DESIGNING AND BUILDING AN EFFICIENT

APPLICATION ON A UNIX NETWORK

FILE SYSTEM

By

JIAN LIU

Bachelor of Medicine

Capital University of Medicine

Beijing, P. R. China

1989

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2000

DESIGNING AND BUILDING AN EFFICIENT

APPLICATION ON A UNIX NETWORK

FILE SYSTEM

Thesis Approved:

Jacques LaFrance
Thesis Advisor

J Chandler

[illegible]

Wayne B. Powell
Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my thesis advisor, Dr. Jacques LaFrance, whose intelligent guidance, inspiration, and devotion to his work and students have made this work possible.

I would also like to thank my committee members, Dr. J. P. Chandler, Dr. H. K. Dai, for their support and invaluable suggestions.

My thanks go to my parents Maotang Liu and Shujun Yang for their constant love and support through my educational endeavors and in every aspect of my life. My gratitude also goes to the rest of my family for their support in everything I do.

My deepest gratitude goes to my wife, Mei Ju, for her never ending love.

Finally, I would like to thank the Department of Computer Science and all of those whom I befriended for their support while I pursued my dream.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	LITERATURE REVIEW	4
	2.1 Distributed Computing System	4
	2.2 Relative Work	7
III.	DESIGN AND IMPLEMENTATION	13
	3.1 Implementation Platform	13
	3.2 Design Features of the Application	15
	3.3 The Architecture of the Application	16
	3.4 Load Balancing Algorithm	19
IV.	RESULTS AND ANALYSIS	22
	4.1 The Algorithm Achieves Better Load Balancing	22
	4.2 Test Results	23
V.	CONCLUSIONS AND FUTURE WORK	30
	5.1 Conclusions	30
	5.2 Future Work	30
REFERENCES		31
APPENDIX		36

LIST OF FIGURES

Figure Page

Chapter III

1. The Architecture of the Application 18

Chapter IV

1. A Sample Summary Log File (Simple) 25
2. A Sample Summary Log File (Better) 26
3. A Simple Load Balancing Result Over 3 Different Hosts 27
4. A Better Load Balancing Result Over 3 Different Hosts 28
5. Total Turnaround Time vs. Number of Hosts 29
6. Computation Speedup Factor vs. Number of Hosts 30

VITA

Jian Liu

Candidate for the Degree of

Master of Science

Thesis: DESIGNING AND BUILDING AN EFFICIENT APPLICATION ON A UNIX NETWORK FILE SYSTEM

Major Field: Computer Science

Biographical:

Personal Data: Born in Beijing, P. R. China, March 6, 1966, the son of Maotang Liu and Shujun Yang

Education: Graduated from Beijing No. 35 High School, Beijing, P. R. China in 1984; received Bachelor of Medicine degree from Capital University of Medicine, Beijing, P. R. China, in July 1989. Completed the requirements for the Master of Science degree at Oklahoma State University in December, 1999.

Experience: Employed as a physician in charge at Beijing Goodwill Hospital, 1989-1996.

Professional Membership: China Medical Society.

CHAPTER I

INTRODUCTION

Networks have been called "decentralized" or "distributed" computing, and for good reason [Comer1997]. It provides many capabilities, and so much is involved. Most networking systems come with utilities for data transfer, remote program execution, and message passing. Under such circumstances, computer programming is no longer restricted to individual computers. Programmers are expected to design and implement application software that can communicate with software on other computers. Thus distributed computing is an emerging technology for creating software applications that can efficiently use the resources available in a network of computers.

It is the present trend for parallel or distributed computing and will continue for the foreseeable future. The practical issues involved in distributed computing are much the same as those of a group of people attempting to work together on a single project. It allows the people to share information and peripheral devices efficiently and economically. It holds many cost advantages, such as by using existing resources there are no additional hardware costs, the aggregate power and memory of existing resources at a site can easily exceed that of a supercomputer and so on. It allows users to run their applications for the same cost but at supercomputer speeds. Through this way, the computing power is distributed [Elbert1994].

Nowadays large workstation networks are widespread, providing an easy means for the sharing of computer resources. However, at any one time–and especially overnight–a significant proportion of workstations on a network may either be unused or be used for lightweight activities such as document editing. These workstations have spare processing capacity, and can be used to run jobs for users who are logged on at other workstations and do not have sufficient capacity at their machine. The problem of making better use of the resources provided by such networks is becoming more and more important from both the research and practical points of view.

The objective of this study is to design and build an application on a Unix Network File System (NFS) which could solve this problem and achieves a more efficient parallelization in a shorter time. The application has the key characters of distributed systems such as support for resource sharing, concurrency, scalability, fault tolerance, and so on. It could automatically detect which machine is idle, which machine is busy, dispatches the jobs among the machines, balances the load over the network and frees the user from the burden of looking for idle or under-utilized machines to use. It actually provides a facility for users to take advantage of the shared resources to achieve fast, reliable, and cost effective performance. It could be used as a front-end dispatcher to handle completely independent tasks. The application actually constitutes a general purpose parallelizing mechanism over a Unix Network File System.

This thesis is organized into the following chapters:

- Chapter II: Overview of distributed computing systems
- Chapter III: The design implementation issues of the application are discussed

- Chapter IV: The results obtained are presented and discussed
- Chapter V: Summary and future work are included

CHAPTER II

LITERATURE REVIEW

2.1. DISTRIBUTED COMPUTING SYSTEM

Distributed computing refers to the services provided by a distributed computing system [Umar1993]. A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software [Coulouris1994]. They are implemented on hardware platforms that vary in size. Distributed system software enables computers to coordinate their activities and to share the resources of the system-hardware, software and data. Users of a well-designed distributed system should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations.

