and throughputs of system nodes are collected. The queueing network model and the simulation model differ in following respect:

1. In the queueing network model, the service time distributions at the server node to service a file request are assumed to be exponential; realistically, however, the service times depend on the file size. For simulation purposes, the service times are uniformly distributed between 2000 to 4000 bytes, average file size being 3000 bytes.

2. The queueing discipline at the server node is considered to be FCFS for the file requests. In simulations, some modifications have to be made in order to make the system work correctly. Whenever a requested file is busy at the server, the server forwards the request message to some client. Any other requests for the same file must wait at the server until the file copy reaches the requesting client. In the meantime, the server services all the other file requests on FCFS basis. Hence, server deviates from FCFS discipline for file requests requesting the same file.

3. To obtain the numerical values for marginal probabilities in the analytical

model, steady state probabilities (p, π and τ) have to be approximated as discussed in Section 3.3. However, simulations do not use steady state probabilities to direct the flow of customers. Initially all the files are free at their respective "host" servers, and client memories are empty. As the simulation proceeds, clients request files from the server, and the server services the requests depending on whether the file is busy or free. The simulation statistics were obtained from 6000 file requests from each client.

4. In the queueing network model, routing information is not required. In simulation, routing information is required in order to simulate the transition of customers from one node to another.

The marginal probabilities of messages (r_M) and files (r_F) for each node are considered for comparing the results of the simulations with the analytical model. Analytically the marginal probabilities are calculated using equation (2), (3) and (4). Queue-length statistics yield these probabilities for simulations. The degree of fit between the mathematical results and simulation model output is the key for validating the results.

1. *For Server Node:* In both models, simulation and analytical, all the servers are identical in terms of operating speed, number of files allocated and other operating behavior. Table 1 shows the analytical and simulation marginal probabilities, P(n), where n is the number of customers at the server.

Table 1: Comparison of analytical and simulation values of marginal probabilities for each server.

n	Analytical	Simulation		
		Server 1	Server 2	Server 3
0	0.507	0.507	0.503	0.506
1	0.194	0.202	0.191	0.200
2	0.109	0.108	0.099	0.101
3	0.050	0.054	0.052	0.053
4	0.027	0.031	0.029	0.033
5	0.011	0.012	0.010	0.009

n is the number of customers at the server node.

Figure 7 depicts the degree of fit among the numerical values.

2. *For Network node:* The network treats each file and message as multiple packets. If packets are considered as customers in the simulation model instead of as files and messages, the network resembles, in its behavior, a node of an open-network system [110] as opposed to a closed one. The marginal probabilities from the simulation model are compared against the ones from the analytical open-model. Table 2 presents the marginal probabilities, P(n), for the network, where n corresponds to the number of packets.

   The Figure 8 shows the degree of fit between numerical values.

3.5 Approximate Model For Cache Size Analysis

Due to the interdependence of the queues at the network and servers, it is difficult to obtain an exact solution to compute optimum cache size at clients. An approximate

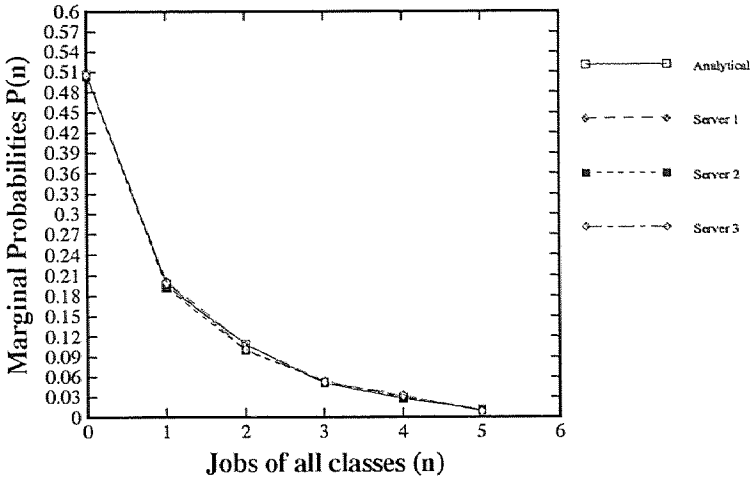

Figure 7: Comparison of analytical and simulation values of marginal probabilities for each server.

Table 2: Marginal probabilities for network compared with simulation results.

n'	Analytical	Simulation
0	0.089	0.080
1	0.019	0.019
2	0.007	0.004
3	0.002	0.001
4	0.001	0.000
5	0.000	0.000

n' is the number of customers at the network node.

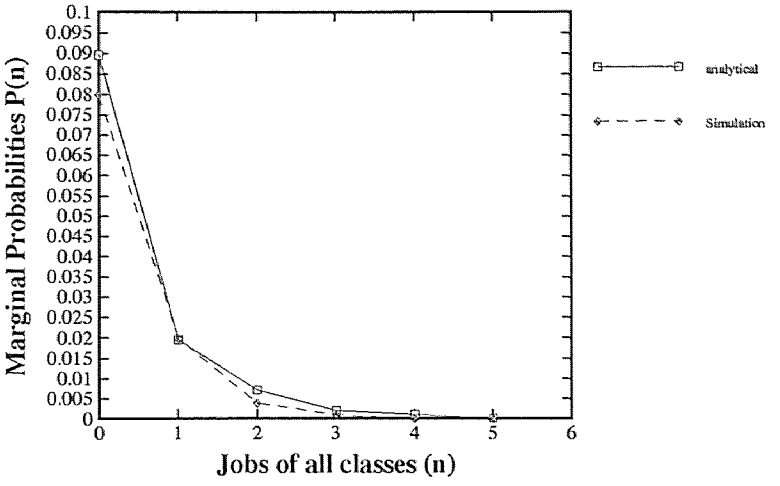

Figure 8: Marginal probabilities for network compared with simulation results.

method is used for estimating optimal cache size and its effect on the performance of the system.

First we analyze how cache size affects the performance. Each client of the 'client node' generates a file request which enters the system and moves around from station to station (clients are service stations also) according to transition probabilities, changes classes and eventually returns to the client as a file. The group of clients is no longer modeled as 'server-per-job' node in an approximated model, but instead each client is treated as a single server with an M/G/1 queue, as shown in Figure 9. Decreasing the think time, hence increasing the number of user-requests per unit time, will NOT assign a separate client (server-per-job strategy), but a user-request will queue up at a client instead. The total time taken to service a file request (response

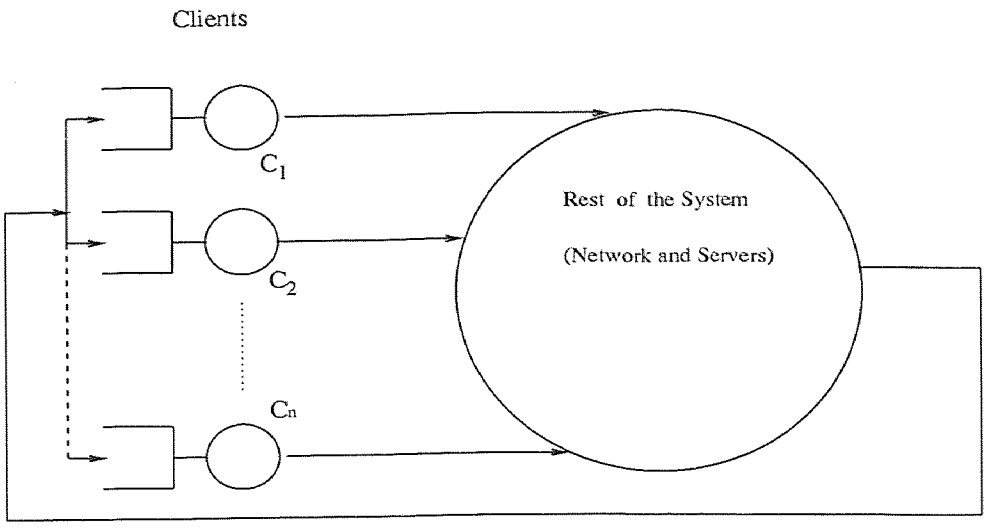

Figure 9: Approximate model; each client is modeled as M/G/1 queue.

time) is approximately equal the total service provided by the network, server and other clients, when the job circulates among different nodes. However, the number of visits to each node and the path taken by the request depends on the steady state probabilities, p and π.

Let Δ_s be the average time to transfer a file from a server to a requesting client, Δ_c be the average time to transfer a file from some client to the requesting client, and Δ_b be the average time to transfer a file from a client back to a server. The average service time to serve a file request, assuming there is no wait at network and server, is

$$E[S] = p\Delta_s + (1-p)\Delta_c + \pi\Delta_b \tag{15}$$

where

$$\Delta_s = 1/\mu_a + 1/\beta_1 + 1/\mu_b$$

$$\Delta_c = 1/\mu_a + 1/\gamma + 1/\mu_a + 1/\beta_2 + 1/\mu_b + 1/\beta_1 + 1/\mu_b$$

$$\Delta_b = 1/\mu_b + 1/\beta_1$$

To see why the above equations are correct, consider a client requesting fileA, which is located at a server, and fileB, which resides at some other client. In the first case, the network passes the message in $1/\mu_a$ time to the host server and the server transfers the file in $1/\beta_1$ time to the network. The network takes $1/\mu_b$ time to pass the file to the requesting client. The total time taken is represented by the equation $\Delta_s = 1/\mu_a + 1/\beta_1 + 1/\mu_b$.

In the second case, the file request is directed to the host server $(1/\mu_a)$. Since the file is located at some other client, the server passes the message to that client $(1/\gamma + 1/\mu_a)$. The client transfers the file $(1/\beta_2)$ to the host server $(1/\mu_b)$, and the server transfers it to the requesting client $(1/\beta_1 + 1/\mu_b)$. Hence the total time, $\Delta_c = 2/\mu_a + 1/\gamma + 1/\beta_2 + 2/\mu_b + 1/\beta_1$.

Since all the waiting in the approximate model is at a client, it is fair to represent each client by a single server (M/G/1). Without considering the customer class distinction, performance of this model can be measured in terms of average number of customers in the queue at the client. Assuming the scheduling discipline for jobs of all classes as FIFO, the average number of jobs waiting at the client can be approximated by the *Pollaczek-Khintchine's* formula [115]:

$$q = \rho + \frac{\theta^2 E[S^2]}{2(1-\rho)} \tag{16}$$

where

θ (throughput of each client) = (throughput of the client node) / number of clients,

throughput for the client node is calculated in terms of G and e_c, Section 3.3,

ρ (the utilization factor) = θE[S], and

E[S^2] is the second moment of the average service time from Equation 15.

Our performance measure depends on the average service time E[S], which is calculated in terms of p and π (Equation 15). These steady state probabilities are obtained in terms of cache size (Section 3.3). Thus keeping all the other parameters the same, varying cache sizes affects the queue length.

The effect of cache sizes was determined by varying cache size from 3000 bytes (representing an average of one file in cache) to 1,000,000 bytes (representing a large number of files). A cache is typically used to reduce the average access time for data storage and retrieval. Besides increasing the cache-hit probability (hit-ratio), cache size affects steady state system probabilities. Simulation results in Figure 10 show that as the size of each cache increases, so does the probability (p) of finding the file busy at the server. This is because, at the steady state, if cache sizes are large more files are located at clients' cache and are marked as busy at the server. If the cache sizes are considerably large, clients behave like servers (data location changes from servers to clients), and servers turn out to be merely communicating nodes of the system. This increases communication delays and hence deteriorates performance. The analytical measure, queue length, validates this assertion. The queue length

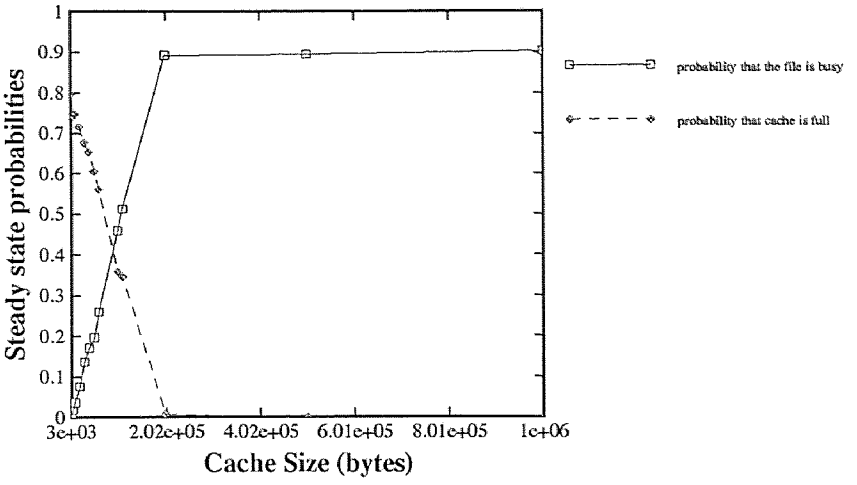

Figure 10: Effect of cache size on probabilities

increases as the cache size increases. Figure 11 shows both analytical results, from Equation (16), and simulation results.

The queue length does not increase indefinitely. For a particular cache-size the curve flattens (Figure 10). This is because the probability π that a cache is full approaches zero (Figure 10).

As the caches are never full, no service is performed to remove the data from the cache, which compensates for extra work done in getting the data from other clients. Figure 12 shows the effect of the cache size on the average system response-time.

Next, from our simulation results we obtain optimum cache size for the given system parameters. The cache of approximately 100,000 bytes is optimum (Figure

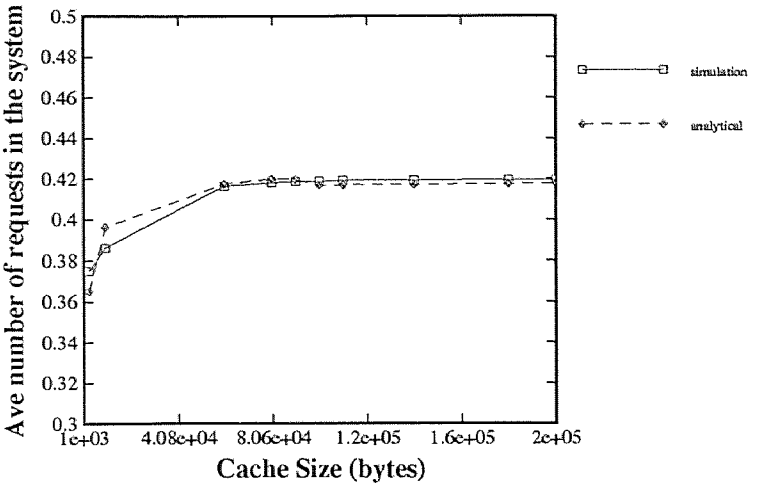

Figure 11: Effect of cache size on performance

12), as smaller cache sizes than 100,000 provide a high response time and larger cache-sizes do not improve the performance.

3.6 Bottleneck Identification

Our approach is to study the system from the terminal's point of view, as shown in the Figure 13. We have a multiple server system serving M terminals (clients). We have a total of K = M customers, each of which generates a job from the terminal at a rate of λ jobs/sec. Each such generated job enters the 'rest of the multiple service station network' (the terminals are service stations also), and moves around from station to station according to the transition probabilities, eventually returning to the terminal at which time the user generates a new job. Each service center (resource) has an

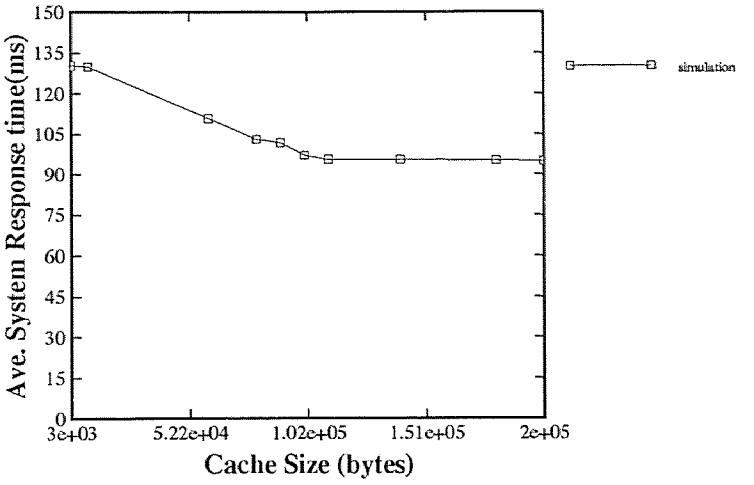

Figure 12: Effect of cache size on the average system response time

arbitrary service time distribution.

Let T be the average response time to pass through the rest of the network and $1/\lambda$ be the average time in the terminal node. The average cycle time is then T $+1/\lambda$ and the system throughput is λ' = M/(T + $1/\lambda$) customers/sec.