The architecture of a distributed system identifies the main hardware and software components and modules of the system and defines the relationships between them. Important aspects of this are the types of computers used, their locations in the network and the locations at which system programs and application programs are executed.

There are three main architectural models for distributed systems that have emerged to date [Casavant 1994].

1. Workstation/server model: In this model each user is provided with a single-user computer, known as a workstation. Application programs are executed in the users' workstations. Users share the file servers and peripheral devices like printer. Sun NFS and Xerox DFS belong to this model.
2. Processor pool model: Programs are executed in a set of computers managed as a processor service. Users are provided with terminals rather than workstations, connected to the network via terminal concentrators and interacting with programs via a terminal access protocol like Cambridge DCS. Since workstations are now widely available and are included in most network systems, so a hybrid model has emerged. It is based on the workstation/server model, but with the addition of some pool computers that can be allocated dynamically for tasks that are too large for workstations or tasks that require several computers concurrently, like Amoeba system.
3. Integrated model: Each computer is provided with appropriate software to enable it to perform both the role of server and the role of application processor. The system software located in each computer is similar to an operating system for a centralized multi-user system, with the addition of networking software, like Locus system, Newcastle Connection and Unix system.

The key characteristics of distributed systems are support for resource sharing, openness, concurrency, scalability, fault tolerance and transparency. They are primarily responsible for the usefulness of distributed systems [Coulouris1994].

1. Resource sharing: The resources range extends from hardware components such as disks and printers to software-defined entities such as files, windows, databases and other data objects. Resources in a distributed system are physically encapsulated within one of the computers and can only be accessed from other computers by communication. Resource manager (a program) offers a communication interface enabling the resource to be accessed, manipulated and updated reliably and consistently. The resource users communicate with the resource managers to access the shared resources of the system and effective sharing each resource. Like the client-sever model and the object-based model.

2. Openness: It is the characteristic that determines whether the system can be extended in various ways. Like hardware extensions-the addition of peripherals, memory or software extensions-the addition of operating system features, communication protocols and resource-sharing services. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added without disruption to or duplication of existing services.

3. Concurrency: It means that when several processes exist in a single computer, then they are executed concurrently. In distributed systems there are many computers, each with one or more central processors. If there are M computers in a distributed system with one central processor each, then up to M processes can run in parallel, provided that the processes are located in different computers.

4. Scalability: Distributed systems operate effectively and efficiently at many different scales. The smallest practicable system could consists of two

workstations and a file server, whereas other systems constructed around a single local-area network may contain several hundred workstations and many file servers, print servers and other special-purpose servers. They enable resources to be shared between all of them.

5. Fault tolerance: When system faults occur in hardware or software, programs may produce incorrect results or they may stop before they have completed the intended computation. The design of fault-tolerant computer systems should have the ability to detect fault and recover them.

6. Transparency: The system is perceived as a whole rather than as a collection of independent components. The implications of transparency are a major influence on the design of the system software.

2.2. RELATIVE WORKS

The motivations for distributed systems became apparent in the early 1970s, a few years after the emergence of minicomputers. But the necessary hardware and software to make single-user computers fully effective, and the communication facilities to enable them to be used in a cooperative fashion, were absent. The earliest and some of the most significant developments aimed at filling those gaps were made by a team of researchers working at the Xerox Palo Alto Research Center (Xerox PARC) in the period of 1971-1980. These included the development of the first single-user workstations (Alto), file and print servers, the first high-speed local network (the Ethernet) and several experimental distributed systems. Like Xerox DFS, a very early file server distributed system was developed in 1977 [Mitchell and Dion 1982][Mitchell 1985].

Cambridge DCS (Cambridge Distributed Computing System) was the first distributed system based on the processor pool model. It was developed in the Computer Laboratory at Cambridge University, England [Needham & Herbert 1982] in 1979. There were no workstations in the original implementation of that system. Users were provided with simple terminals and were allocated processors from a heterogeneous processor pool.

Developed by Popek and his colleagues at the University of California, Los Angeles in 1980, Locus was one of the earliest distributed operating systems. It is an advanced distributed operating system that emulates UNIX [Popek & Walker 1985] and it supports transparent access to data through a network-wide file system. Known for such features as transparent process migration and a uniform distributed shared memory abstraction, Locus was extremely influential in the early development of parallel and cluster-style computing systems.

Another two distributed systems developed in 1980 were Apollo Domain by Apollo Computers [Leach et al.1983] and Newcastle Connection by Newcastle University [Brownbridge et al.1982]. Apollo Domain was the largest existing configuration commercial distributed system, which connected with 1800 workstations. Newcastle Connection was the precursor to the Sun Network File System.

From the start of the 1980s there was a rapid expansion of research and development in distributed systems.

Xerox PARC developed Grapevine system in 1981 [Birrell et al.1982] and Cedar system in 1982 [Teitelman 1984][Donahu 1985]. Grapevine system was a distributed replicated application-oriented database service. It is an early example of a distributed system designed to meet specific goals and to offer guarantees of the systems' behavior.

Which means it provides guarantees concerning the reliability of the service in the face of client program errors, hardware failures and network failures. Cedar system was a research environment for the development of office and personal systems.

The V system was made by Stanford University in 1982 [Cheriton 1984][Cheriton & Zwaenpoel 1985]. It is a testbed for research in distributed systems. The architecture of V system consists of four logical components: a distributed Unix kernel, service modules, runtime support libraries, and additional user commands. It manages the workstations (client) and the servers (back-end processors) transparently to give the user an impression of one large centralized system.

MIT developed Argus in 1983 [Liskov & Scheifler 1984]. It is an integrated programming language and system for implementing distributed programs.

In 1984, Amoeba was developed at the Vrije University in Amsterdam. It is a complete distributed operating system design, including all the basic facilities that one would expect from a conventional operating system [Tanenbaum et al. 1990][Tanenbaum 1992]. It is an example of a system that supports the processor pool model. Other designs that have incorporated processor pools include Plan9 [Pike et al. 1990] and the Clouds system [Dasgupta et al. 1991].

SUN Microsystems developed a distributed UNIX system (Sun Network File System) in 1985 [Coulouris 1994]. It is an extension of the UNIX operating system that provides a distributed file service based on networked UNIX systems. Now, it has been widely adopted in industry and in academic environments.

Carnegie-Mellon University developed Accent in 1982 [Rashid 1986] and Mach in 1986 [Boykin et al. 1993] [Loepere 1991]. Accent is a distributed operating system

based on message passing and it is the precursor of Mach, which is an operating system kernel for distributed systems which can run on both multiprocessor and uniprocessor computers connected by networks. Many of Mach's features are derived from Accent. The UNIX emulation enables Mach to provide an environment in which users may continue to use programs that use the UNIX system call interface.

Chorus was made by Chorus Systems company in 1988 [Rozier et al.1988][Rozier et al. 1990]. It is an object-oriented operating system for distributed computing and it is an operating system kernel to support distributed and real-time systems. It is architecturally similar to Mach in many ways such as emulate operating systems, notably UNIX.

The 1990s is the decade of distributed systems due to the excellent price/performance ratio offered by microprocessor-based systems and the steady improvements in networking technologies [Wu1999]. Distributed computing has become a key component of high-performance computing [Baker1996]. It has resulted in a major shift towards distributed systems and away from centralized and multi-user computers. This trend has been accelerated by the development of distributed system software, designed to support the development of distributed applications. Many companies and organizations as stated above, continued to enhance and upgrade their developed distributed system. Some of the systems received widespread industrial recognition such as Mach which is available on many machines, including IBM PCs and SUN workstations [Umar 1993] and Chorus, which has become one of the major vehicles for commercial UNIX development and for real-time computing products [Birman 1996].

There are also many new systems developed around the world during this period, such as Delta-4 system which was one of the first systematic efforts to address reliability

and fault-tolerant concerns. It was launched in Europe in 1991 [Powell 1991]. The HAS (Highly Available System) was developed by IBM's Almaden research laboratory in 1991 and subsequently contributed technology to a number of IBM products [Cristian 1994]. NavTech was developed in 1993 and it is aimed at wide area applications with real-time constraints, such as banking systems [Verissimo1995]. Phoenix was launched in 1995 [Malloth1995] and the emphasis of this system is on issues that occur when process group techniques are used to implement wide area transactional systems or database systems. Some other systems like Psync developed by the University of Arizona [Mishra et al 1991], Rampart by AT&T Bell Laboratories [Reiter 1996], Relaces by the University of Bologna [Babaoglu et al. 1995], RMP by the University of California, Berkeley [Callahan 1996], StormCast by the University of Tromso [Asplin 1995] and Transis by Hebrew University of Jerusalem [Dolev1996]. Each of these systems was innovative in some areas and borrowed ideas from prior systems in other areas [Singh 1999].

Some research and work have been done in the parallel and distributed computing system area at Oklahoma State University. Hsiao [Hsiao 1992] designed and implemented an interactive parallel program slicer for the distributed-memory parallel programs in 1992. Alabdulkareem [Alabdulkareem 1993] designed and implemented a development tool for parallel algorithms by using the Linda approach. It was implemented on IBM PC with Microsoft Windows and it was one of the first models, which was developed on PC environment. In 1994, Duan presented a new scalable parallel processing architecture called TR-machine, which is based on conventional machine model and graph reduction. It was implemented by using object-oriented

technique [Duan 1994]. Prohmbhadra designed and implemented a distributed license server by using the concept of k-coteries and a distributed k-mutual exclusion algorithm to control access to the licenses in 1995 [Prohmbhadra 1995]. Valliere developed a hypertext system to process and visualize network management data in 1996 [Valliere 1996]. Burge developed a Java-based mobile actor system for heterogeneous distributed parallel computing in 1998 [Burge 1998].

CHAPTER III

DESIGN AND IMPLEMENTATION

3.1 IMPLEMENTATION PLATFORM

UNIX [Ritchie1974] is perhaps the best-known example of a multi-user operating system. Since it was already in widespread use and easily available when distributed systems were being developed, many distributed system researchers and developers adopted the model provided by the original UNIX as their goal [Brown1994].

The developments began with the extension of the original UNIX system design to include support for interprocess communication. It was achieved in the 4BSD versions of UNIX developed at the University of California at Berkeley in the late 1970s [Stevens1990]. Subsequently, these communication facilities were extended and exploited as a basis and the necessary components for the development of full-distributed systems.

The distributed UNIX model has been implemented in several forms. The most widely used implementation is that developed by Sun Microsystems, a workstation manufacturer that took the Berkeley BSD UNIX as the starting point for an extensive software effort, which led to the development of the well known Network File System (NFS). This software component and its associated Remote Procedure Calling (RPC) and Network Information Service (NIS) components are now offered by almost every

workstation vendor and are used as the basis for most current distributed UNIX implementations [Leffler1989].

Using the Network File System, it is possible to work locally with files that are stored on a remote computer system's disks, as if they were present on your own computer. The remote system acts as the file server; the local system is the client, making requests. NFS does not communicate with the kernel in the way that Remote File Sharing (RFS) does. Instead, it relies on Remote Procedure Calls (RPC), which allows machines to access services on a remote machine via a network. RPC handles remote requests and then hands them over to the operating system on the local machine. The local system has daemons running that attempt to process the remote request. These daemons issue the system calls needed to do the operations.