Let $\overline{N}$ = E[number of jobs in the rest of the system]

and $\overline{M}$ = E[number of jobs in the terminal node]

By Little's result T = $\overline{N}$ / λ'

since M = $\overline{N}$ + $\overline{M}$; therefore, T = M/λ' - $\overline{M}$ /λ'

For Client node (terminals), for one customer $\overline{M}$ /λ' = $1/\lambda$,

$$T = M/\lambda' - 1/\lambda \tag{17}$$

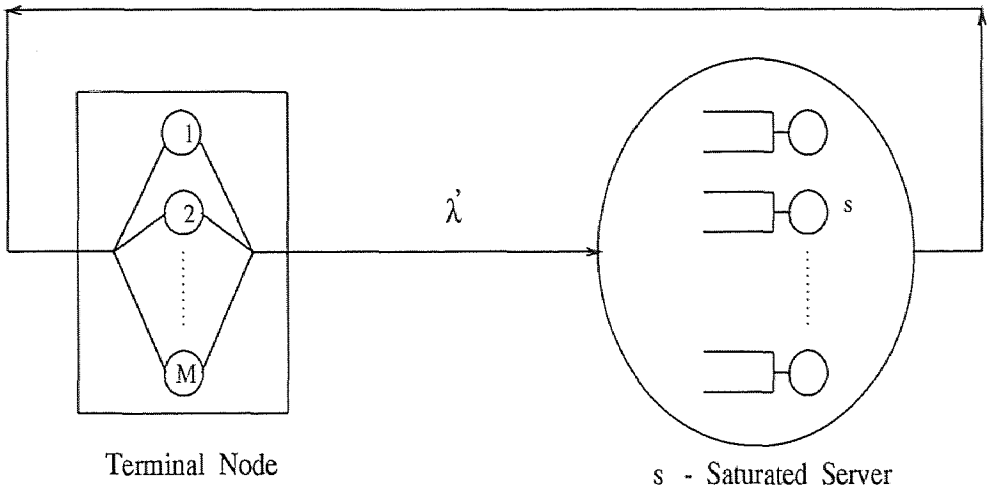

Figure 13: Multi-client multi-server model

Let s be the bottleneck or saturated server in the rest of the network. Then $e_s = \sum_r e_{sr}$, the relative number of visits of all job classes at the s_{th} node, is maximum in the system. If for the client (terminal) node, $e_c = \sum_r e_{cr}$, then the average number of times the bottleneck is visited for each visit to the terminal node is e_s/e_c.

Assuming for $M >> M*$ (M* is the number of clients or terminals which saturates the server s), when the server node is beyond saturation, the output rate of server s is approximately μ_s. Thus the output rate of jobs from the rest of the network can be represented as $\mu_s/(e_s/e_c)$, *i.e.*,

$$\lambda' = \mu_s e_c/e_s \tag{18}$$

Substituting the value of λ' in 17, we get

$$T = \frac{M e_s}{\mu_s e_c} - \frac{1}{\lambda} \qquad M >> M* \tag{19}$$

This is asymptotic behavior for T when $M >> M*$. It is linear with M at a slope $\frac{e_s}{\mu_s e_c}$.

To find the number M*, we argue that it must be equal to the maximum number of perfectly scheduled jobs that cause no mutual interference. If all service times are assumed to be deterministic, then the maximum number of jobs at the bottleneck node equals the total service required by a job in a cycle/service time spent by a job in the saturated node per cycle.

The total service required by a job per cycle is $\frac{\sum e_i(1/\mu_i)}{e_c}$. The service time spent by a job in the saturated node per cycle is $\frac{e_s(1/\mu_s)}{e_c}$. Thus:

$$M* = \frac{\sum e_i(1/\mu_i)/e_c}{e_s(1/\mu_s)/e_c} \tag{20}$$

$$M* = \frac{\sum e_i(1/\mu_i)}{e_s(1/\mu_s)} \tag{21}$$

For M = 1, total service time required by a job in a cycle = T + $1/\lambda$, from (20)

$$M* = \frac{(T + 1/\lambda)}{e_s(1/\mu_s)/e_c} \tag{22}$$

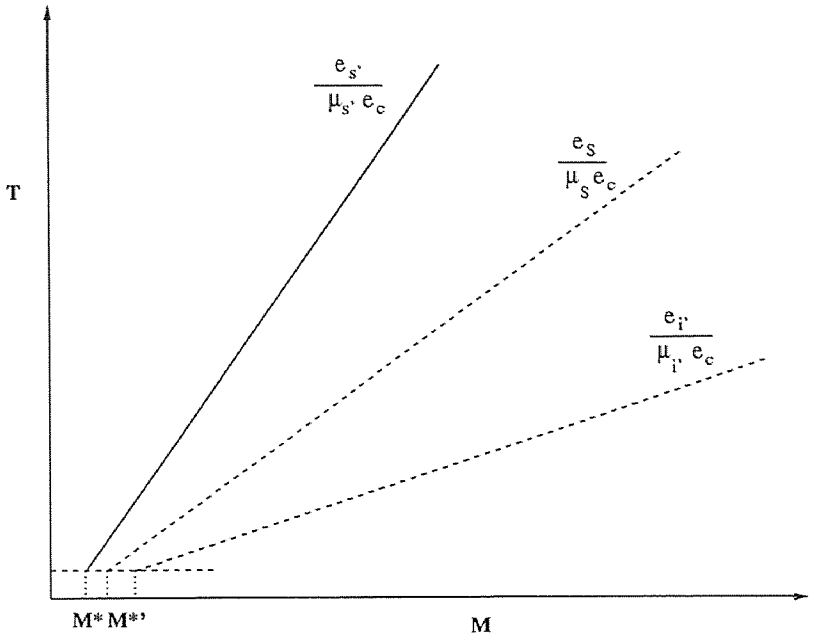

Figure 14: Asymptotic behavior of multiple resource system. Average response time (T) as a function of number of clients (M)

$$T = \frac{M * e_s}{e_c \mu_s} - \frac{1}{\lambda} \tag{23}$$

From (19) and (23) the asymptotic behavior of the multi-client system can be predicted as shown in Figure 14. If we remove the bottleneck by increasing the service rate of the servers or by adding another server to the system, some other node, denoted by s', will become the new bottleneck, and the asymptotic behavior will again be similar to that in (19) and (23) with a new slope $\frac{e_{s'}}{\mu_{s'} e_c}$ and new saturation number M*'. In fact, if we continue this procedure of removing bottlenecks, we will always expose a new one with slope $\frac{e_{i'}}{\mu_{i'} e_c}$ as sketched in Figure 14, where M* must be recalculated with the new rates.

For numerical results, we considered a distributed system with 50 clients and

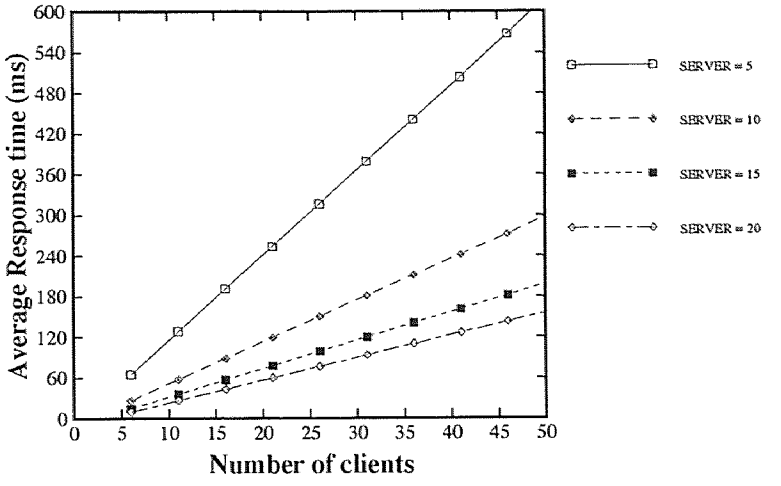

Figure 15: Performance of multiple resource system

20 servers. There were total of 10000 files, distributed equally at each server. Average file size was 2000 bytes and the cache size of 100,000 bytes was considered for each client. The disk transfer rate of 10 Mb/s was assumed for each server.

The performance of the system, for think time of 1 milli-sec, is shown in Figure 15. The response time decreases considerably when the number of servers increases from 5 to 10. However, for the given system parameters, results suggest that it is not worth to invest in more than 15 servers, because any increase in the number of servers from 15 to 20 does not show significant improvement in performance. The behavior shown in the Figures 14 and 15 is very important for distributed and parallel system design since we can predict the number of servers required, for a particular number of clients and for a desired response time. Analysis also predicts the improvement in

system performance if we decide to invest in more powerful resources.

3.7 Conclusion

In this chapter, we proposed a simple distributed system model. The model distributes files (data) among clients and servers. The servers controls the system operations, manipulates and provides data (files) to the processes executing at the clients.

We developed a performance prediction method to estimate various measures in message passing distributed system. The method is based on known product form queueing networks. We used simulations and the analytical method to evaluate the performance of the system model.

The main advantage of the modeling methodology described here is its applicability to very general models. Using this approach, distributed and parallel systems with multiple classes, multi-tasking and job spawning can easily be modeled and their performance can be predicted. We also studied approximate analytical models to study the cache behavior and to identify bottleneck server.

To achieve high performance from any distributed system it is important to efficiently utilize all the resources by distributing data and processes. In the next chapter, we will show how performance can be improved by optimally distributing data among processing sites.

CHAPTER 4

OPTIMAL FILE PLACEMENT STRATEGY

4.1 Introduction

After presenting the basic distributed system model and our performance prediction method, we are now ready to proceed towards our goal to achieve high performance. In this chapter we extend our model such that all the nodes are homogeneous in terms of processing and storage. Each node request and service file requests; behaves as both client and server. Such interconnected sites are also called *peer-to-peer networks.*

To achieve higher system throughput and better response time we design a file placement strategy, such that all the resources are optimally utilized. We then use our prediction method to estimate performance measures and study the system behavior.

The file placement problem in the model is formulated as a routing problem in multiple chain closed queueing networks. The objective is to judiciously place files among nodes such that file requests are optimally routed in the system. Solutions to the file allocation problem in the past have optimized several performance measures like overall response time [50], or storage cost [58]. We optimize *Average node response time*, which is the average over all requests made to a node. To do so, first, the relative throughput at each node is established as a function of file placement. Then we compute the derivative of the average response time with respect to relative

throughput at each node and show that it can be easily obtained from the MVA algorithm [116]. The *steepest descent direction* is obtained by summing the derivatives over all closed chains intersecting at a node. A closed network version of the FD algorithm [95] is then introduced to find the optimum file allocation in a multiple chain queueing network. The solution methodology is similar to the one used by Gerla in [85, 37], however the problem formulation and optimization criteria are related to file placement in message passing distributed systems. The algorithm leads to local minimas, since the convexity of the solution cannot be established in multiple chains [85]. Finally, numerical results are presented to demonstrate the correctness of the algorithm.

For estimating performance measures, we use the method described in the previous chapter. Each node is represented as a single server service center [17]. For validation, the model is simulated on an nCube, a commercial MIMD architecture. In the nCube, each node can be programmed separately and nodes communicate via send and receive calls for message passing. It provides an ideal environment to simulate distributed memory systems.

The remainder of this chapter is organized as follows: In Section 4.2 we describe the extented model of the distributed system. In Section 4.3 we discuss and design the file allocation algorithm. In Section 4.4, we carry out the analysis of the system and estimate different performance measures. Numerical examples are presented in Section 4.5. In Section 4.6 we discuss the simulation model on the nCube and validate

the analytical results against the measurements obtained from the simulation. Section 4.7 concludes the chapter.

4.2 The Extended Model

The target distributed system we consider is a family of processing units, which we interchangeably call "sites" or "nodes", each of which is able to store files, and execute processes. These sites are completely interconnected using a network, which may either be a local area network, or a fast switch or similar device.

The distributed system considered consists of J interconnected processing units, each composed of a processor and storage. Since each site has a storage facility, we do not consider cache memories at the processing sites. Any task or process is generated at some PU, and requires a file for execution. Each file resides normally in some PU's storage; as before, we refer to this node as "host node". Files can be transferred to other PU's on demand. Any file is either *free* or *busy*. It is busy if a copy has been transferred to some other PU; otherwise it is free. A file-copy at a PU is in one of two states: *active* or *non-active*. It is active if it is currently used by a task executing at the node and is non-active if it still resides in that PUs storage without being used.

When a file is requested by some task at a PU, the request is directed to its host node. Then:

- If the file is free, a file-copy is transferred to the requesting node. This action

marks the file as "busy".

- If the file is busy but not active, the file-copy is transferred back to the host node from the node which currently holds it. The host node updates the file status, then sends it to the node requesting it.

- If the file is busy and the file-copy is active, a "file-busy" message is sent to the host node; the corresponding process or task must then wait for the file to become available.

If all the messages, files, user-requests are considered as customers of different classes; the behavior of a file-request can be represented as a closed chain. The four alternatives discussed above are different paths (subchains) which the file request could follow. The probability of following a particular subchain depends on the following conditions:

1. The state of the file (free or busy).

2. The state of the file-copy (active or non-active).

3. The availability of a file at the local node (chance of hit).

Let L be the probability that a file is available on a requesting node, I be the probability that a file is in busy state and Z be the probability that a file-copy is in non active state. These probabilities depends on the file placement. When all the nodes simultaneously submit jobs, file-request from each node follow a similar chain with

four paths. The number of closed chains in the model are exactly equal to the number of nodes in the system. Different chains intersect at nodes. The queueing discipline and the service demands are assumed such that the system satisfies product form requirements. Computations of performance measures pertaining to our model are discussed in Section 4.4.

All processes are executed locally at the node where the user submits them whereas file copies move among the nodes on demand. On job completion, the user enters a "think time" and then submits a new job.

During the execution of processes in the system, new files can be created and old ones could be deleted. Initially, a file is always allocated at the site where it is created and other nodes are informed about its creation. After some fixed time interval t_D, the load balancing algorithm we propose is executed which optimally reallocates the files. Only one site at a particular time is in charge of executing the algorithm, and this site is selected by a token passing algorithm.

To locate a file in the system, each node stores an address table which keeps track of the host node for all the files in the system. A request for a file originating at a node is always directed to the host node. The address table is updated if files are distributed and change location. It also provides location transparency to the user.

4.3 Average Node Response Time Based File Allocation Algorithm

Before we go into the details of the algorithm, we formulate the file allocation problem

as a routing problem in closed queueing network. We first present the problem and results for the single-chain case. Then we extend the results to the multiple chain case.

4.3.1 Single Chain Case

The distributed system, where only one site requests file service can be modeled as a "single chain" closed queueing network. We define the following system parameters for each node j ($j = 0, 1, \ldots, J$).

N : number of customers in closed chain,

y_j : relative throughput,

μ_j : service rate,

$\lambda_j(N)$: throughput,

$L_j(N)$: mean queue length,

$T_j(N)$: mean queueing time (including service time).

In a closed queueing network the relative throughputs satisfy following set of flow conservation equations:

$$y_j = \sum_{i=0}^{J} y_j p_{ij} \tag{24}$$

where p_{ij} is the transition probability from node i to j. The relative throughput does not have a unique solution, however assuming relative throughput at the source node equal to 1, the solution to the Equation (24) becomes unique. The relative throughput y_j can be interpreted as the function of traffic that goes through node j. Let $\lambda(N)$ be

the actual throughput of the chain. Then the throughput at node j is given by:

$$\lambda_j(N) = \frac{y_j}{y_0}\lambda_0(N) = y_j\lambda(N) \tag{25}$$

Under the assumptions, the model satisfies the well known product form solution, the equilibrium probability of the state $(n_0, n_1, \ldots, n_J)$ is given by [97]:

$$P(n_0, n_1, \ldots, n_J) = \frac{1}{G(N)}(\frac{1}{\mu_0})^{n_0} \cdots (\frac{y_j}{\mu_j})^{n_j} \cdots (\frac{y_J}{\mu_J})^{n_J} \tag{26}$$

where n_j is the number of customers at node j, and for $N = n_0 + n_1 + \cdots + n_j$,

$$G(N) = \sum_{n_0+n_1+\cdots+n_j=N} (\frac{1}{\mu_0})^{n_0} \cdots (\frac{y_j}{\mu_j})^{n_j} \cdots (\frac{y_J}{\mu_J})^{n_J} \tag{27}$$

The throughput, mean queue length and mean response time at node j are conveniently expressed using the above constant [17, 89]:

$$\lambda_j = y_j \frac{G(N-1)}{G(N)} \tag{28}$$

$$E[n_{(j)}] = \frac{1}{G(N)} \sum_{i=1}^{N} \rho_j^i G(N-i) \tag{29}$$

where $\rho_j = y_j/\mu_j$

$$E[r_{(j)}] = \frac{E[n_{(j)}]}{\lambda_j} = \frac{\sum_{i=1}^{N} \rho^i G(N-i)}{y_j G(N-1)} \tag{30}$$

From the above equations we note that performance measures for node j are obtained from the normalization constants G(N) and G(N-i), $1 \leq i \leq N$. Alternatively, these quantities can be computed more efficiently by using the MVA algorithm [116]. MVA is based on the following recursive relations of the equilibrium quantities for node j ($j = 0, 1, \ldots J$):

$$T_j(N) = \frac{1}{\mu_j}[1 + L_j(N-1)] \tag{31}$$

$$\lambda_j(N) = \frac{N}{\sum_{i=1}^{N} \frac{y_i}{y_j} T_j(N)} \tag{32}$$

$$L_j = \lambda_j(N) T_j(N) \tag{33}$$

4.3.1.1 File Allocation as a Routing Problem

Our aim is to distribute the files among nodes to minimize the average node response time at each node. The files should be allocated at the nodes such that file requests are routed to nodes to balance the system load and improve performance.

Let T be the total number of files in the system and X_j^f be a binary variable, which is equal to 1 when file f is assigned to node j, 0 otherwise. The following notations refer to each node j:

$F_j = \sum_j^T X_j^f$: number of files allocated to node j,

$P_j = \frac{F_j}{T}$: probability that any arbitrary file is present at local node j.

Assuming all files have the same access probability, the location of a file in the

System dictates the flow of file requests among different subchains. Thus the relative throughput at a node j is a function of the file placement in the system. We approximate the relative throughput y_j in terms of the file placement at node j as follows:

$$y_j = \begin{cases} P_j & \text{if j is a requesting node} \\ (1 - P_j)\frac{F_j}{\sum_{j=0}^{J} F_j} & \text{if j is not a requesting node} \end{cases}$$

Now, the file placement problem in a closed queueing network can be defined as follows:

Given : Service rates $\{\mu_j\}$, chain population {N}, total number of files {T}, number of nodes {J}.

Minimize: the mean response time E[r_j],

With respect to: relative throughput y_j.

Subject to following constraints:

- $y_j \geq 0 \ \forall j$,
- $\sum_{j \in IN(s)} y_j = \sum_{j \in OUT(s)} y_j \forall s = 0, \cdots, J$,

where

J = number of nodes in the network,

IN(s) = set of subchains incoming into node s,

OUT(s) = set of subchains outgoing from node s.

4.3.1.2 Objective Function

To compute the objective function we take the derivative of the mean response time E[r_j] at node j with respect to the relative throughput y_j, from Equation (30):

$$\frac{\partial(E[r_j])}{\partial y_j} = \frac{\partial \sum_{i=1}^{N} \rho^i G(N-i)}{y_j G(N-1)} \tag{34}$$

$$= \frac{y_j G(N-1) \frac{\partial(\sum_{i=0}^{N} \rho^i G(N-i))}{\partial y_j} - \sum_{i=0}^{N} \rho^i G(N-i) \frac{\partial(y_j G(N-1))}{\partial y_j}}{y_j^2 G^2(N-1)} \tag{35}$$

To compute the derivative of the terms of the numerator we argue as follows:

The gradient of the normalization function G(N) with respect to y_j is given as [85]:

$$\frac{\partial G(N)}{\partial y_j} = \frac{G(N)L(N)}{y_j} \tag{36}$$

Using the result of Equation (36) and under the "independence assumption", that is, that the service time at each node is assumed to be independent of the interarrival time [95], it can be proved that:

$$\frac{\partial(\sum_{i=1}^{N} \rho^i G(N-i))}{\partial y_j} = \frac{G(N) \sum_{i=0}^{N} \rho^i L_j(N-i)}{y_j} \tag{37}$$

The proof is included in the appendix for completeness.

Substituting (37) in (35) and from Equations (28) and (29):

$$\frac{\partial(E[r_j])}{\partial y_j} = \frac{1}{y_j \lambda_j} \left(\sum_{i=0}^{N} \rho^i L_j(N-i) + L_j(N) \left[L_j(N-1) + 1 \right] \right) \tag{38}$$

This derivative is very convenient to compute since every quantity on the right hand side can be evaluated from the MVA algorithm.

4.3.2 Extension to Multiple Chain

The distributed system where every node may request for a file is modeled as multiple chain closed queueing network. The number of chains in the model is equal to the number of nodes. For multiple chains we define:

K	: number of closed chains or number of nodes,
N_k	: population of chain k, $k = 1, \ldots, K$,
$N = \sum_{k=1}^{K} N_k$	: total population size,
$y_{(k,j)}$	: chain k relative throughput at node j,
μ_j	: service rate at node j,
$\lambda_{(k,j)}(N)$	: chain k throughput at node j,
$L_{(k,j)}(N)$	: chain k mean queue length at node j,
$T_{(k,j)}(N)$	: mean queueing time (including service time),

where N is the population size vector such that N = $(N_1, N_2, \ldots, N_k)$. Using the

normalization constant G(N), the chain throughput of chain k is given by [17]:

$$\lambda_k(N) = \frac{G(N - e_k)}{G(N)}, N \geq e_k, k = 1, \ldots, K, \tag{39}$$

where e_k is a vector with the kth element equal to one and all others equal to zero. The chain k throughput, mean queue length and mean response time at node j can be computed as follows [17]:

$$\lambda_{(k,j)}(N) = y_{(k,j)}\lambda_k(N) \tag{40}$$

$$E[n_{(k,j)}] = \frac{1}{G(N)} \sum_{i=1}^{N_k} \rho^i G(N - ie_k) \tag{41}$$

$$E[r_{(k,j)}] = \frac{E[n_{(k,j)}]}{\lambda_{(k,j)}(N)} = \frac{\sum_{i=1}^{N_k} \rho^i G(N - ie_k)}{y_{(k,j)} G(N - e_k)} \tag{42}$$

Similar to the single-chain, the derivative of the normalization function G(N) is given by:

$$\frac{\partial(G(N))}{\partial y_{(k,j)}} = \frac{G(N) L_{(k,j)}(N)}{y_{(k,j)}} \tag{43}$$

Using this we take the derivative of Equation (42) as follows:

$$\frac{\partial(E[r_{(k,j)}])}{\partial y_{(k,j)}} = \frac{y_{(k,j)} G(N - e_k) \frac{\partial(\sum_{i=1}^{N_k} \rho^i G(N - ie_k))}{\partial y_{(k,j)}} - \sum_{i=1}^{N} \rho^i G(N - ie_k) \frac{\partial(y_{(k,j)} G(N - e_k))}{\partial y_{(k,j)}}}{y_{(k,j)}^2 G_2(N - e_k)} \tag{44}$$

Using Equations (39), (41), (42), (43) , (37),we obtain ;

$$\frac{\partial(E[n_{(k,j)}])}{\partial y_{(k,j)}} = \frac{1}{y_{(k,j)}\lambda_{(k,j)}} \left[\sum_{i=0}^{N_k} \rho^i L_{(k,j)}(N - ie_k) + L_{(k,j)}(N)[L_{(k,j)}(N - e_k) + 1] \right] \tag{45}$$

The last equation has the same properties as the single chain Equation (38).

Next, we assign a weight $w_{(k,j)}$ at node j due to chain k as follows:

$$w_{(k,j)} = \frac{\partial(E[n_{(k,j)}])}{\partial y_{(k,j)}} \tag{46}$$

The total weight due to all chains at node j is given by:

$$w_j = \sum_{k=1}^{K} w_{(k,j)} \tag{47}$$

These weights are easily determined by the MVA computation.

The relative throughput of chain k at node j, $y_{(k,j)}$, is computed from the file placement as follows:

$$y_{(k,j)} = \begin{cases} P_j & \text{j = k} \\ (1 - P_j)\frac{F_j}{\sum_{j=1}^{J} F_j} & j \neq k \end{cases} \tag{48}$$

4.3.2.1 Algorithm

Now we propose the file allocation algorithm for closed queueing networks which is a combination of the FD algorithm for open networks and the MVA method developed

for closed networks. We assume that we have a initial file assignment $F^{(0)}$. This $F^{(0)}$ can be easily obtained by choosing an arbitrary node and assigning all the files to it. We use here a description similar to the one used for the FD algorithm in [95].

Step 1 : Let n = 0, let $F^{(0)}$ be the initial file assignment,

Step 2 : Compute weights, $w_j = \sum_k w_{(k,j)}$, for each node, using the MVA method,

Step 3 : Identify the nodes with maximum and minimum weights, j_{max} and j_{min} respectively. Let F^* be the vector obtained by assigning one file from j_{max} to j_{min},

Step 4 : Compute the $b^{(n)}$ and b^* for vector $F^{(n)}$ and F^* respectively as follows;

$$b^{(n)} = \sum_j w_j F_j^{(n)}$$

$$b^* = \sum_j w_j F_j^*$$

Step 5 : If $|b^{(n)} - b^*| < \epsilon$, where $\epsilon > 0$ is properly chosen tolerance, then STOP, else go to Step 6.

Step 6 : Assign u number of files from the node with maximum weight (j_{max}) to the node with minimum weight (j_{min}) . The objective function (45) is evaluated (using MVA). If the objective increases, half number of files ($u/2$) is tried and so on, until the objective decreases.

Step 7 : Let $n = n + 1$ and go to Step 2.

4.3.3 Algorithm Convergence and Complexity

The convergence of the algorithm is guaranteed by the fact that the iterations always reduce the objective function. It is evident from Equations (48) and (33) that during each iteration when some x number of files are moved from node j to some other node, the relative throughput and mean queue length at node j decreases. As queue length decreases so does the objective function and mean response time at node j (see Equations (31) and (38)). In Section 4.5 we experimentally establish this fact.

The choice of u (the number of files to be assigned from the node with maximum weight to the node with minimum weight) in Step 6 is a major factor. If u is small, the algorithm will be slow as it will distribute the files in small groups. However, if u is large, algorithm may lead to oscillations around a local minima or around several minimas.

For the single chain case, each iteration of the algorithm performs O(JN) computations for computing weights in Step 2 and O(JN) for computing the the MVA values. For multiple chains, the MVA calculations in each iteration perform O(JNK) computations, where the total number of customers N $= n_0 + n_1 \cdots + n_k$.

Since, the number of chains is equal to the number of nodes, *i.e.*, K = J . Hence the algorithm in each iteration performs maximum of $O(J^2N)$ computations. The most time consuming Step of the algorithm is step 6, where the MVA calculations are repeated several times to find the optimal number of files to be reallocated.

4.4 Performance Analysis

To estimate performance measures and predict system behavior we take a similar approach as described in Chapter 3. Let S denote the state of the multiple chain queueing network:

$$S = (n_0, n_1, \ldots, n_j) \tag{49}$$

where $n_j = (n_{j1}, n_{j2} \ldots, n_{jR})$, n_{jr} is the number of class r customers at node j. Let $1/\mu_{jr}$ be the average time to service class r customer at node j and $1/\mu_j$ be the mean service time of node j. If all the nodes are assumed to be of type 1 service centers, the equilibrium state probability of such closed queueing network with multiple classes has the following product form [17]:

$$P(S) = \frac{1}{G(N)} \prod_{j=1}^{K} g_j(n_j) \tag{50}$$

where $N = n_0 + n_1 + \cdots + n_j$,

$$g_j(n_j) = \frac{n_j! \prod_{r=1}^{R} [(\Lambda e_{jr})^{n_{jr}} / n_{jr}!]}{\prod_{j=1}^{n_j} \mu_j} \tag{51}$$

and $1/\Lambda$ is the think time. All service time distributions are assumed to be exponential. The quantity e_{jr} is proportional to the total arrival rate of class r at node j and is interpreted as the relative arrival rate of class r customers to the service center j. The existence of the steady state distribution, for a closed network, depends on the

solution of the following set of flow equations:

$$e_{jr} = \sum_{i,s} e_{is} P_{is,jr} \tag{52}$$

G(N) is a normalizing constant which must be calculated over all possible states using multi-variate convolutions [111, 112, 113]. If there are k chains and N customers in the closed network, the normalization constant is given by:

$$G_N(Y_1, Y_2, \ldots, Y_k) = \sum_{n_0+\cdots+n_N=N} [\prod_{j=1}^{k} \gamma_j(n_j)] \tag{53}$$

where Y_k denotes the variable for chain k and

$$\gamma_j(n_j) = \sum_{n_0+\cdots+n_N=N} [\prod_{j=1}^{N} g_j(n_j)^{n_j}] \tag{54}$$

The flow equations (52) can be easily obtained from the model description in Section 4.2. The quantity $g_j(n_j)$ can be computed from flow equations, which yields the the normalization constant. The mean service time of node j: $1/\mu_j = \sum_r 1/\mu_{jr}$.

The performance measures for any job class at any node can be obtained from the calculations for normalization constant. The average throughput θ_{jr} of class r job in chain Y_k through node j in a closed network is given by [17]:

$$\theta_{jr} = e_{jr} \frac{G_N(Y_1, Y_2, \ldots, Y_k - v_r)}{G_N(Y_1, Y_2, \ldots, Y_k)} \tag{55}$$

where v_r is the vector where rth element is equal to one and all others equal zero. The average utilization due to class r job at node j is given by [17]:

$$U_{jr} = \frac{\theta_{jr}}{\mu_{jr}} \tag{56}$$

the average node utilization and the average node throughput is given by [17]:

$$U_j = \sum_r U_{jr}; \theta_j = \sum_r \theta_{jr} \tag{57}$$

For J nodes the steady state probabilities (L, I, Z) can be approximated as follows:

1. the probability that a file is available locally : L = F_j/T
2. the probability that a file is in busy state : I = 1/J
3. the probability that a file copy is in non-active state is assumed to be 0.9 *i.e.* Z = 0.9

4.5 Numerical Examples

In this section we show numerical results for the file allocation algorithm and performance measures estimated from the prediction method. For both examples, we consider five node distributed system. Each node is assumed to be identical and to have same service times. All time measurements are carried out in "ticks" where one tick is 128 micro-secs.

4.5.1 Example 1

To show that the file allocation algorithm works for any arbitrary distribution of files, we start from three different initial file allocations, as shown in Tables 3, 4 and 5. A total of 1500 files are considered for distribution. The chain population is assumed to be 5, *i.e.*, $N_k = 5$, and the file transfer times are exponentially distributed with a mean of 300 ticks. Each message transfer from one node to another takes 2 ticks. Think time is assumed to be close to zero, therefore, all the time there are 5 customers in a chain. The number of files (u) to be reallocated for each iteration at step 6 was chosen to be 10.

The same final file assignment pattern was reached from all three initial allocations. The numerical values of average node response time and average throughput clearly show that as a consequence of file allocation all the nodes are balanced and and the performance improved.

4.5.2 Example 2

For estimating performance measures the steady state probabilities (L, I and Z) have to be approximated as discussed in section 4.4. For five nodes, the probability that a file is available on a local node is 0.2, which means that most of the file-requests travels through the network. The number of files, message and file transfer times are the same as in the previous example. Each user creates one process, and will create a new one as soon as it is informed that the previous one it created has finished execution.

Table 3: Initial file-allocation pattern # 1 of Example 1

j	w_j	# of files	Av. Node Response Time	Av. Throughput
1	2.536618	0	136.000000	0.01858
2	2.536618	0	136.000000	0.01858
3	2.536618	0	136.000000	0.01858
4	2.536618	0	136.000000	0.01858
5	5.605441	1500	301.389404	0.09292

Table 4: Initial file-allocation pattern # 2 of Example 1

j	w_j	# of files	Av. Node Response Time	Av. Throughput
1	1.596591	500	186.323120	0.03867
2	1.055574	0	136.000000	0.02841
3	1.596591	500	186.323120	0.03867
4	1.055574	0	136.000000	0.02841
5	1.137262	500	186.323242	0.03867

Table 5: Initial file-allocation pattern # 3 of Example 1

j	w_j	# of files	Av. Node Response Time	Av. Throughput
1	1.415123	750	221.448975	0.05657
2	0.958248	250	158.703003	0.03185
3	0.928247	200	153.954102	0.03101
4	0.900157	150	148.953003	0.03024
5	0.839783	150	148.952881	0.03024

Table 6: Optimal file-allocation pattern of Example 1

j	w_j	# of files	Av. Node Response Time	Av. Throughput
1	1.247308	300	163.025635	0.03687
2	1.247274	300	163.001221	0.03687
3	1.247274	300	163.001221	0.03687
4	1.247274	300	163.000732	0.03687
5	1.248510	300	163.977783	0.03687

The time to execute a process is considered to be negligible and is executed as soon as the requested file is available. Furthermore, we assume that each node creates and deletes 20 files every 2,500,000 ticks such that the number of files remains the same for the steady state analysis. Later, from our simulation results we show that the estimated measures are robust to the rate at which files are created and deleted. The load balancing algorithm is executed every 3,500,000 ticks to reallocate the files. The performance measures are obtained by decreasing the user think time.

As the nodes are loaded more heavily by the user's file requests, the average utilization, response time and throughput of a node increases and then remains constant, as shown in Figures 16, 17 and 18. This is because the maximum number of active jobs among chains is fixed. The effectiveness of the file allocation (load balancing) algorithm is apparent from Figure 17, where we show the average node response time of all the nodes is the same all the time. The results also show that increasing the number of nodes does not have a considerable effect on an individual node's behavior. This is due to the fact that each node is both a request generating site

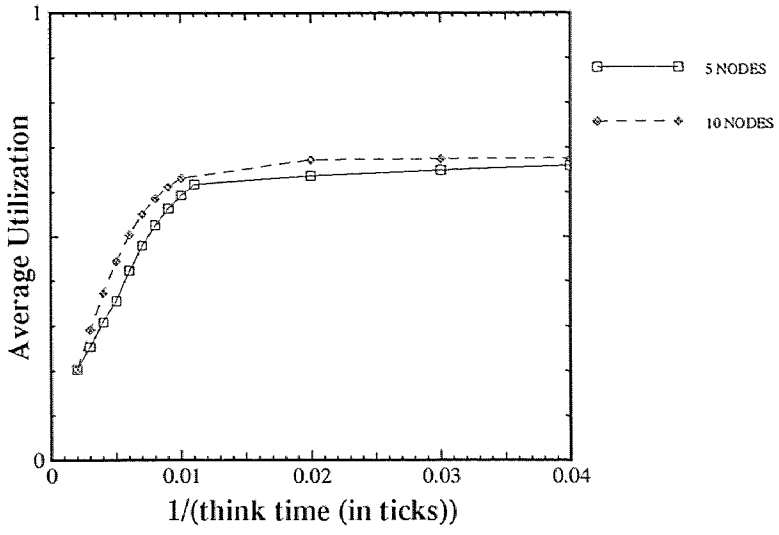

Figure 16: Average node utilization as a function of think time.

and request processing site. However, increasing the number of sites in a distributed system definitely increases the total system throughput.