Remote Procedure Calls (RPC) were first proposed in White [1976], studied in detail by Nelson [1981], and efficiently implemented by Birrell and Nelson [1984] [Renaud1996]. Sun Microsystems developed the Sun RPC protocol in the mid-1980s. As with any RPC protocol, its objective was allow development of client/server applications without having to program at the session or transport (socket) level. The popularity of Sun Microsystems' public domain RPC protocol has led to wide-spread use of RPC as a basis for client/server programming. Today, RPC are the major communications mechanism that underpins most client/server communications [Kochan1989]. For example, most client/server middleware, such as message queuing, mail-enabled messaging, TP monitors, network SQL, and distributed object protocols are built on top of RPC.

The application was tested using four Sun workstations in Oklahoma statwide NMR Facility. One machine was served as the master and distributed jobs to the other three machines by using the RPC. File sharing between Sun workstations was implemented by using Unix Network File System protocol.

3.2. DESIGN FEATURES OF THE APPLICATION

1. The interface of this application is transparent to the user. The user just simply need to specify the input file which contains the names of the jobs and the rest is automatically taken care of by the application. To the user, it is as though all of the computations were done on a single machine. It is assumed that all jobs take the same time interval and all jobs must be available initially.
2. The application makes the best use of the available machines on the network by using the RPC (Remote Procedure Calls) and system daemons to monitor each computer; locating and allocating "idle" machines; and balancing the loads on the network.
3. This application assures the "owners" of the machines have the highest priority over their own workstations. It will run the jobs in the background at a low priority (typically a nice value of 10), so that the operating system will preferentially allocate CPU time to the owner's jobs. If the monitoring status indicates that a particular machine is "too busy", the application will remove the background job from that machine and redistribute it to another machine.

4. The application is tolerant to many types of faults like premature terminations of the computation due to machine crash. In such case, the uncompleted job is redistributed to another machine.

5. The application can provide a means (through a local host table in the current working directory) for users to select which hosts to use instead of, by default, all the hosts on the local network. This allows some machines to be excluded for various reasons: critical file servers, machines which are too slow to be useful, machines whose owners don't like their machines to be shared, etc.

6. The application supports a set of optional parameters to provide users with some flexibility. For example, the user can define what it means for a host to be "idle" by setting the IDLE-LOAD parameter; likewise, BUSY-LOAD is set to define "too busy"-when a running job has to be removed. These parameters are defined in terms of the 1-minute load-average on the machine. Other parameters include the PRIORITY (nice value) at which the job runs in the background and so on.

7. The application can provide some facilities to its users: a progress report while it is running, a final summary of all critical statistics (like the total turnaround time, each host's performance data, etc.), and a log file that records what has happened for each job. The final jobs' result will be collected by the application and stored at a file which name is P-log-d.

3.3 THE ARCHITECTURE OF THE APPLICATION

The architecture of the application is depicted in Figure 1. The functionality of each box is briefly summarized as follows:

1. QUEUING SYSTEM: It consists of three queues of same type: wait-q, run-q, and done-q. Each one of them is a linked list of the jobs. wait-q holds all the jobs that are waiting to be dispatched; run-q lists all the jobs that are currently running on some hosts; and done-q has the completed jobs. Initially, all the jobs are placed in wait-q, when the system running, these jobs migrate among the queues.
2. CONTROLLER: This is the heart of application's operation. Its responsibilities include updating the status of hosts, monitoring the progress of running jobs, balancing the load over the network, and detecting any abnormal conditions such as machine crashes and premature terminations of processes, and so on. In particular, the status of each host is checked and marked as either down, up and idle, or up and non-idle so that Executor will know whether or not to use a particular host. The progress of each running job is periodically monitored so that if it is done, it will be moved to done-q, thus freeing the host for some other jobs. And if any resubmit condition is satisfied (like the load average on a machine exceeds the user-defined busy load threshold (via BUSY-LOAD); the job abnormally terminates before completion; and so forth), the job is removed from the host and placed on wait-q again. When a job is aborted for one of these reasons, all the calculations performed during the process are lost, and upon resubmission the computation is reinitiated from the beginning.
3. EXECUTOR: According to the host status report generated by Controller, Executor dispatches jobs on wait-q to available idle hosts, which means a dispatched job is moved from wait-q to run-q.

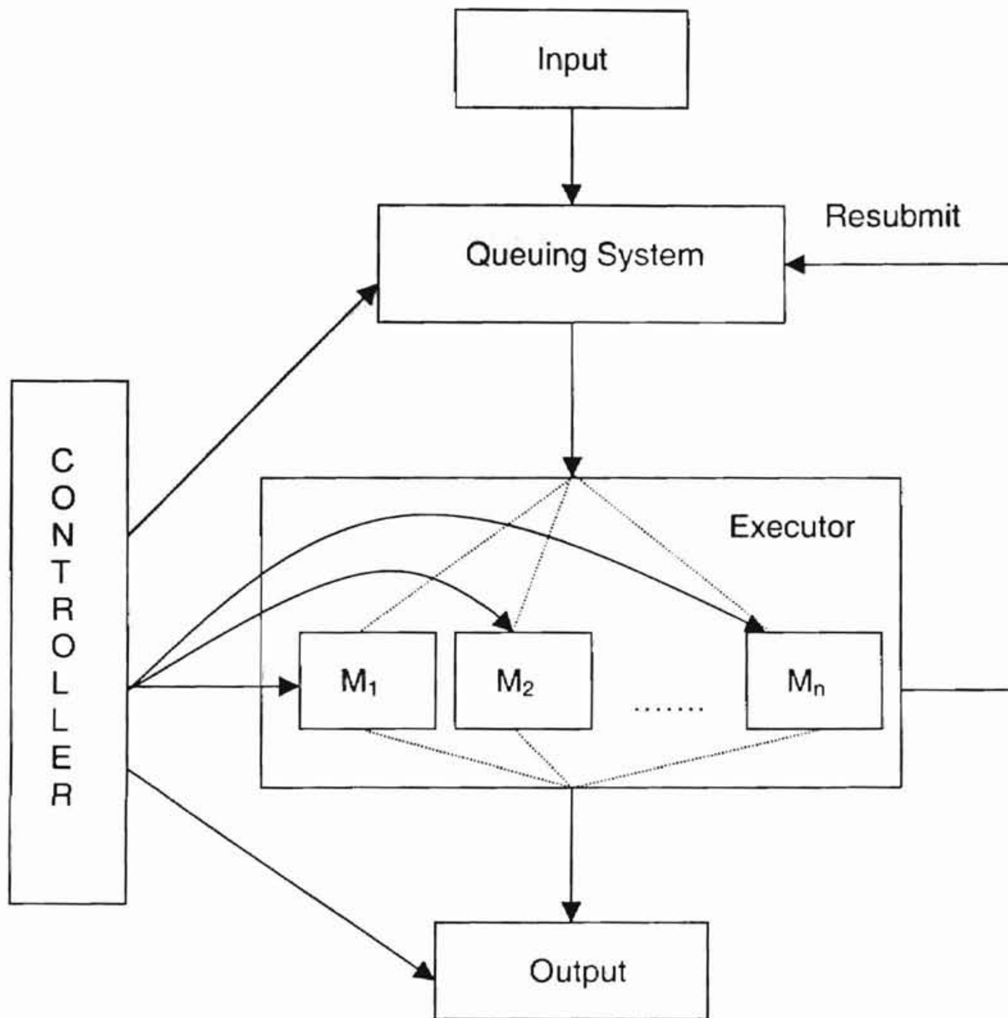

Figure 1. The Architecture of the Application

3.4. LOAD BALANCING ALGORITHMS

One of the greatest challenges in implementing network parallelization is to effectively perform load balancing and resource allocation. The problem is essentially different from that of job scheduling on an internally parallel machine due both to the natural inhomogeneity of the network metacomputer, as well as to the random and unknown fluctuations in individual machine availability and performance. This problem would appear to be a fertile area for future research. One possibility might be to allow for dynamically varying the batch time size for the jobs. Another approach would involve waiting for faster machines to perform the last few jobs which to be described later.

1. A SIMPLE ALGORITHM

Even if we do not use any load balancing algorithm, the way that Controller and Executor were implemented implies that whenever a host is available and there are some waiting jobs on wait-q, that host will be used to run one job from the waiting queue. Clearly, if the number of total jobs is many times the number of hosts, the final completion times on the hosts are expected to be very close, thus balancing the load among the hosts almost for free. On the other hand, if the batch size of each job is too small (the amount of execution time is small), then the overhead will increase as a fraction of the total running time. The overhead primarily involves the time waiting for the load average on the machines to drop below the IDLE-LOAD value.

2. A BETTER ALGORITHM

Although the load is intrinsically fairly well balanced among the hosts at almost no cost by using above simple algorithm, the application did implement a better improvement to the load balancing algorithm: when each job is initiated, the Executor submits it to the "fastest" one among all available hosts. The speed of each machine is dynamically determined from their past performance on the completed jobs. Therefore, both the machine's intrinsic speed and its availability for running the job are taken into account. The machines' speed was computed based on assuming that each job has the equal batch time size (although it is often unrealistic and not a common case). The jobs were compute the longest common subsequence problem.

Support that there are two machines M_1 and M_2, and that M_1 is r times faster than M_2. Let T_1 and T_2 denote the average run time for M_1 and M_2 for jobs of the same length (batch size). Then $T_2/T_1 = r$. As far as the load balancing between the two is concerned, the worst case happens when M_2 takes the last job because M_1 is not yet available (let us say that M_1 is still εT_1 minutes from completion). In this case, it would take an additional T_2 minutes for the job to complete. Alternatively, if we wait εT_1 minutes until M_1 finishes its current job and then let it take the last job, it would take $(1+\varepsilon)\, T_1$ minutes for the job to complete. Clearly, the second alternative is preferred when

$$(1 + \varepsilon)\, T_1 < T_2 \qquad (1)$$

or,

$$1 + \varepsilon < r, \text{ where, } 0 < \varepsilon \leq 1 \qquad (2)$$

Thus, we can see that if $r \geq 2$, then the latter alternative will always produce a shorter total turnaround time.

For another perspective, M_2 should never be used if the number of waiting jobs (denoted by n) is such that

$$n \leq \lfloor r - \varepsilon \rfloor \quad (3)$$

For example, let $r = 1.5$, and $\varepsilon = 0.4$. Then if $n = 1$, we should wait for M_1 to finish in order to take the last job, instead of using M_2. As another example, suppose $r = 4$, and $\varepsilon = 0.9$. Then the stop point for M_2 is when $n = 3$.

Clearly, this algorithm can be easily extended to the case in which there are more than 2 hosts. First, one would order the hosts according to their speeds with the fastest host being the first in the list. Second, before starting off a job on M_i, one would compute the stop point value against all M_j ($j = 1,2,\ldots, i-1$) using Eqn. (3). If the sum of all these values is greater than the number of waiting jobs, neither M_i nor any of the hosts that follow M_i in the order list would be used.

3. EVERY MACHINE'S WAITING TIME AT THE END IS AT MOST ITS AVERAGE EXECUTION TIME OF ONE JOB

PROOF: According to the better algorithm, the necessary condition to not use M_i is that if it were used, the total turnaround time would be longer. That is, the actual total turnaround time must be less than that when M_i would be used. Therefore, M_i waits at most T_i (its average execution time per job) for all other machines to finish their last jobs.