Figure 16 shows that when the think time at each node is very small, close to 3ticks, the utilization of nodes is approximately 65%, which means that file-requests from all the nodes takes a maximum of 65% of the CPU time. The remaining 35% of the CPU time could be used for local processing such as running window programs and other local processes. With a think time close to 64 ticks, the average case, the node utilization is about 20%. This suggests that an average 80% of the CPU time isavailable for computing and 20% is used for resource sharing.

The performance measures for each customer class are obtained using Equations (55) and (56). Figure 19 shows the utilization of a node due to different customer

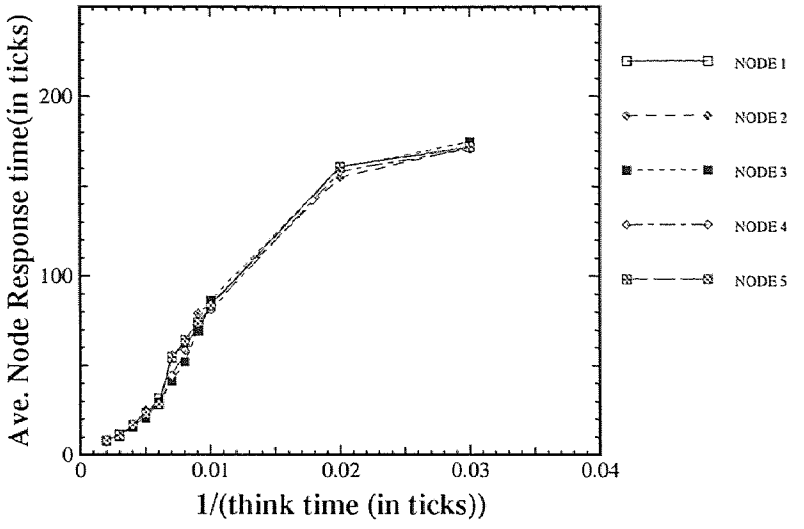

Figure 17: Average node response time as a function of think time for all the five nodes. As a consequence of load balancing algorithm, all nodes exhibit same behavior.

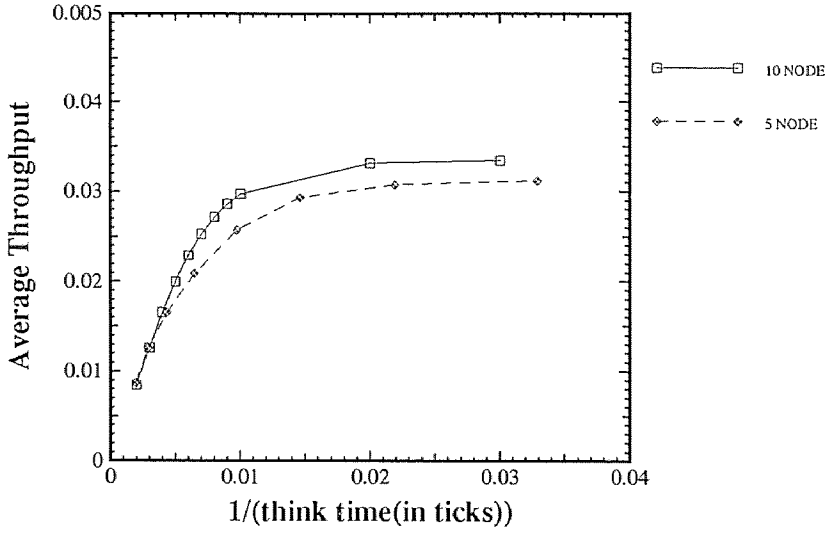

Figure 18: Average throughput of a node as a function of think time.

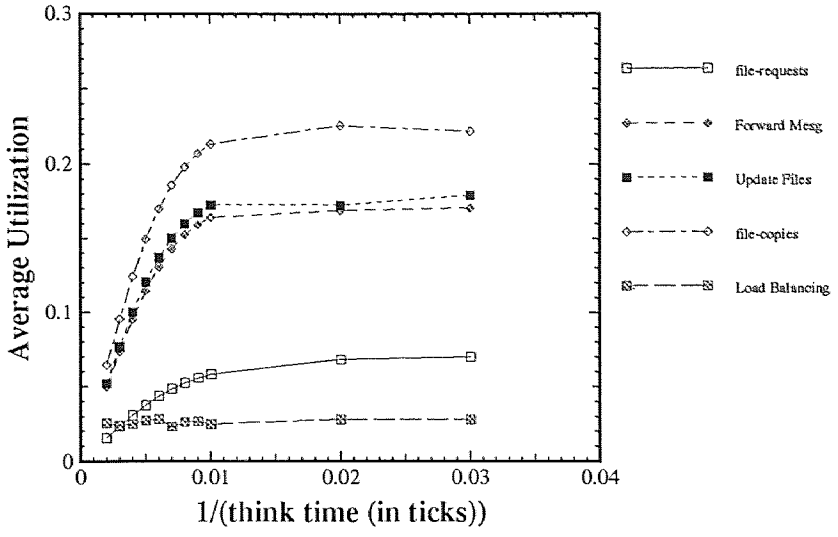

Figure 19: Average utilization of a node due to different job classes. Maximum node utilization is due to file-copy transfers, followed by other customer classes. Utilization due to load balancing overhead is considerably low.

classes. It is important to know which customer class utilizes the node most, so that measures like installing a co-processor or increasing the processor speed can be taken.

File creation, deletion and load balancing also cause message exchanges and file transfers. These messages and files are also modeled as customers which follow closed chains. Figure 19 shows the node utilization due to these file transfers and message exchanges. For the given parameters, the overhead due to load balancing is less than 4%; however, overhead depends on the rate at which files are created and deleted as well as the number of times the load balancing algorithm is executed.

4.6 Measurements and Validation

The accuracy of the prediction method described in the last section is established by comparing the estimated analytical values with the measurements obtained from simulations on the nCube.

Each nCube processing node represents a computing site of the distributed system. Following initial file allocation pattern was considered:

Node	*# of files*
1	100
2	100
3	600
4	400
5	300

A small number of files was chosen to introduce more conflict among different file requests. On average 20 files were created at each node every 2,500,000 ticks. Same average number of files were removed from the system every 3,000,000 ticks. All times and parameters are the same as in the queueing model (Example 2). The system was made to run for a extended period of time and measurements for node utilization, throughput and response time for different think times were made. The queueing network model differs from the simulated model in the following respects:

1. For simulation purposes, the service time is uniformly distributed between 200 and 400 ticks. In the queueing network model, the service time distributions to service a file are assumed to be exponential.

2. In the queueing network, we assumed that the number of files in the system remains constant, however in our simulation the number of files actually vary and the files are dynamically created and deleted. Sender-initiative strategies [33] are used for both creation and deletion of files.

3. The queueing discipline at a node is considered to be FCFS for file requests. In simulations, some modifications have to be made to make the system work correctly. Whenever a requested file is busy at a node, the node forwards the request as a forward message to some other node. Any other requests for the same file must wait at the host node until the file copy is non-active. In the meantime, the node services all the other file requests on an FCFS basis. Hence, the node deviates from the FCFS discipline for file requests requesting the same file.

4. To obtain the numerical values for marginal probabilities in the analytical model, steady state probabilities (L, I and Z) have to be approximated as discussed in Section 4.4. However, our simulations do not use steady state probabilities to direct the flow of customers. Initially all the files are free at their respective host nodes. As the simulation proceeds, nodes request files and requests are serviced depending on whether the file is busy or free.

The measurements taken for node utilization and node response time are considered for validating the analytical results. Analytically the utilizations are calculated using Equations (56) and (57) and mean node response time is obtained from

Table 7: Node utilization compared with measurements from simulation

1/(think time)	Average Utilization of node j due to two job classes			
	file-request		files-Copies	
	Analytical	Simulation	Analytical	Simulation
0.00200	0.015786	0.016089	0.064504	0.063456
0.00300	0.023531	0.023982	0.095651	0.093645
0.00400	0.030894	0.031245	0.124427	0.120023
0.00500	0.037590	0.039873	0.149498	0.145671
0.00600	0.043415	0.045734	0.170121	0.169351
0.00700	0.048299	0.049879	0.186277	0.184567
0.00800	0.052291	0.054389	0.198461	0.197819
0.00900	0.055505	0.056015	0.207392	0.206999
0.01000	0.058075	0.059023	0.213796	0.213123
0.02000	0.067824	0.067682	0.225959	0.224989
0.03000	0.069867	0.069812	0.222062	0.222011

(42). Both analytical results and measurements obtained from the nCube are compared in Tables 7 and 8.

4.7 Conclusion

In this chapter, we presented a file allocation algorithm for distributed system architectures composed of an arbitrary number of processing sites, interconnected by a high-speed network. We use multiple chain closed queueing networks to design and evaluate the algorithm. We also analyze the distributed system and estimate various performance measures.

The file placement algorithm attempts to balance the load among different nodes by minimizing mean response time of each node. It is invoked periodically

Table 8: Comparison of average node response time and utilization

1/(think time)	Average Response time		Average Utilization	
	Analytical	Simulation	Analytical	Simulation
0.00200	8.053	8.4066	0.203878	0.200897
0.00300	11.521	10.421	0.253603	0.267610
0.00400	15.603	15.918	0.308903	0.325671
0.00500	22.630	20.508	0.356481	0.387689
0.00600	31.456	29.292	0.423960	0.446718
0.00700	53.988	41.195	0.480543	0.496712
0.00800	62.717	52.148	0.526786	0.521348
0.00900	69.283	69.982	0.564015	0.571231
0.01000	83.789	86.689	0.593794	0.599993
0.02000	161.43	160.74	0.636730	0.622496
0.03000	172.80	175.15	0.650403	0.623611

and optimally allocates the files. A least busy node is chosen using the MVA algorithm, taking into account service costs, throughput and mean queue lengths at nodes. Files are moved from the most busy node to the chosen node using a downhill, flow deviation technique.

The algorithm can also be activated when one or more nodes has to be repaired or shut down for maintenance. All the files of the node can be optimally distributed among other nodes. The algorithm's efficacy is illustrated via numerical examples. Even though the algorithm presented here is for file allocation, the same technique can be easily applied to distribute other static objects in a distributed system.

The performance analysis presented here is also very beneficial. The use of multiple classes suggests that we can analyze behavior of a particular information unit on different nodes. This provides vital information about the nature and type of

resources required for the system. The performance prediction methodology described here shows the impact of users' load on the node response time and node utilization. These results provide a set of guidelines to compare different system models and their resources allocation policies.

The complete system was simulated on an nCube multiprocessor. The measurements not only validates the analytical results but also shows that the performance of the distributed system can be predicted without worrying about the network design. Furthermore, they suggest that in a distributed system where every node generates and serves requests, increasing the number of nodes does not considerably affect node performance. However, it does increase the overall system throughput.

In short, this chapter describes an efficient technique to distribute data and control among sites to improve the performance. To further enhance the system performance and utilize available resources it is important to distribute processing. In the next chapter, we design dynamic policies to move processes and data in the system.

CHAPTER 5

ON-LINE ADAPTIVE ALGORITHM FOR PROCESS MIGRATION

5.1 Introduction

In this chapter, we design and evaluate *Adaptive Load Balancing* algorithms for both processes or tasks, and data or files. We introduce a the gradient descent paradigm to compute on-line load balancing decisions. The purpose is to substantially improve system performance. We have implemented and tested this paradigm on the nCube target architecture, and compared it with the case where no load balancing is carried out, as well as with "random" load balancing. We demonstrate via extensive measurements that simple adaptive load balancing algorithms can substantially improve distributed system performance, over that of systems with no load balancing. The order of magnitude improvement obtained is a reduction of 50% in the average process response time. We also show that simple algorithms can achieve better performance than more sophisticated algorithms, and that they result in less overhead. Numerous measurement results on an nCube are presented, using performance metrics which have already been discussed.

Full load balancing, *i.e.* one which dynamically and optimally reallocates all tasks and files as new jobs or tasks arrive, is NP-hard. However *incremental load balancing*, which makes decisions concerning newly created tasks, or which makes adjustments to task or file assignments as a function of current information, can

be carried out efficiently, with algorithms of low complexity. This is what we will demonstrate and test here. We design simple algorithms for dynamic load-balancing of both tasks and files. The load balancing policies we propose are based on gradient descent of a function which expresses the cost of task execution at different sites.

We compare these algorithms with cases where no load balancing is carried out, and test them in an nCube distributed processing environment.

The results indicate that simple load balancing policies can be very beneficial to system and user performance.

5.2 Overview of Adaptive Algorithms

There is an abundant and substantial literature on adaptive load sharing and balancing policies for distributed systems, some of which is given in the bibliography. Much of this work uses simulation to evaluate dynamic load balancing policies.

Several adaptive load sharing strategies for process migration and data placement in distributed systems [78, 77, 79, 81, 76] have been proposed. Comparative and comprehensive studies include [83], and [77], which point to the potential benefits of adaptive load sharing and compare different policies, concluding that very simple adaptive load sharing policies which collect a small amount of system state information and use this information in simple ways, can yield dramatic performance improvement.

Load balancing has also been investigated in other distributed resource envi-

ronments. In computer networks, routing can be examined as an instance of load balancing [117, 118, 119, 75]. The goal is to find the optimum paths for packet flow so that some performance measure (delay or response time) is optimized. In [120] a central controlling site or network management center, is used to monitor loads and traffic patterns, to periodically compute optimal load distribution, and provide load balancing information to nodes. The load monitoring and load distribution computations can also be carried out in a distributed fashion [121]. These policies are static in nature and are exploited for dynamic load balancing. In distributed programming environments, where a program can be represented as parallel tasks or modules, the optimum module to processor assignment problem is a version of dynamic load balancing [122, 123, 124]. In distributed databases, transaction response time is the measure of interest; it is optimized by efficient data allocation and appropriate transaction routing [125].

More recently in [38], a general method for quantitative and qualitative analysis of adaptive load sharing algorithms in distributed systems is discussed, suggesting that remote execution should be relatively restricted, and that more than 90% of the decisions made by the adaptive algorithm should be correct. Most of the adaptive algorithms [77, 38] considered deal only with task or process movement; they assume that jobs and data are one information unit and can migrate to any site.

Our approach will cover both process and file movement, and distribution among processing units. The load balancing policies are adaptive and they utilize

current system state to make decisions. The control is distributed; and each node makes independent decision to balance the load.

5.3 The Distributed System Environment

Our message passing, distributed memory, distributed system allows both processes and data (files) to move among various sites. When a user submits a task or job at a site, it requests for a file. Files are placed at their "host nodes". We will call the processing units (PU) where a task or job is created the "local node"; all other PUs are "foreign nodes" for the requested file and the task. When a task is created the following Four Alternatives can occur:

1. The local node is the same as the host node for the requested file; the task is simply executed at the local node. The only extra cost (other than job execution) would be in case the requested file is busy, and the file-copy has to be transferred from a foreign node to the host node. This overhead cost will be denoted *file-busy-overhead.*

2. In this and the other two cases below, the local and host nodes for the file requested are not the same. The local node requests a file-copy from the host node and executes the job locally. Overhead in this case includes the file-copy transfer cost from the host to the local node plus the file-busy-overhead.

3. The job migrates to the host node for execution. Overhead includes process migration cost and file-busy-overhead.

4. The job migrates to some foreign node and the foreign node requests the file-copy from the host node to compute the job. Extra expense includes process migration cost, file-copy transfer cost and file-busy-overhead.

Case 1 requires the least overhead, while Case 4 is the most costly in terms of overhead. The choice among these alternatives must be made so that the job completion time - including overhead - is minimum and load is equally distributed among all PUs.

5.3.1 Assumptions concerning system operation

We assume the *single request acceptance policy* [93], with no rejection allowed *i.e.* migrated tasks arriving at a node cannot be refused and will be executed there.

During the execution of processes in the system, new files can be created and old ones could be deleted. Initially, a file is always created at the local node and other nodes are informed about its creation. After a fixed time interval t_D, if files are not distributed equitably, the file allocation algorithm, discussed in chapter 4, is executed which redistributes files so that each node has roughly the same occupied space. Files can also be deleted to make sure that the file-system is not full. File deletion is carried out only when 1) a delete process is invoked at a host node whose file system is full, and 2) the file is free. Both file deletion and redistribution use the send-acknowledge policy [33].

The service discipline at all PUs is FCFS for all messages, files, new job re-

quests etc. except for file requests and migrated jobs. Any job generated at some PU, can migrate to any site and request a file for execution. File requests for the same file are served in Time-Stamp order by the host node. When a file is busy at a node, the node forwards the file request to some foreign node which holds the file-copy. Any subsequent requests for the same file must wait at the host node until the file copy reaches the requesting node. In the meantime, the node services the other requests in FCFS order.

File placement and job execution are transparent to the users. A user submits a task at the node where it is logged (local node), but the task receives service at some arbitrary node determined by the load balancing algorithm. Each node periodically updates and broadcasts its current load information. A load vector at each node maintains load status information of all the nodes.

In the experiments described here, we use the nCube, to experiment the algorithms and obtain measurements for different strategies, where each PU of the nCube represents a site of the distributed system. We will compare our proposed dynamic gradient descent based policies with the case when no load balancing is carried out, and with random load balancing, using the efficiency and performance measures discussed in [38].

5.4 Simple and effective load balancing policies

Before we enter into the specific algorithms we will study in the sequel, let us consider

four simple strategies to migrate processes and transfer files-copies.

For the ease of comparison, in all four strategies we initially allocate an equal number of files at each node; as files are created and deleted, the file allocation algorithm reallocates files in such a manner that an equal number of files are kept at all nodes. Only one node at a particular time is in-charge of executing the algorithm, and this node is selected by a token passing algorithm. When the quantum for file redistribution t_R is reached at a node, it holds the token and redistributes the files. When redistribution is complete or if the node has not reached its time quantum, it passes the token over to the next node.