CHAPTER IV

RESULTS AND ANALYSIS

Even though the worst case with or without the better algorithm is the same, in practice the better algorithm gives better load balancing among the hosts.

4.1. THE ALGORITHM ACHIEVES BETTER LOAD BALANCING

PROOF: As every machine's waiting time at the end is at most its average execution time of one job, if any machine steals one more job from another machine, the total turnaround time would be longer.

1. The improvement on the total turnaround time brought by the better algorithm is about 10% in general. As an example, Figure 3 is the improved result of the same task from Figure 2, reducing the total turnaround time from 2.60 minutes to 2.33 minutes. Its corresponding graphical representation on the load balancing effect is depicted in Figure 4 and Figure 5. The machine real time speeds vary from 0.18 minutes to 0.54 minutes for one job. The shorter white intervals represent the average delays between the completion of the last job and the start of the next one, part of which is application's own overhead and the rest is because the host was considered "non-idle" (usually because someone else was using the machine). The large shaded intervals represent the average computational times for each job. It can be seen that

each host takes essentially as many jobs as it can and that there is relatively little dispersion of the final completion times.

2. From many past runs, before the better algorithm was implemented, it was almost always true that a slower machine finished last just like figure 4 displayed.
3. The average run time per job T_i is dynamically determined, thus accounting for the machine's intrinsic speed and its dynamic usage.
4. In the real implementation, the delays (overheads) before two successive jobs were taken into consideration, thus resulting in a slightly different algorithm from what was described above.
5. In reality, because of the unpredictable nature of machines' usage, the load balancing performance by the algorithm may be slightly off from the optimum.
6. This algorithm gives a much better balancing effect when the speeds of the machines very greatly than when they are close (compared with the simple algorithm above).

4.2. TEST RESULTS

1. ACCURACY: A necessary condition for this application to be considered useful and successful is that its final output must be equal to that produced by a sequential calculation on a single machine. I have confirmed through explicit test that this application's output is correct.
2. SPEEDUP TEST: Test results for speedup are presented in Figure 6 and Figure 7. The same number of jobs was run using 1, 2, 3 machines. To ensure a fair comparison, I run the experiment during a period when all the machines were

virtually idle (mid-night). It can be seen that with the better algorithm, the total turnaround time is shorter and the speedup factor is greater than without it.

```
-----------------------------------------------
File: P-log-h
   (automatically generated log file)
Host running Program:  nmrserv
-----------------------------------------------

------ Parameter setting ------
IDLE_LOAD = 2.00
BUSY_LOAD = 4.00
CPU_PERCENT = 25
PRIORITY = 10
DEBUG = 1
LOG_FILE_PREFIX = P
------ End of Parameter setting ------


---------- Job statistics ----------

Total time: 2.60 minutes.
      From: Wed Nov  3 21:05:32 1999
        To: Wed Nov  3 21:08:08 1999

   Total hosts:  3
    Used hosts:  3
Unused hosts:  0

----- Itemized statistics for each host --------
-------------------------------------------------------------------------------------
   Host    Done  Abort  Avg.    Last done          Lag      Delay  D/D+A
-------------------------------------------------------------------------------------
   i400    8      0      0.18m   Nov  3 21:07:54  0.12m  0.13m  42.3%
   i600    8      0      0.19m   Nov  3 21:07:54  0.12m  0.12m  38.3%
   g300    4      0      0.54m   Nov  3 21:08:01  0.00m  0.11m  16.7%
-------------------------------------------------------------------------------------
   Total:  20     0      0.26m                     0.08m  0.12m  32.2%
```

Figure 2. A Sample Summary Log File (Simple Load Balancing Result)

File: P-log-h
(automatically generated log file)
Host running Program: nmrserv

------ Parameter setting ------
IDLE_LOAD = 2.00
BUSY_LOAD = 4.00
CPU_PERCENT = 25
PRIORITY = 10
DEBUG = 1
LOG_FILE_PREFIX = P
------ End of Parameter setting ------

---------- Job statistics ----------

Total time: 2.33 minutes.
From: Wed Nov 3 21:00:26 1999
To: Wed Nov 3 21:02:46 1999

Total hosts: 3
Used hosts: 3
Unused hosts: 0

----- Itemized statistics for each host --------

Host	Done	Abort	Avg.	Last done	Lag	Delay	D/D+A
i400	8	0	0.18m	Nov 3 21:02:30	0.15m	0.09m	32.8%
i600	9	0	0.19m	Nov 3 21:02:39	0.00m	0.07m	28.6%
g300	3	0	0.55m	Nov 3 21:02:09	0.50m	0.06m	10.0%
Total:	20	0	0.24m		0.22m	0.08m	24.7%

Figure 3. A Sample Summary Log File (Better Load Balancing Result)

File: P-log-h
(automatically generated log file)
Host running Program: nmrserv

------ Parameter setting ------
IDLE_LOAD = 2.00
BUSY_LOAD = 4.00
CPU_PERCENT = 25
PRIORITY = 10
DEBUG = 1
LOG_FILE_PREFIX = P
------ End of Parameter setting ------

---------- Job statistics ----------

Total time: 2.33 minutes.
From: Wed Nov 3 21:00:26 1999
To: Wed Nov 3 21:02:46 1999

Total hosts: 3
Used hosts: 3
Unused hosts: 0

----- Itemized statistics for each host --------

Host	Done	Abort	Avg.	Last done	Lag	Delay	D/D+A
i400	8	0	0.18m	Nov 3 21:02:30	0.15m	0.09m	32.8%
i600	9	0	0.19m	Nov 3 21:02:39	0.00m	0.07m	28.6%
g300	3	0	0.55m	Nov 3 21:02:09	0.50m	0.06m	10.0%
Total:	20	0	0.24m		0.22m	0.08m	24.7%

Figure 3. A Sample Summary Log File (Better Load Balancing Result)

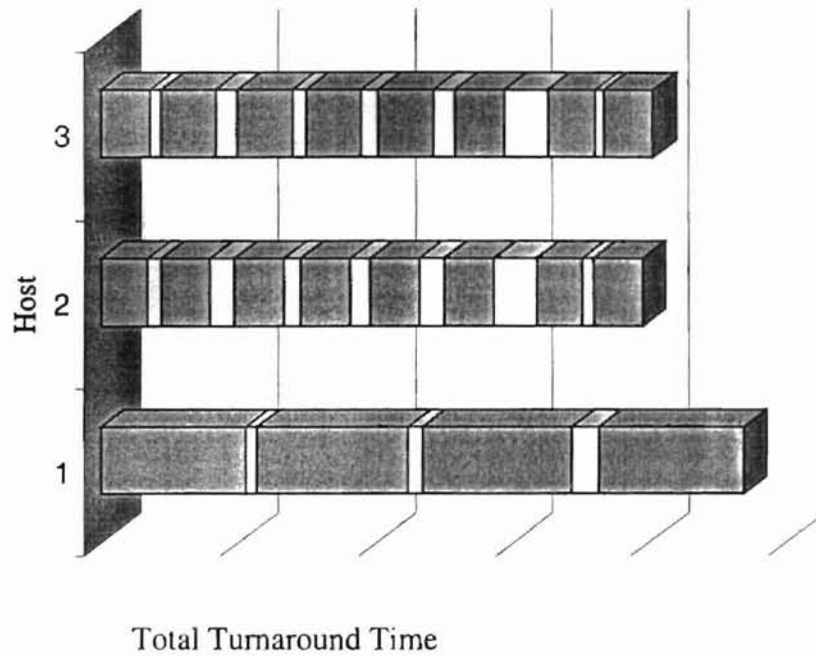

Figure 4. A Simple Load Balancing Result Over 3 Different Hosts

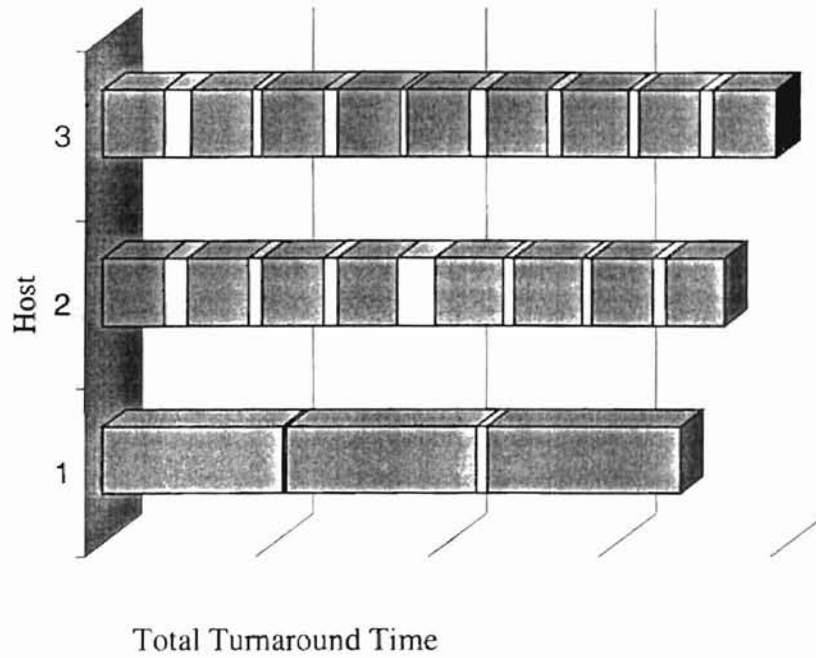

Figure 5. A Better Load Balancing Result Over 3 Different Hosts

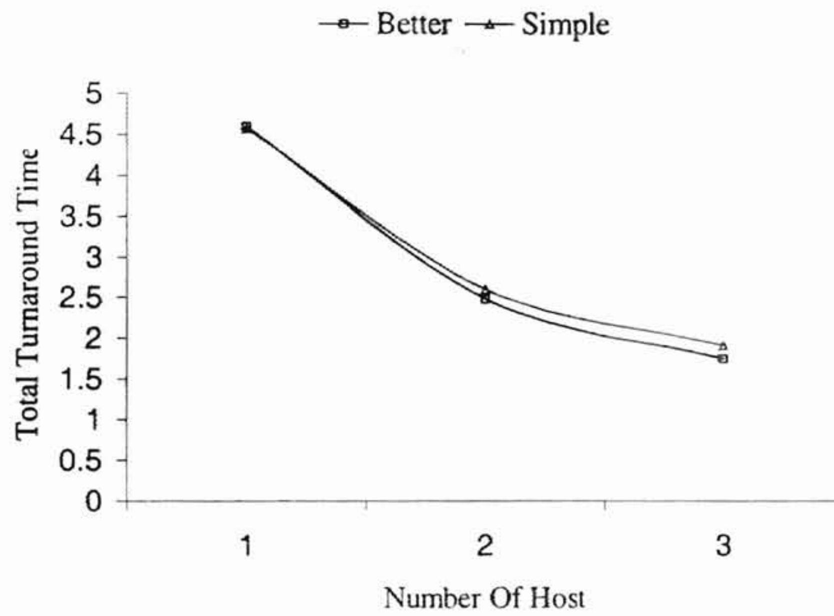

Figure 6. Total Turnaround Time vs. Number Of Host

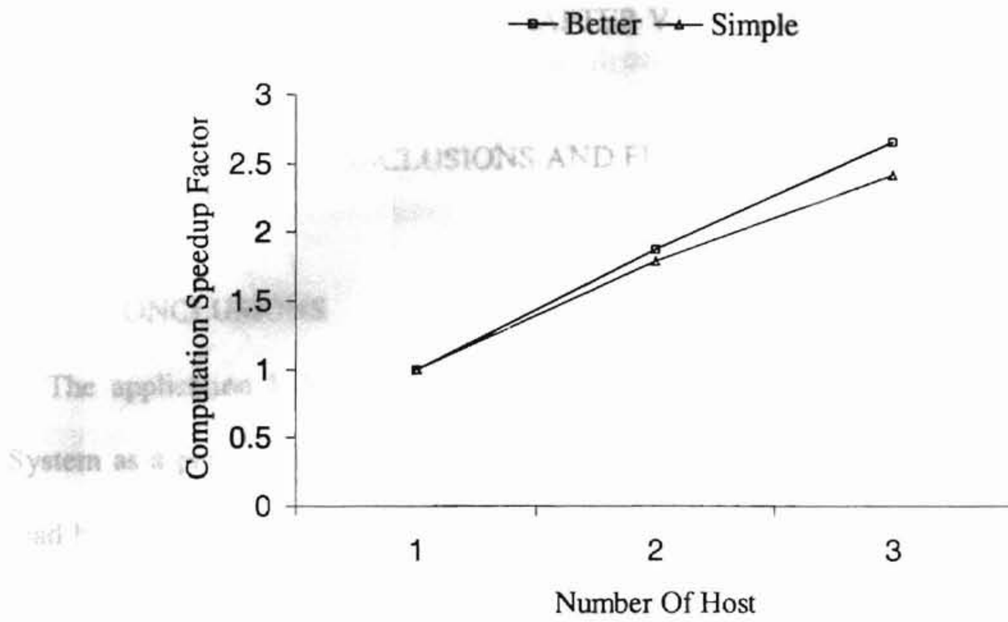

Figure 7. Computation Speedup Factor vs. Number Of Host

CHAPTER V

CONCLUSIONS AND FUTURE WORK

5.1. CONCLUSIONS

The application I developed enables us to use a UNIX workstation Network File System as a parallel metacomputer for computation. This application does an efficient load balancing among hosts over the Network File System. It has the ability to recover from many types of faults, requires no manual intervention from the user, and provides a lot of flexibility to the user. And the real beauty of the application is that adding more workstations to boost the computational power is easy: simply add new host names to the host file. Therefore, as more and powerful stations are purchased and installed, the application will get more power automatically.

5.2. FUTURE WORK

This application can be enhanced along the following several directions.

1. Migrating jobs instead of aborting them when one or more of the resubmit conditions are met.
2. Upgrading it to different architectures and operating systems.

3. Design and construct a divide-and-merge algorithm, then combine it with this application in order to build a more powerful machine which can run parallel programs in a parallel distributed system.
4. Design and implement an optimal algorithm which could distribute the jobs according to the vary batch time size of jobs and the different machine speed to achieves an optimal loading balance.

REFERENCE

[Alabdulkareem 1993] Alabdulkareem, M. A., "A development environment for parallel algorithms based on linda", 1993, Oklahoma State University, MS thesis.