The four process allocation policies we compare are:

- NO-PROCESS-MIGRATION (NP): In this strategy only file-copies are allowed to move and all processes are executed at the local node. This strategy does not use the adaptive algorithm, and every file request is forwarded to the host node. It incurs no overhead since load status information is not collected, nor exchanged between nodes.

- ADAPTIVE-ALL-NODES (AD-ALL): This strategy uses adaptive load sharing to choose between executing the job at the local node or migrating it to *any* other node. A process can execute at the local, host or foreign node. Load information is maintained and exchanged, resulting in overhead.

- ADAPTIVE-TWO-NODES (AD-TWO): Instead of choosing among all nodes, the algorithm only considers the local and the host node for process execution.

This strategy is interesting because if a process is allocated to some foreign node, the host node still has to send (and receive) the file-copy to (and from) the foreign node, resulting in additional delay. Moreover in AD-TWO the adaptive algorithm is limited to a choice between two rather than between N sites, resulting in a much lower computational overhead.

- RANDOM-TWO (RD-TWO): Here a random selection, with equal probability, is made for process execution between the host node and the local node. Just as with the NP policy, here we do not have system overhead related to the collection of a load information and its transmission to (or reception from) other sites.

To illustrate the effect of the file allocation algorithm we have described above, we also test a NO-FILE-PLACEMENT (NO-FI) strategy, which operates in conjunction with the best of the above four policies, but *without* file redistribution.

5.5 Adaptive Algorithm Design

Adaptive algorithms for load sharing, comprise two main activities - *information dissemination* and *decision making (control)* [38]. In order to maintain a complete and consistent view of the entire system, load information about all nodes is maintained and periodically updated at each site.

Load indices considered here are based on total load information and will use the number of processes assigned to the node and their average service time. In

addition, the load index will also include all other information units such as messages and file-copies. Let L denote this instantaneous load index for some arbitrary node.

Periodically each t_B time units, the node broadcasts its load to all other nodes, and each node maintains a load vector which is updated whenever information arrives from other nodes. It updates the load of other sites when it receives them, which depends on t_B, and refreshes its own when a new customer starts service. Overhead due to broadcast can be controlled by varying the t_B, and a balance should struck between overhead and performance improvement.

The algorithms we consider make all decisions locally, and are activated upon the creation of a tasks. The transfer policy (which job should be migrated) will depend not only on the load vector but also on the task's characteristics and other system parameters. These characteristics include process execution time and requested file size; relevant system parameters are the process migration and file transfer delays. We summarize below the main properties of the algorithms we consider:

Algorithm Characteristics

Decision making invocation	event driven (process creation, file redistribution)
Transfer policy	global load information, process characteristics, system parameters
Location policy	any node which provides minimum cost
Acceptance policy	single request - no rejection allowed
Information policy	periodic load dissemination

5.5.1 Algorithm design

Consider a fully connected J node system, where each site maintains a load vector as well as a file-table which indicates the host node for each file.

For any process created at a node, let us introduce the following notation:

- t_E is the "pure" execution or run time of the process.
- t_{FT} is the time it takes to transfer a particular file from the host to the local node, including the file-busy overhead. This quantity depends on the size of the file.
- t_M is the time it takes to move a process to a remote site.
- L_l is the current total load at the local node, expressed as the estimated execution time of all waiting processes, messages, etc..
- L_r is the total load at the remote node, expressed as the estimated execution time of all waiting processes, messages, etc..
- ζ is the probability of deciding that the process will be executed at the local node, where it was created.

Notice that ζ is the decision variable; it will be computed by the algorithms we design. In practice, if $0.5 \geq \zeta$, the decision will be *not* to move the process. However in certain cases a decision threshold larger than 0.5 may be used.

The algorithms we consider will compute ζ using an update rule of the form:

$$\zeta_{k+1} \Leftarrow \zeta_k - \eta \frac{\partial H}{\partial \zeta}|_h \tag{58}$$

where h is the index of the update step we are considering, and H is an appropriate cost function obtained from the quantities defined above, and evaluated at each update step. Note that this is a gradient descent rule which is guaranteed to reduce the cost at each step. η is the speed of gradient descent, and the algorithm must be stopped whenever two successive values of the cost function are less than some "level of diminishing returns" ϵ.

Notice that we will *not* invoke the algorithm if the local node is same as host node (Alternative 1 of section 5.3). In the other cases (Alternatives 2,3 & 4 of Section 5.3), the cost H of executing the job on every node is computed using a cost function and the decision is taken using the above iterative procedure for computing ζ_h.

We will now indicate how a meaningful cost H is chosen. The cost of executing the process locally W_l, or of executing it remotely W_r, are computed at the local node from the load information and job characteristics:

$$W_l = t_{FT} + t_E + L_l \tag{59}$$

$$W_r = \begin{cases} t_M + t_E + L_r & \text{if the remote site is a host node} \\ t_M + t_E + L_r + 2 * t_{FT} & \text{if the remote site is a foreign node} \end{cases} \tag{60}$$

The net cost H is then the average value of the total execution time of the process, under the policy ζ:

$$H = \zeta W_l + (1 - \zeta) W_r \tag{61}$$

Clearly:

$$\frac{\partial H}{\partial \zeta} = W_l - W_r + \zeta(\frac{\partial W_l}{\partial \zeta}) + (1 - \zeta)(\frac{\partial W_r}{\partial \zeta}) \tag{62}$$

The last two terms in the above equation represent the effect of the decision ζ on the workload of the nodes. These work increments can be computed from:

$$W_l(before - decision) - W_l(after - decision) = \zeta_{stay}(t_{FT} + t_E + \rho_l) \tag{63}$$

This yields:

$$\frac{\partial W_l}{\partial \zeta} = (t_{FT} + t_E + L_l). \tag{64}$$

Similarly we have:

$$\frac{\partial W_r}{\partial \zeta} = \begin{cases} t_M + t_E + L_r & \text{if n is a host node,} \\ t_M + t_E + L_r + 2 * t_{FT} & \text{if n is a foreign node.} \end{cases} \tag{65}$$

Substituting (64) and (65) in (62) we get:

$$\frac{\partial H}{\partial \zeta} = W_l(1+\zeta) - W_r(2-\zeta). \tag{66}$$

The algorithm can now be summarized as follows:

- Whenever a decision must be taken, i.e. when a new process is created, first set ζ_0 and H_0 to the most recently stored values of ζ (or threshold value initially) and of the cost function at the host node, respectively.
- The gradient descent is initiated and at any hth step of the algorithm, ζ_h is computed, using Equations (58) and (66). Then the cost H_h is computed based on ζ_h.
- This is repeated until the point of diminishing returns, i.e. until $|H_h - H_{h-1}| \leq \epsilon$.
- The resulting value of ζ is used for the decision. Clearly if ζ is less than 0.5 the process is moved. Otherwise the process is executed locally.

Obviously this does not apply to the NP (no process migration), or the RD (random two) policies, which are both static in that they do not use on-line load information to make decisions.

For the AD-TWO policy, where we choose between the local and host nodes, the simple computation of ζ suffices to determine the choice of a site to execute the process.

However for the AD-ALL policy, where a choice must be made among all nodes, this gradient iteration will compute the alternative between a local node and *every* other node; thus the computation is $J-1$ times more time consuming than the AD-TWO policy. Let ζ^j denote the probability that the local node is chosen over some other node j. Then the node j^* with the *smallest* value of ζ^j will be chosen for remote execution *if* the corresponding probability is less than 0.5; otherwise the execution will be local.

5.6 Performance metrics and experimental comparison of policies

The four policies discussed in Section 5.4 are compared in this Section with respect to the following set of pragmatic performance criteria [32].

- *Hit ratio:* This is the ratio of correct decisions to the total number of decisions, in the sense that the effective response time of the execution is smallest with the decision taken.

- *Overhead:* This is the fraction of CPU time consumed by the load balancing algorithm's execution time only.

- *Quality:* This metric measures the distance of some particular load balancing policy X from the NP (no-process-migration) strategy; it is defined by $Quality(X) = \frac{W(NP)-W(X)}{W(NP)}$,

  where $W(.)$ is total process response time. A positive $Quality(X)$ indicates that

policy X is worth being pursued, and the larger it is the more useful X will be. However if $Quality(X)$ is negative, policy X will be detrimental to system performance and should not be considered.

- *Percentage Remote Execution:* This measures the percentage of processes executed on remote (foreign and host) nodes.

Earlier studies [77, 93, 38] have supported the view that for an adaptive load balancing algorithm to be of practical interest, the percentage of remote execution should be limited, say of the order of 10%, and the hit ratio should be high, say greater than 90%. Since these studies considered systems which allowed only process migration and not data transfer, we can argue for a higher percentage of remote executions in our case.

Other, more conventional, performance measures which can easily be obtained on the nCube environment will also be considered here. They include:

- Average node response R_n; this is the response time obtained by any request for service at a node (including processes and messages),

- Average process response time R_p; this is just the response time, including process migration, experienced by processes.

5.6.1 Experimental results

The nCube provides an ideal test environment for implementing such adaptive load

balancing algorithms. We have therefore tested all policies on a 5 node nCube with an artificially generated workload. The time measurements were carried out in "ticks" where one tick is 128 micro-secs. The delay for sending a message from one node to another is of less than one tick.

The AD-TWO and AD-ALL results are based on the simplest possible form of the load-balancing algorithm since *only one step of the gradient descent* is used, requiring the execution of 8 instructions only. A very simple decision rule is used, based *only* on deciding to use the remote node if ζ is reduced by one step of the iteration.

In this section we present a first set of experimental results for the following workload:

- Each task or process's execution time is uniformly distributed between 9,000 and 12,000 ticks. This corresponds to an average execution time of 1.34 seconds of CPU time. Process transfer takes 6,000 ticks. Each message transfer from one node to another will take up 4 ticks (0.512 milliseconds).

- Each node creates one process, and will create a new process as soon as it is informed that the previous one it created has finished execution.

- File sizes are uniformly distributed between 1K and 4K bytes so that file transfer times are uniformly distributed between 1000 and 4000 ticks. This results in an average file transfer time of 0.32 seconds.

Table 9: Comparative efficiency and performance for four load balancing policies. All times are given in ticks. AD-TWO is the best, followed by AD-ALL. RD is the third and NP the worst.

Strategy	% Remote Exec.	Overhead	R_n	Quality	R_p
NP	not relevant	not relevant	3,838	not relevant	19,355
AD-ALL	34%	0.00082	1,977	0.48	18,568
AD-TWO	30%	0.00079	1,923	0.50	18,082
RD	42%	not relevant	3,352	0.13	19,037

- The load indices are computed as indicated in Section 5.5. They are broadcast every $t_B = 50,000$ ticks or 6.4 secs.

- Initially, 500 files are allocated to each site; then on an average 20 files are created at each node every 2,500,000 ticks and are deleted at the same rate.

More detailed experimental evaluations are presented in the next sections. Table 9 summarizes measurements which were collected for a total of 1500 process creations at each node.

The *Average Node Response Time*, R_n is an average over *all* requests made to a node. *Quality* expresses the improvement over the NP policy. These are the primary measures of interest. We see that the *Average Process Response Time*, R_p is less sensitive to the policy; this is simply because process migration time is an important component in the AD-ALL and AD-TWO policies. Clearly, load balancing is always beneficial. Note that the overhead presented only includes the algorithm's computations, and is therefore very limited.

The ADAPTIVE-TWO-NODES strategy compares favorably with the ADAPTIVE-ALL-NODES strategy, since the results it yields are slightly better but the overhead is also much less. This supports our claim that simple load balancing policies are better than sophisticated ones, and much better than not having load balancing at all.

For the ADAPTIVE-ALL-NODES policy, we noticed that only a very small fraction (1%) of the processes were transferred to foreign nodes. Thus, the effort overhead of the AD-ALL policy only benefits these 1% of the processes.

This observation validates the assertions made earlier [77], and encourage us to consider restricted load balancing policies. However, the results concerning AD-TWO and its comparison with NP also indicate that load balancing can be effectively used in order to improve user and system performance.

The results of Table 10 illustrate the advantage of file redistribution which we used in all policies. To do so, we modify the AD-TWO strategy so that the system does *not* redistribute the files; initially all the nodes have an equal number of files and as time proceeds files at all the nodes are created and destroyed such that nodes have different numbers of files, without redistribution. We call this the NO-FILE-REDISTRIBUTION (NO-FI-RE). The measurements shown below concerning this policy are taken at the nodes having the minimum and the maximum number of files, and the two values are shown. We clearly see the superiority of policies which do redistribute files, over one which does not adjust file distribution.

Table 10: Comparative efficiency and performance with and without file redistribution.

Strategy	% Remote Exec.	Hit Ratio	Overhead	R_n	Quality
AD-TWO	30%	0.97	0.00070	1,923	0.50
NO-FI-RE	40%-29%	1.0 0-0.97	0.00070	2,100	0.33

On Figures 20 and 21 we present measures over time (in milliseconds) of the total load at a node. On Figure 20 we compare the system running under AD-TWO with the system running under the NP policy. On Figure 21 we compare the system running with the AD-TWO policy, and the system running with the RD policy. The parameters are the same as those for the experiments described above. Very clearly, the AD-TWO policy is very effective in reducing total load at each node.

The results of Figures 20 and 21 are not altogether intuitive. Indeed, Figure 20 indicates that the AD-TWO policy results in lower load on nodes than no load balancing (NP). Yet one would expect that load balancing would not affect average node, and only reduce differences between heavily loaded and lightly loaded nodes. What is being observed however is an overall improvement of performance, under the effect of load balancing. The same can be said about Figure 21.

On Figure 22 we compare the four policies considered when the average execution time C of a process is varied. Process execution time is uniformly distributed as follows: it is between [C-200, C+200] if $200 \leq C$; for smaller values of C it is uniformly distributed between 0 and 2C.

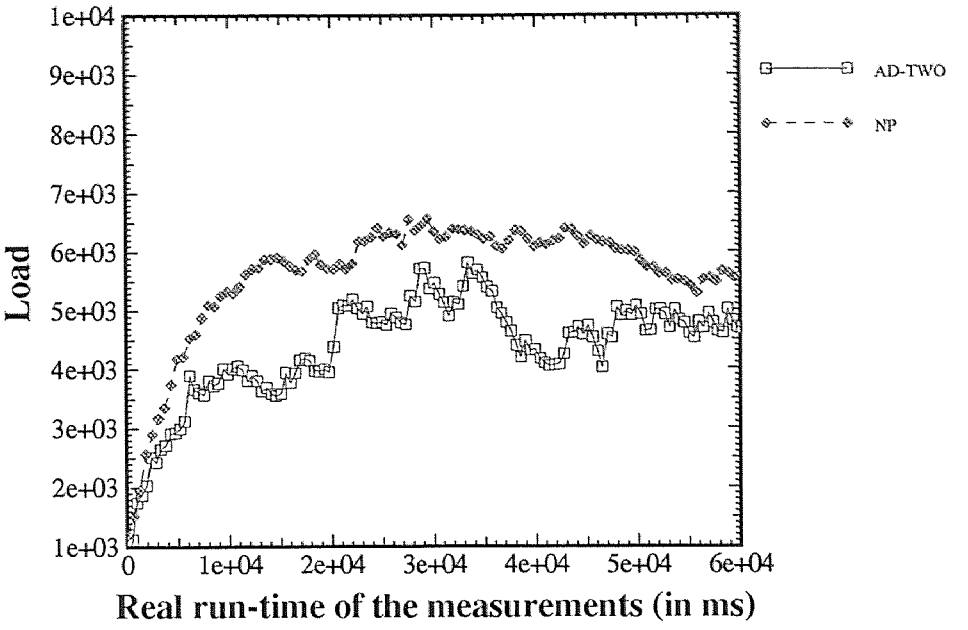

Figure 20: Comparison of the instantaneous load on a node for the AD-TWO and NP policies over a long period of time. The AD-TWO policy provides uniformly better performance.

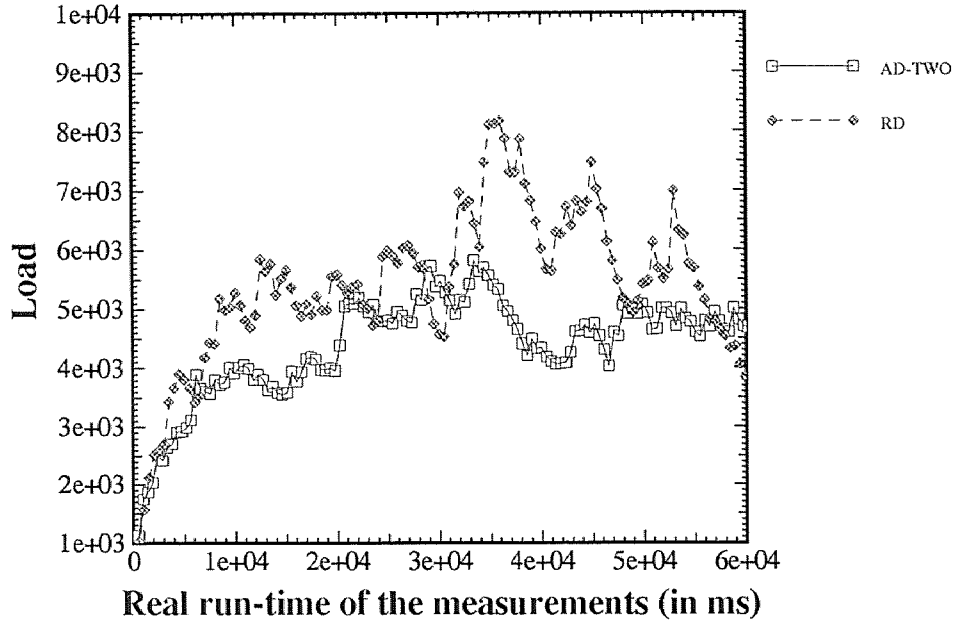

Figure 21: Comparison of the instantaneous load on a node for the AD-TWO and RD policies over a long period of time. The sporadic effect of the RD policy, and the improvement obtained with AD-TWO are clearly apparent.

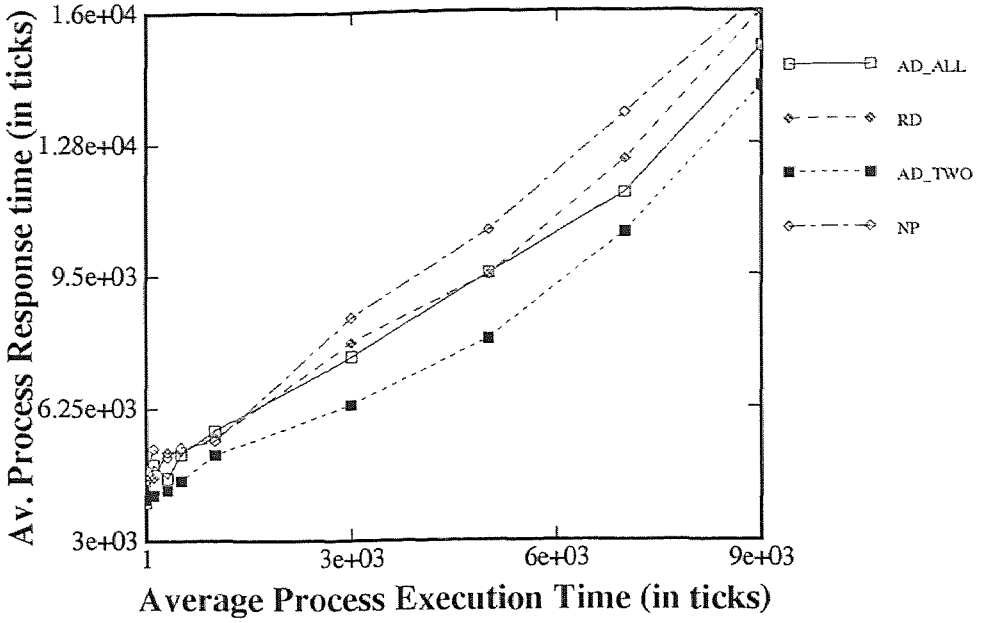

Figure 22: Average process response time as a function of average process execution time C for the four policies. For medium to high average process execution times, AD-TWO provides superior performance.

The workload is being changed by varying the average run time of each process, and each point on the curve corresponds to the average process response time measured for 1500 process executions. The measurements are taken under the following load conditions: each PU generates a new process *as soon as* the previous process it has created has completed execution, and the PU has received information about this. Thus there is exactly one active process running for each PU in the system. We see that when the average process execution time is small, the four policies are relatively equivalent, though AD-TWO still remains the best. However when the average process execution time is large, there is a great benefit in choosing the best load balancing policy.

5.6.2 Response Time Comparisons with Different Load Values

In the experiments described above, recall that each site is allowed to have a single active process, and a new process is created only when the previous one is completed. This workload does not allow us to vary the load on the system, except by modifying the average execution time of processes.

In this section we present results of experiments which we have conducted by increasing the number of processes generated at each node. To do so, each node is allowed to generate synthetic processes, on the average each X milli-seconds or at a rate of $\phi = 1/\mathrm{X}$ processes per unit time. A process will execute on the average for C time units, and we will also present measurements obtained by varying C. Of course, all these times are specified at the system level in "ticks". All other parameters are the same as those indicated in the previous subsection.

In all cases reported here, for each value of ϕ, response time and overhead measurements are collected for 1000 process executions.

Measurements are collected for the process response time R_p for all four policies considered: NP, AD-TWO, AD-ALL and RD. Some results are reported on Figure 23. Here the process generation rate is 55 processes/sec, or 7.04 processes per 1000 ticks, at each node. For small values of C under 200 ticks (which is of the order of 25 milli-seconds) AD-ALL, AD-TWO and RD seem to provide equivalent performance. For larger values of C, AD-TWO provides superior performance.

We have measured the overhead for AD-TWO and AD-ALL and report it

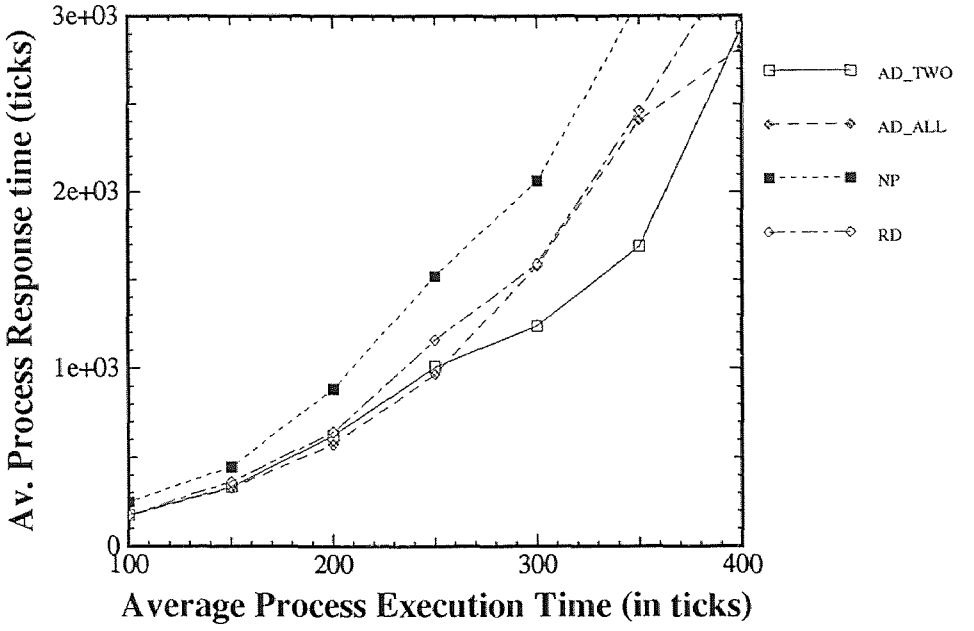

Figure 23: Average process response time as a function of average process execution time C for the four policies. Here the process generation rate is 55 processes/sec, or 7.04 processes per 1000 ticks, at each node. For low C, AD-ALL, AD-TWO and RD seem to provide equivalent performance. But for medium to high C, AD-TWO provides superior performance.

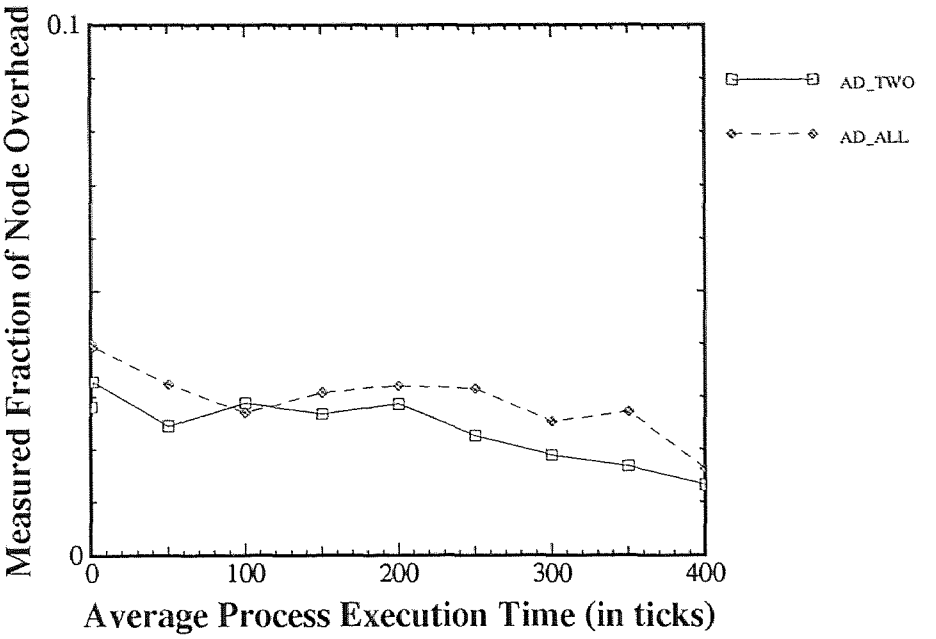

Figure 24: Overhead as a function of average process execution time C for two AD-TWO and AD-ALL. The process generation rate is 55 processes/sec, or 7.04 processes per 1000 ticks, at each node. Overhead decreases as C increases. As expected AD-ALL has systematically higher overhead than AD-TWO. Overhead is under 5% (0.05) in all cases.

on Figure 24. This is the fraction of node processing time devoted to the control policy including load data collection and broadcasting, and carrying out the gradient algorithm. We see that overhead remains acceptably low, at less than 5%.

On Figure 25 we report measurements where we keep C constant at 1050 ticks or 0.1344 seconds, with uniform distribution between 900 and 1200 ticks. We vary ϕ, the process creation rate. The average process response time as a function of average process creation rate is plotted for the four policies. For very low ϕ, AD-ALL, AD-TWO and RD provide equivalent performance. For medium to high ϕ AD-TWO is the best, very closely followed by AD-ALL. Both are much better than RD and NP.

Finally on Figure 26 we present measurements of the fraction of node overhead

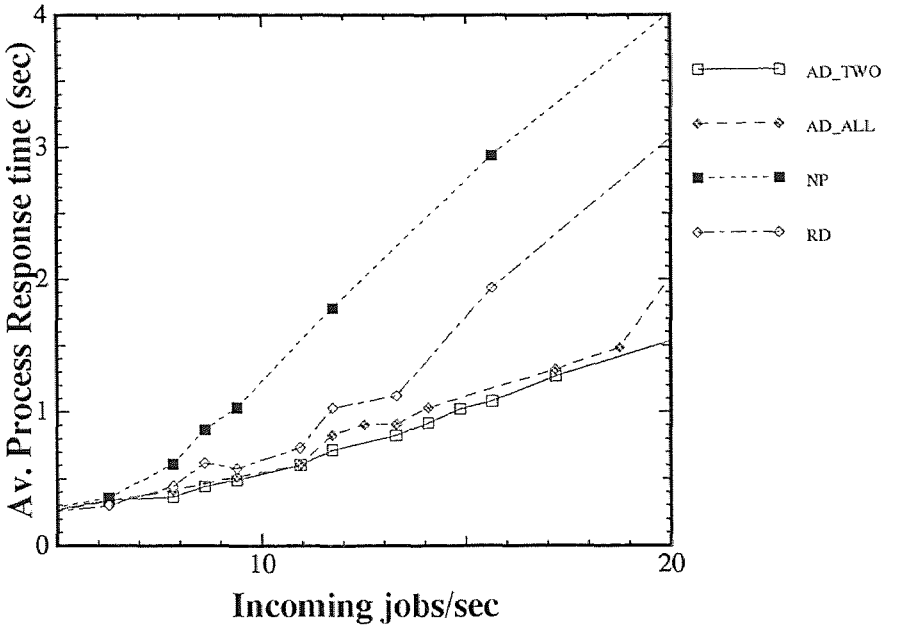

Figure 25: Average process response time as a function of average process creation rate for the four policies. The average process execution time C is constant at 1050 ticks or 0.1344 seconds. For varying process creation rates, AD-ALL, AD-TWO and RD provide equivalent performance. For medium to high rates, AD-TWO is the best, and is closely followed by AD-ALL. Both are much better than RD and NP.

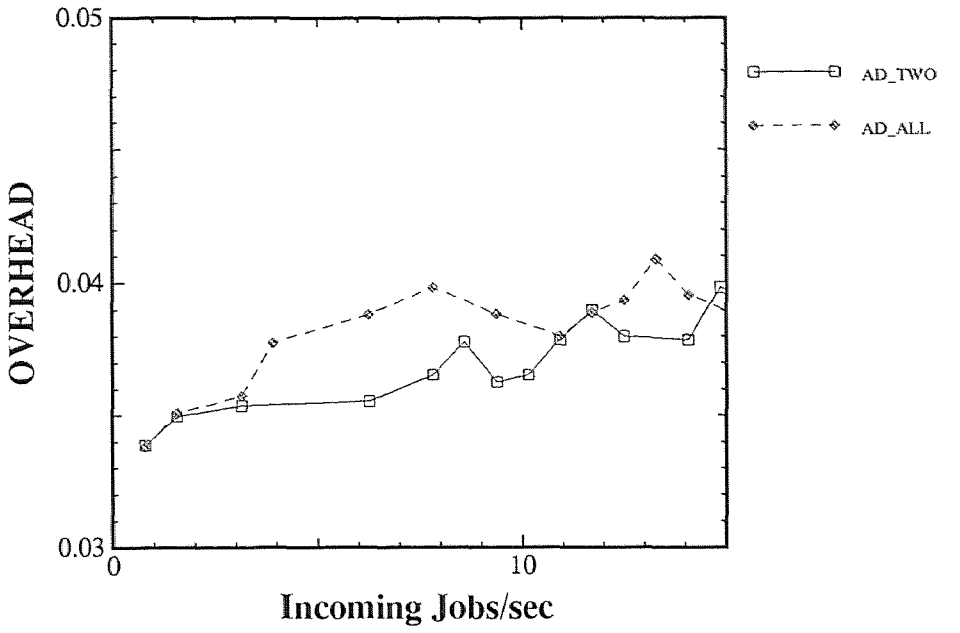

Figure 26: Overhead as a function of the process creation rate at each node, for the two policies adaptive policies AD-TWO and AD-ALL. The average process execution time C is constant at 1050 ticks or 0.1344 seconds. As expected AD-ALL has systematically higher overhead than AD-TWO. Overhead is under 5% (0.05) in all cases.

as a function of load, in number of processes created per unit time at the nodes. This comparison is restricted to the two adaptive policies, even though the random RD policy would also generate some overhead. Clearly AD-TWO results in less overhead, but for both AD-TWO and AD-ALL overhead is relatively low (below 5%) even at relatively high loads.

5.7 Conclusions

In this chapter we have presented a dynamic load balancing principle based on on-line gradient descent for distributed system architectures. The purpose is to substantially improve system performance using adaptive decisions concerning the choice of pro-

cessing units where processes are executed.

We have implemented and tested two gradient-based load balancing algorithms on an nCube target architecture, and compared them with the case where no load balancing is carried out, as well as with random load balancing. We demonstrate via extensive measurements that simple adaptive load balancing can substantially improve distributed system performance, over that of systems with no load balancing. A 50% reduction in average process response time is observed compared to no load balancing.

We also indicate experimentally that a simple algorithm can achieve better performance than more sophisticated algorithms, and that it obviously results in less overhead.

CHAPTER 6

SUMMARY AND CONCLUSION

In this chapter, we summarize the general principles and major features of our methodology, discuss its applicability to performance modeling and design evaluation of distributed systems, assess its merits with respect to computational efficiency and generality, compare it with other currently available methods and finally, provide suggestions for further research in this area.

6.1 Summary of the Methodology

We have drawn upon the concepts of *queueing network* techniques and load balancing policies and have combined them in an original way to produce a foundation for an effective modeling methodology. Execution environment, system behavior, allocation of static components and dynamic task assignment are considered separately. Queueing network elements are used to model the operation of physical components of a distributed system. Customers of different classes are used to represent the user requests, tasks, files, and messages. We have utilized the principles of the BCMP theorem to design an efficient performance prediction method. This is done by approximation of each site of the distributed system by a properly chosen service center having an exponential service time distribution. The customers change classes as they circulate among service centers and follow a particular chain. The behavior of each

customer class is predicted and various performance measures are estimated. The method described is general and can be used for estimating behavior of any message passing distributed memory system.

To improve the performance we used a static load balancing strategy which optimally allocates files at different nodes. The file allocation algorithm is designed using closed queueing networks. The file allocation problem is formulated as a routing problem in multiple chain networks. We optimize *Average node response time*, which is the average over all requests made to a node. The relative throughput at each node is established as a function of file placement. We then compute the derivative of the average response time with respect to relative throughput at each node and show that it can be easily obtained from MVA algorithm. The *steepest descent direction* is obtained by summing the derivatives over all closed chains intersecting at a node. A closed network version of flow deviation algorithm is then introduced to find the optimum file allocation in multiple chain queueing network. The performance measures are obtained by constructing and solving a multiple class multiple chain closed queueing network. The complete model, along with file allocation algorithm is simulated on nCube. Each processor of nCube is programmed to behave as a node of the distributed system. The analytical results are validated against the one measured from nCube.

Novel adaptive load balancing policies are designed and evaluated for process and data migration. We introduced a gradient descent paradigm to compute on

line load balancing decisions. We implemented and tested these algorithms on the nCube architecture, and compared it with the case where no load balancing is carried out, as well as with random load balancing. We proved that simple adaptive load balancing algorithms can substantially improve distributed system performance over that of systems with no load balancing. We also showed that simple algorithms can achieve better performance than more sophisticated algorithms, and that they result in less overhead. Numerous measurement results on an nCube are presented using performance metrics which have already been discussed in the literature.

In short, to achieve high performance in distributed systems we have proposed a modeling methodology which combines the static and dynamic policies for distributing data, processes and control on distributed hardware.

6.2 Application Considerations

Our methodology can be applied to design virtually any message passing distributed system. The applicability is limited to neither specific system architecture nor to system functionality. The same general principles can be utilized for each individual case. When representing the system architecture of a particular execution environment, a modeler does not have to be concerned about performance at the preliminary stage. All that needs to be known are the number of nodes, their interconnections, the information units and how they interact. Next, either existing or newly designed load balancing policies can be used to optimally distribute various information units

for performance improvement. The policies could be either static or dynamic or a combination of both. We infer that hybrid policies which combine optimal static methods with dynamic solutions are especially beneficial.

By properly using the set of tools provided by our methodology, designers and system planners can evaluate the performance of a proposed system with several benchmark applications in a cost-effective way, enabling them to 'pilot' their design to meet specific objectives. Analysts can use the same methods for experimenting with various parameters within a system (without disturbing the standard configuration and/or operation of the system itself) to optimally adjust resource utilizations and response times or to determine system bottlenecks. With dynamically reconfigurable architectures, optimal (or nearly optimal) configurations for specific applications can be quickly determined. In case of a static and dynamic policy being employed by a system, one can "test" a number of potential algorithms and adopt the one yielding the best performance.

It is important to emphasize that the solutions obtained by applying the analytic techniques presented in this thesis are estimates of the actual values and should be treated as such. The quality of approximation will vary with each particular case and in general cannot be determined a *priori*. However, we do believe that in most cases the results yielded by our methodology will provide a reasonable indication of the relative performance of the system being considered with respect to competing architectures or different parameter selections. Thus, we feel that this methodology is

best suited for the kind of systems where relative merits of alternate design proposals and different system configurations are being compared.

6.3 Comparisons with Other Methods

Compared to general-purpose simulation packages, our methodology requires substantially less processing time in both the model development and solution phases. In virtually all cases, the design and implementation of a model using a general-purpose simulation language would be significantly more time-consuming than the construction of queueing network model. Even simulation on an nCube is less time consuming, because of the efficient message passing libraries for designing communication mechanism. Moreover, for homogeneous systems one program can be replicated on all the other nodes in an nCube programming environment.

It should be noted, however, that our analytic results are only estimates of the 'actual' values and that no guarantee of achieving a specific accuracy level can be made. Thus, our analytical methodology is no substitute for detailed simulation in cases where high accuracy is of utmost importance. However, as evident from our measurements on the nCube, the available empirical data indicates that its overall level of accuracy is sufficient. As far as performance evaluation of distributed systems is concerned, the potential range of applications of our analytic techniques is comparable with that of most general-purpose simulation tools. On an nCube practically any distributed system can be simulated.

The considerations of computational efficiency and accuracy discussed above are also applicable to special-purpose simulators. The high development cost of such simulators is also an important factor. Furthermore, the generality of application of our analytic and simulation methods far exceeds that of an individual special-purpose simulator. These simulators, though able to produce highly accurate performance statistics, are particularly effective only when the same system is frequently evaluated with different programs or when intricate architectural or operating system details are to be included in the model.

We will now consider other currently available analytic methods. Graph methods are usually applicable to modeling only very simple precedence relationships, eg., fork-join constructs, and cannot capture the behavior of complex distributed systems. As far as graph-based methods go, there are two basic categories. Those in the first category assume either an infinite capacity of each system resource or some other overly simplified architecture. Such methods, although usually reasonably accurate, are very limited in their scope of practical application since they are not capable of modeling (much more complex) features of realistic systems – which are easily handled by our methodology. In the second category, all architectural details included in a model are represented by nodes and arcs in a graph, intermixed together with the representation of a program's precedence relationships. Such techniques, *e.g.*, Stochastic Petri nets, are prone to rapid state space explosion as the number of system components increase. (In our methodology, the complexity of computing also

increases with system size, but not nearly as fast!). Also, a minor architecture change may require that a completely new graph be constructed and solved.

6.4 Suggestions for Future Research

From the material presented in the previous chapters, we can see that research already accomplished has resulted in the development of an original and viable methodology for performance modeling of distributed and parallel systems. The modeling methodology is general and can be applied to any class of distributed systems. One such class of systems are *distributed object oriented systems.* We will briefly describe our approach to develop and design a performance model in this paradigm.

Furthermore, there are still number of interesting and important issues left to be resolved, worthy of further investigation. We will identify the more important of these issues and discuss possible research approaches.

6.4.1 Distributed Object-Oriented Model

Object oriented analysis endeavors to model a situation in terms of a collection of interacting entities, each of which provides a well-defined set of behaviors and attributes. If a system is described in words, the *nouns* correspond to objects, the message interface is determined from the *verbs*, and the logical properties are derived from *adjectives* [126]. An object encapsulates data, processing logic, and communicates via messages. We propose to efficiently distribute the objects to processing

sites, such that communication overhead is reduced and performance improved.

The system we refer to operates as follows: whenever an object receives a message it updates or changes state and then replies back with a message, if necessary. Depending on the message class an object invokes a particular *method*. The method can request information from one or more other objects. These objects may or may not reside at the same site. Such objects in turn again send messages to other objects. The object behavior is completely independent and encapsulated. We introduce the notion of *depth* in an object-oriented system which is the number of objects involved in processing a user request (user is an object also).

If objects are placed on different sites and sites are modeled as service centers of a queueing network model, the behavior of a user request can be represented as a chain as described in Chapter 3. The queueing discipline and the service demands can be assumed such that the system satisfies product form requirements. Various performance measures can be extracted using the prediction methodology described in the thesis.

We simulated a simple object model on the nCube, where one object was assigned per node. The user on each node requests some service, and the user request (message) is sent to an appropriate object. The object processes the message, may or may not request service from other objects, and returns a message to the user (object). The performance measures such as throughput and response time were measured as a function of think time and depth. The results obtained validates the

analysis obtained from the prediction method.

The performance modeling of object oriented system suggests number of interesting research issues. For further research in this area we would like to address following:

- Object classification and their relationship identification;
- Since objects can be dynamically created and destroyed, techniques to efficiently distribute them are needed for performance improvement;
- To balance the load, static, dynamic or hybrid policies are required for different classes of objects; and
- Since the object model is very general, it is important to design techniques which can map any (nearly all) applications to the object model. This can lead to the development of software engineering environment, where all distributed applications specified in one common language, could be mapped to a general distributed object model.

6.4.2 Other Issues

This thesis provides evidence that optimal static solutions are consistent with adaptive solutions in the case of distributed systems. However, additional, more systematic experimentation in this direction is required to corroborate the claim. The literature has seen an abundance of static and adaptive solutions for various distributed system

models; it may be worthwhile to attempt to develop computationally feasible methods which combine the two types of solutions for performance improvements.

If the physical model consists of many service centers with deterministic service times, then the BCMP theorem may not always yield sufficiently accurate results. In light of this observation, it is desirable to investigate other ways of modeling the distributed systems which considers general service time distribution and/or different queueing disciplines. In Chapter 4 and 5 we used approximate methods to compute the steady state probabilities for the solution procedures. It would be beneficial to identify other accurate constructs to determine these steady state probabilities.

We have also not yet been able to develop a general procedure for theoretically determining the accuracy level of the results yielded by our methodology. It is important to investigate whether it is possible to place theoretical error bounds on our analytic estimates, as a function of architecture of the system being modeled. Using such bounds, a modeler can determine the range of application spectrum where the accuracy of our methodology is sufficient, the range where it is marginal and the range (if any) where it is unacceptable.

APPENDIX A

To prove: In a product form closed queueing network model with single server fixed capacity (SSFR) center:

$$\frac{\partial(\sum_{i=1}^{N}\rho^i G(N-1)}{\partial y_j} = \frac{G(N)}{y_j}\sum_{i=0}^{N}\rho^i L_j(N-i) \tag{67}$$

NOTE: Assuming a single closed chain network for simplicity, the same approach can be used for multiple chain networks.

$$\frac{\partial\sum_{i=1}^{N}\rho^i G(N-1)}{\partial y_j} = \frac{\partial\rho G(N-1)}{\partial y_j} + \frac{\partial\rho^2 G(N-2)}{\partial y_j}\cdots + \frac{\partial\rho^N G(N-N)}{\partial y_j} \tag{68}$$

$$\begin{aligned} = \; & \rho\frac{\partial G(N-1)}{\partial y_j} + G(N-1)\frac{\partial\rho}{\partial y_j} + \rho^2\frac{\partial G(N-2)}{\partial y_j} + G(N-2)\frac{\partial\rho^2}{\partial y_j} + \cdots \\ & + \rho^N\frac{\partial G(N-N)}{\partial y_j} + G(N-N)\frac{\partial\rho^N}{\partial y_j} \end{aligned} \tag{69}$$

where $\rho = y_j/\mu_j$ and from Equation (36) $\frac{\partial G(N)}{\partial y_j} = \frac{G(N)L(N)}{y_j}$

$$\begin{aligned} = \; & \left[\frac{\rho G(N-1)L_j(N-1)}{y_j} + \frac{\rho^2 G(N-2)L_j(N-2)}{y_j} + \cdots + \frac{\rho^N G(N-N)L_j(N-N)}{y_j}\right] \\ & + \left[G(N-1)\frac{1}{\mu} + G(N-2)\frac{2\rho}{\mu} + \cdots + G(N-N)\frac{(N)\rho^{(N-1)}}{\mu}\right] \end{aligned} \tag{70}$$

$$= \frac{1}{y_j}\left[\rho G(N-1)L_j(N-1) + \rho^2 G(N-2)L_j(N-2) + \cdots \rho^N G(N-N)L_j(N-N)\right]$$
$$+\frac{1}{y_j}\left[\rho G(N-1) + \rho^2 G(N-2) + \cdots + \rho^N G(N-N)\right] \quad (71)$$

$$= \frac{1}{y_j}\left[\rho G(N-1)[1 + L_j(N-1)] + \cdots + \rho^N G(N-N)[1 + L_j(N-N)]\right] \quad (72)$$

from Equation (31)

$$= \frac{1}{y_j}\left[\rho G(N-1)\mu T_j(N) + \cdots + \rho^N G(N-N)\mu T_j(N-(N-1))\right] \quad (73)$$

Substituting Equation (30) in (73)

$$= \frac{1}{y_j}\left[\sum_{i=1}^{N} \rho^i G(N-i) + \rho \sum_{i=1}^{N-1} \rho^i G(N-i) + \cdots + \rho^N \sum_{i=1}^{N-N} \rho^i G(N-i)\right] \quad (74)$$

$$= \frac{G(N)}{y_j}\left[\frac{\sum_{i=1}^{N} \rho^i G(N-i)}{G(N)} + \rho\frac{\sum_{i=1}^{N-1} \rho^i G(N-i)}{G(N)} + \cdots + \rho^N \frac{\sum_{i=1}^{N-N} \rho^i G(N-i)}{G(N)}\right] \quad (75)$$

From (29)

$$= \frac{G(N)}{y_j}\left[L_j(N) + \rho L_j(N-1) + \cdots + \rho^N L_j(0)\right] \quad (76)$$

$$= \frac{G(N)}{y_j}\sum_{i=0}^{N} \rho^i L_j(N-i) \quad (77)$$

APPENDIX B

Comparison with other Methodologies

The modeling methodology described in the thesis contributes significantly towards the design of high performance distributed and parallel systems. The *ability to predict and optimize performance*; and *the usage of hybrid policies for performance improvement* are the most significant.

Since development of large distributed systems is time consuming and expensive, the mechanism to model such systems and analyze their behavior at the design stage is certainly beneficial. Dissertation also provides evidence to claim that hybrid policies, which combines static load balancing solutions with dynamic policies do improve the performance in distributed system.

Ni and Hwang [84] also proposed a static load balancing scheme to enhance the performance in a multiple processor system. They formulated load balancing as a nonlinear programming problem with linear constraints. An optimal probabilistic algorithm is proposed to solve this nonlinear programming problem. In contrast, we use queueing network models to design the file allocation algorithm and performance prediction method. Among all the analytical methods, queueing methods are known to be more attractive (see chapter 2) because of their simple applicability and ease of computation. The mathematical foundations of queueing models also allow easy calculation of various performance measures.

The performance of the multiple processing system, as proposed by Ni and Hwang, has been analyzed by modeling the distributed system as a set of N parallel queues which represent processing units and a central dispatcher which distributes load among these queues. Such systems are less reliable and dispatcher node is the bottleneck to the incoming jobs. The control of such a system is non-distributed. Our approach is to model the systems where the control is distributed among processing sites.

Singh and Krouse [73] designed a distributed load balancing algorithm for distributed system to improve performance. The static algorithm proposed distributes jobs among various processing sites, using queueing techniques. They assume that either the jobs are entirely computational and do not need data for execution; or process and data are one unit which can be migrated to any site. Since the jobs are not entirely computational, they need data to work on and if this data can not be replicated on all the nodes, we need strategies which could move data and/or processes among sites to get work done efficiently. We consider processes and data as separate components; and develop load balancing algorithms to optimally distribute both.

APPENDIX C

Comparison of AD-TWO Policy with Other Adaptive Algorithms

The efficacy of the AD-TWO adaptive policy, described in chapter 5, is established here by comparing it to the well known and widely referenced adaptive algorithms. These algorithms include early works of Stankovic [127], Eager [77] and more recent ones by Zhou [93]. Stankovic's work was one of the first studies of adaptive load-sharing algorithms, offering new insights into the problem and suggesting novel solutions. Later, Zhou's extensive experimentation established that adaptive algorithms improve performance in distributed systems.

Recently in [38], a general method for quantitative and qualitative analysis of adaptive load sharing algorithms is discussed. Study suggests that activities related to remote execution should be bounded and restricted to a small proportion of the activity in the system. They propose following efficiency measures for comparing different adaptive algorithms:

% Remote execution	The percentage of job executed on the remote nodes
Hit Ratio	Ratio of correct decisions to the total number of decisions made
Overhead	The fraction of CPU time consumed by the load balancing algorithm's execution

The properties of the different algorithms we consider are summerized below.

AD-TWO	
Decision making invocation	event driven (process creation)
Transfer policy	global load information, process characteristics, system parameters
Location policy	least loaded among local and host node
Acceptance policy	single request - no rejection allowed
Information policy	periodic load dissemination

Stankovic	
Decision making invocation	basic method, periodic and also event driven (application completion)
Transfer policy	local and indirect nonlocal information on a comparitive basis
Location policy	least loaded node
Acceptance policy	single request - no rejection allowed
Information policy	periodic state dissemination

Zhou	
Decision making invocation	event driven (application arrival)
Transfer policy	local information only (threshold)
Location policy	least loaded node
Acceptance policy	single request - no rejection allowed
Information policy	periodic state dissemination

Eager	
Decision making invocation	event driven (application arrival)
Transfer policy	local information only (threshold)
Location policy	Random
Acceptance policy	Request-reply to allow rejection (threshold)
Information policy	probing (request-reply) upon application arrival

The following table compares the efficiency measures for the four algorithms:

Comparison of Efficiency Measures

	% Remote Exe.	Hit Ratio	Overhead
AD-TWO	30	0.95	0.005
Stankovic	37	0.77	0.028
Zhou	22	0.77	0.006
Eager	16	0.60	0.063

Clearly, gradient descent rule for AD-TWO adaptive policy provides high hit ratio and less overhead. The % remote execution is better than algorithms by Zhou and Eager; and comparable to Stankovic's algorithm. This is because all of the above algorithms, except AD-TWO, deal only with task or process movement; they assume that jobs and data are one information unit and can migrate to any site.

References

[1] L. Kleinrock, "Distributed systems," *Communications of the ACM*, vol. 28, pp. 1200–1213, November 1985.

[2] P. Enslow, "What is a distributed data processing system ?," *IEEE Computer*, pp. 13–21, January 1978.

[3] G. LeLann, "Motivation, objectives and characterization of distributed systems," in *Distributed Systems - Architecture and Implementation* (B. Lampson, M. Paul, and H. Siegert, eds.), Springer-Verlag, 1983.

[4] S. Mullender, *Distributed Systems*. Addison Wesley, 1989.

[5] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Transaction on Computers*, vol. C-21, pp. 948–960, September 1972.

[6] M. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, 1966.

[7] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*. New-York: McGraw-Hill, 1984.

[8] *MasPar System Overview*, March 1991. Document Order No. 9300-0100 A3.

[9] *Connection Machine Programming in C**, Nov 1990. Version 6.0.

[10] *nCUBE Processor Manual*, Dec 1990. Part No. 101636.

[11] *iPSC/2*, 1988. Document Order No. 280110-001.

[12] *Alliant FX/Series Product Summary*, Dec 1988.

[13] J. Dennis and D. Misunas, "A preliminary architecture for a basic data flow processors," in *Proceedings of IEEE Symposium on Computer Architecture*, p. 291, May 1975.

[14] Arvind and V. Kathail, "A multiple processor dataflow machine that supports generalized procedures," in *Proceeding of 8th ACM Symposium on Computer Architecture*, pp. 291, May 1981.

[15] M. Amamiya *et al.*, "A dataflow processor array system for solving partial differential equations," in *Proceedings of International Symposium on Applied Mathematics and Information Science*, (Kyoto University), March 1982.

[16] W. Ackerman, "Data flow languages," *Computer*, vol. 15, pp. 15–25, February 1982.

[17] E. Gelenbe and I. Mitrani, *Analysis and Synthesis of Computer System.* Academic Press, 1980.

[18] A. Tanenbaum, *Modern Operating Systems.* Prentice Hall, 1992.

[19] A. Tanenbaum, *Computer Networks.* Prentice Hall, 1988.

[20] A. Kapelnikov, *Analytic Modeling Methodology for Evaluating the Performance of Distributed, Multiple-Computer Systems.* PhD thesis, UCLA, 1986.

[21] R. Watson, "Distributed system architecture model," in *Distributed Systems - Architecture and Implementation* (B. Lampson, M. Paul, and H. Siegert, eds.), Springer-Verlag, 1983.

[22] A. Birell and B. Nelson, "Implementing remote procedure calls," *ACM Transaction on Computer Systems*, vol. 2, pp. 39–59, February 1984.

[23] B. Liskov *et al.*, "Implementation of Argus," *Proc. of 11th Symposium on Operating systems Principle*, vol. 21, pp. 111–122, Nov 1987.

[24] M. Fridrich and W. Older, "The FELIX File Server," in *Proc. of 8th Symposium on Operating System Principles*, (Pacific Grove, CA), pp. 37–44, December 1981.

[25] B. Lyon *et al.*, "Overview of the SUN Network File System," in *Proceeding Unix Conference*, (Dallas), pp. 1–8, January 1985.

[26] G. Popek and B. Walker, *The LOCUS Distributed System Architecture.* MA: The MIT press, 1985.

[27] D. Reed and L. Svobodova, "SWALLOW : A distributed data storage system for a local network," in *Local networks for computer communications* (A. West and P. Janson, eds.), pp. 355–373, North-Holland Publishing Company, 1981.

[28] A. Litman, "Dunix - a distributed UNIX system," *Operating Systems Review*, vol. 22, pp. 42–50, January 1984.

[29] M. Brown, K. Kolling, and E. Taft, "The Alpine File System," *ACM Transactions on Computer Systems*, vol. 3, pp. 261–293, November 1985.

[30] M. Powell and B. Miller, "Process migration in DEMOS/MP," in *Proc. of 9th Syposium on Operating Systems Principles*, (Bretton Woods, N.H.), pp. 110–119, October 1983.

[31] R. Finkel *et al.*, "The Charlotte distributed operating system," Tech. Rep. 502, Univ. of Wisconsin-Madison Computer Sciences, 1983.

[32] D. Black, "Scheduling support for concurrency and parallelism in the Mach operating system," *IEEE Computer*, vol. 23, pp. 35–43, May 1990.

[33] A. Goscinski, *Distributed Operating Systems, Logical Design.* Addison Wesley, 1991.

[34] U. Borghoff and K. Nast-kolb, "Distributed systems: a comprehensive survey," Tech. Rep. TUM-I8909, Techn. Univ. Munchen, Munich, Germany, 1989.

[35] V. Mak and S. Lundstrom, "Predicting performance of parallel computation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 257–270, July 1990.

[36] D. Towsley, C. Rommel, and J. Stankovic, "Analysis of fork-join program response time on multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 286–302, July 1990.

[37] E. Silva and M. Gerla, "Queueing network models for load balancing in distributed systems," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 24–38, 1991.

[38] O. Kremin and J. Kramer, "Methodical analysis of adaptive load sharing algorithms," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, pp. 747–760, November 1993.

[39] A. Barak and O. Paradise, "Mos - a load-balancing UNIX," in *Proc. EUUG Autumn '88*, pp. 273–280, September 1986.

[40] D. Gifford, R. Needham, and M. Schroeder, "The Cedar file system," *Communications of the ACM*, vol. 31, pp. 288–298, March 1988.

[41] M. Satyanarayanan, "The ITC distributed file system: principles and design," in *Proceedings of the Ninth ACM Syposium of Operating Systems Priciples*, pp. 35–47, 1983.

[42] J. Ousterhout *et al.*, "The Sprite network operating system," *IEEE Computer*, vol. 21, 1988.

[43] M. Nelson, B. Welch, and J. Ousterhout, "Caching in the Sprite network file system," *ACM Transactions on Computer Systems*, vol. 6, 1988.

[44] J. Howard *et al.*, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, pp. 55–81, 1988.

[45] S. Carson and S. Setia, "Analysis of the periodic write policy for disk cache," *IEEE Transactions on Software Engineering*, vol. 18, Jan 1992.

[46] C. G. Gary, *Performnace and fault-tolerance in a cache for distributed file service.* PhD thesis, Stanford University, December 1990.

[47] A. Agarwal, "Performance tradeoffs in multithreaded processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, 1992.

[48] C. Ramamoorthy and K. Chandy, "Optimization of memory hierarchies in multiprogrammed systems," *JACM*, July 1970.

[49] S. Arora and A. Gallo, "Optimal sizing loading and re-loading in a multi-level memory hierarchy system," in *AFIPS Proceedings*, pp. 337–344, 1971.

[50] P. Chen, "Optimal file allocation in multi-level storage systems," *Proc. AFIPS National Computer Conference*, vol. 42, pp. 277–282, 1973.

[51] P. Chen and G. Mealy, "Optimal allocation of files with individual response time requirements," in *Proceedings of the 7th Annual Princeton Conference on Information Sciences and Systems*, (Princeton University), March 1973.

[52] L. Dowdy and D. Foster, "Comparative models of the file assignment problem," *ACM Computing servey*, vol. 14, pp. 287–313, June 1982.

[53] D. Brownbridge, L. Marshall, and B. Randell, "The Newcastle connection of Unixes of the world unite," *Software Practice and Experience*, vol. 12, pp. 1147–1162, Dec 1982.

[54] A. Siegal, *Performance in Flexible Distributed File Systems.* PhD thesis, Cornell University, February 1992.

[55] K. Levin, *Organizing Distributed Data Bases in Computer Networks.* PhD thesis, University of Pennslyvania, Sept 1974.

[56] H. Morgan and K. Levin, "Optimal program and data locations in computer netwroks," *CACM*, vol. 20, pp. 315–322, May 1977.

[57] S. Mahmoud, "Resource allocation and file access control in distributed information networks," Tech. Rep., Dep. Syst. Eng., Carleton University, Canada, Jan 1975.

[58] L. Laning and M. Leonard, "File allocation in a distributed computer and communication network," *IEEE Trans. Computers*, vol. C-32, pp. 232–244, 1983.

[59] D. Tabak, *Multiprocessors.* Englewood Cliffs, NJ: Prentice Hall, 1990.

[60] S. Majumdar, D. Eager, and R. Bunt, "Scheduling in multiprogrammed parallel systems," in *in Proceedings 1988 ACM SIGMETRICS Conf. Measurement Modeling Comput. Syst.*, pp. 104–113, 1988.

[61] E. Gelenbe and R. Kushwaha, "Incremental dynamic load balancing in distributed systems." Submitted for publication in MASCOTS' 94.

[62] A. Thomasian and P. Bay, "Analytical queueing network models for parallel processing of task systems," *IEEE Transactions on Computers*, vol. C-35, pp. 1045–1056, Dec 1986.

[63] D. Menasce and L. Barroso, "A methodology for performance evaluation of parallel applications on multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 14, pp. 1–14, 1992.

[64] A. Tantawi and D. Towsley, "Optimal static load balancing in distributed computer systems," *Journal ACM*, vol. 32, pp. 445–465, April 1985.

[65] F. Baccelli and A. Makowski, "Simple computable bounds for the fork-join queue," in *Proc. Conf. Inform. Sci. Syst.*, 1985.

[66] F. Baccelli, W. Massey, and D. Towsley, "Acyclic fork-join queueong networks," *JACM*, July 1989.

[67] C. Kim and A. Agrawala, "Analysis of a fork-join queue," *IEEE Trans. Comput.*, vol. 38, pp. 250–255, Feb 1989.

[68] R. Nelson and A. Tantawi, "Approximate analysis of fork-join synchronization in parallel queues," *IEEE Trans. Comput.*, vol. 14, pp. 532–540, April 1988.

[69] S. Setia, M. Squillante, and S. Tripathi, "Analysis of processor allocation in multiprogrammed parallel processing systems," Tech. Rep. CS-TR-2840, University of Maryland, College Park, Feb 1992.

[70] R. Nelson, D. Towsley, and A. Tantawi, "Performance anlysis of parallel processing systems," *IEEE Transactions on Software Engineering*, vol. 14, pp. 533–540, April 1988.

[71] A. Tantawi and D. Towsley, "A general model for optimal static load balancing in star network configurations," in *Proc. Performance'84, (Paris, Dec 19-21)*, (New York), pp. 277–291, North-Holland, 1984.

[72] E. Silva and M. Gerla, "Load balancing in distributed systems with multiple classes and site constraints," in *Proc. Performance'84*, pp. 17–33, 1984.

[73] J. F. Kurose and S. Singh, "A Distributed algorithm for optimum static load balancing in distributed computer systems," in *Proc. IEEE INFOCOM'86*, pp. 458–467, 1986.

[74] H.-C. Lin, J. R. Yee, and C. Raghavendra, "Optimal joint load balancing and routing in message switched computer networks," in *IEEE INFOCOM'88 Conference*, pp. 3C.2.1–3C.2.8, March 1988.

[75] L. Ni, "A distributed load balancing algorithm for point-to-point computer networks," in *Proc. of IEEE COMPCON*, pp. 116–123, 1982.

[76] L. Ni, C. Xu, and T. Gendreau, "A distributed drafting algorithm for load balancing," *IEEE Transaction Software Engineering*, vol. SE-11, pp. 1153–1161, October 1985.

[77] D. Eager, E. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transaction on Software Engineering*, vol. 12, pp. 662–676, 1986.

[78] T. Liu, "Dynamic load balancing algorithm in homogeneous distributed systems," in *Proc. of the Sixth Int. Conf. on Distributed Computing Systems*, pp. 216–222, May 1986.

[79] R. Mirchandaney, D. Towsley, and J. Stankovic, "Adaptive load sharing in hetrogeneous systems," in *Proc. of the Ninth International Conference on Distributed Computing Systems*, (Newport Beach, California), pp. 298–306, June 1989.

[80] R. Mirchandaney, D. Towsley, and J. Stankovic, "Analysis of the effects of delays on load sharing," *IEEE Transaction Computers*, vol. 38, Nov 1989.

[81] H.-C. Lin, G.-M. Chiu, and C. Raghavendra, "Performance study of dynamic load balancing policies for distributed systems with service interruptions," in *IEEE INFOCOM'91 Proceedings*, pp. 797–805, 1991.

[82] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," in *Proc. ACM Comput. Network Performance Symp.*, pp. 47–55, 1982.

[83] Y. Wang and R. Morris, "Load sharing in distributed systems," *IEEE Transactions on Computers*, vol. C-34, pp. 204–217, March 1985.

[84] L. Ni and K. Hwang, "Optimal load balancing in a multiple processor system with many job classes," *IEEE Transaction on Software Engineering*, vol. 11, pp. 491–496, 1985.

[85] H. Kobayashi and M. Gerla, "Optimal routing in closed queuing networks," *ACM Transactions on Computer Systems*, vol. 1, pp. 294–310, Nov 1983.

[86] C. Gao, J. Liu, and M. Railey, "Load balancing algorithms in homogeneous distributed systems," in *Proc. 1984 International Conference on Parallel Processing*, (Silver Spring, MD), pp. 302–306, IEEE Computer Society, 1984.

[87] K. Lee and D. Towsley, "A comparison of priority-based decentralized load balancing policies," in *Proc. Performance '86 and 1986 ACM SIGMETRICS Conf*, pp. 70–77, 1986.

[88] T. Yaun and H. Lin, "Adaptive load balancing for parallel queues," in *Proc IEEE International Conf. on Communications*, (Amsterdam), 1984.

[89] S. Lavenberg, ed., *Computer Performance Modeling Handbook*. San Deigo, CA: Academic Press, 1983.

[90] C. Sauer and K. Chandy, "Computer/communication system modeling with the research queueing package, version 2," Tech. Rep., IBM, T.J. Watson Research Center, Yorktown Heights, New-York, 1981.

[91] M. Vernon, *Performance Oriented Design of Distributed Systems*. PhD thesis, UCLA, Dec 1982.

[92] R. Thomas, *A Dataflow Architecture with Improved Asymptotic Performance*. PhD thesis, UCI, 1981.

[93] S. Zhou, "A trace-driven simulations study of dynamic load balancing," *IEEE Transactions on Software Engineering*, vol. 14, No. 9, pp. 1327–1341, Sept 1988.

[94] F. Darema-Rogers, "Parallel appilcations performance methodology," Tech. Rep. RC 14320, IBM, Yorktown Heights, May 1988.

[95] L. Kleinrock, *Queueing Systems, Vol II*. New York: John Wiley, 1976.

[96] I. Akyildiz, "Performance analysis of a multiprocessor system model with process communication," *Computer Journal*, vol. 35, pp. 52–61, 1992.

[97] F. Baskett, K. Chandy, R.R.Muntz, and F. Palacios, "Open, closed and mixed networks of queues with different classes of customers," *Journal of the Association for Computing Machinery*, vol. 22, pp. 248–260, April 1975.

[98] P. Heidelberger and K. Trivedi, "Queueing network models for parallel processing with asynchronous tasks," *IEEE Transaction Computers*, vol. C-31, pp. 1099–1109, Nov 1982.

[99] P. Heidelberger and K. Trivedi, "Analytical queueing models for program with internal concurrency," *IEEE Transaction Computers*, vol. C-32, pp. 73–82, Jan 1983.

[100] E. Gelenbe, "Performance analysis of the Connection Machine," in *Proc. ACM-SIGMETRICS Symposium on System Performance Evaluation*, pp. 183–191, May 1989.

[101] K. Shin, C. Krishna, and Y. Lee, "Optimal dynamic control of resources in a distributed system," *IEEE transactions on software Engineering*, vol. 15, Oct 1989.

[102] J. Mohan, *Performance of Parallel Programs: Model and Analysis*. PhD thesis, Carnegie-Mellon University, July 1984.

[103] A. Kapelnikov, R. Muntz, and M. Ercegovac, "A modeling methodology for the analysis of concurrent systems and computations," *Journal of Parallel and Distributed Computing*, vol. 6, pp. 568–597, 1989.

[104] J. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[105] Y. Shieh, D. Ghosal, S. Tripathi, and P. Chintamani, "Modeling of hierarchical distributed systems with fault-tolerance," *IEEE Transactions on Software Engineering*, vol. 16, pp. 444–457, April 1990.

[106] P. Chen and J. Akoka, "Optimal design of distributed information systems," *IEEE Transactions on Computers*, vol. C-29, pp. 1068–1080, Dec 1980.

[107] V. Norton and G. Pfister, "A methodology for predicting multiprocessor performance," in *Proc. 1985 International Conference on Parallel Processing*, pp. 772–781, Aug 1985.

[108] W. Gilio and W. Schroeder-Preikschat, "A new programming model for massively parallel systems." Submitted for Publication.

[109] B. Welch and J. Ousterhout, "Prefix tables: A simple mechanism for locating files in a distributed system," in *Proc of sixth conf. on Distributed Computing System*, pp. 184–189, May 1986.

[110] E. Gelenbe and G. Pujolle, *Introduction to Queueing Networks*. John Wiley & sons, 1987.

[111] J. Wong, *Queueing Network Models for Computer Systems*. PhD thesis, UCLA, 1975.

[112] M. Reiser and H. Kobayashi, "Queuing networks with multiple closed chains: theory and computational algorithms," *IBM Journal Research and Development*, vol. 19, pp. 283–294, May 1975.

[113] M. Reiser, "Numerical methods in separable queueing networks," Tech. Rep. RC 4145, IBM, IBM Thomas J. Watson Research Center, Yorktown heights, 1976.

[114] R. Wong and J. Wong, "Efficient computational procedures for closed queueing networks model," in *Proc. of Seventh Hawaii International Conference on System Science*, (Honolulu, Hawaii), pp. 33–36, Jan 1974.

[115] L. Kleinrock, *Queueing Systems, Vol I.* New York: John Wiley, 1975.

[116] M. Reiser and S. Lavenberg, "Mean value analysis of closed multi-chain queueing networks," *J. ACM*, vol. 27, pp. 313–322, April 1980.

[117] A. Agrawala, S. Tripathi, and G. Ricart, "Adaptive routing using a virtual waiting time technique," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 76–81, 1982.

[118] C. Brown and M. Schwartz, "Adaptive routing in central computer communication networks," in *Porc. IEEE Int. Comp. Commum.*, pp. 12–16, June 1979.

[119] Y. Chow and W. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Transaction Computers*, vol. 28, pp. 354–361, 1979.

[120] A. Avritzer *et al.*, "The advantage of dynamic tuning in distributed asymmetric systems," in *Proc. Infocom '90*, 1990.

[121] R. Gallager, "A mimimum delay routing algorithm using distribted computation," *IEEE Transaction on Comm*, vol. COM-25, pp. 73–85, 1977.

[122] T. Chou and J. Abraham, "Load balancing in distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-8, July 1982.

[123] G. Rao, H. Stone, and T. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Trans. Comput.*, vol. C-28, pp. 291–298, April 1979.

[124] H. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 85–94, Jan 1977.

[125] A. Leff and P. Yu, "An adaptive startegy for load sharing in distributed database environments with information lags," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 91–103, 1991.

[126] K. Rubin and A. Goldberg, "Object behavior analysis," *Communications of the ACM*, vol. 35(9), pp. 48–62, September 1992.

[127] J. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms," *Computer Networks*, vol. 8, pp. 199–217, August 1984.