[Asplin 1995] Asplin, J., "Performance Experiments with the Storm View Distributed Parallel Volume Renderer", 1995, *Computer Science Technical Report 95-22. University of Tromso.*

[Babaoglu et al. 1995] Babaoglu, O., "Failure Detectors, Group Membership, and View-Synchronous Communication in Partitionable Asynchronous Systems", 1995, *Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna.*

[Baker1996] Baker, Lou and Smith, J. Bradley, *Parallel Programming*, 1996, McGraw-Hill Companies, Inc.

[Birman 1996] Birman, K. P., *Building Secure and Reliable Network Applications*, 1996, Manning Publications Co. CT.

[Birrell et al.1982] Birrell, A. D., Levin, R., Needham, R. M. and Schroeder, M. D., "Grapevine: an exercise in distributed computing", 1982, *Comm. ACM, Vol.25, No.4, pp.260-73.*

[Boykin et al. 1993] Boykin, J., Kirschen, D., Langerman, A. and LoVerso, S., *Programming under Mach*, 1993, Reading MA: Addison-Wesley.

[Brown1994] Brown, Chris. *Unix—Distributed Programming*, 1994, Prentice Hall International Limited.

[Brownbridge et al.1982] Brownbridge, D. R., Marshall, L. F. and Randell, B. "The Newcastle connection, or UNIXes of the world unite!", 1982, *Software Practice and Experience, Vol.12, pp.1147-62.*

[Burge 1998] Burge, Legand L., "JMAS: a Java-based mobile actor system for heterogeneous distributed parallel computing", 1998, Oklahoma State University, Ph.D thesis.

[Callahan 1996] Callahan, J., "Approaches to Verification and Validation of a Reliable Multicast Protocol", *Proceedings of the 1996 ACM International*

Symposium on Software Testing and Analysis, 187-194.

[Casavant 1994] Casavant, Thomas L., *Distributed Computing Systems*, 1994, IEEE Computer Society Press, Los Alamitos, California.

[Cheriton 1984] Cheriton, D. R., "The V kernel: a software base for distributed systems", 1984, *IEEE Software, Vol.1 No.2, pp.19-42.*

[Cheriton & Zwaenpoel 1985] Cheriton, D. R. and Zwaenepoel, W., "Distributed process groups in the V kernel", 1985, *ACM Trans. USENIX Summer Conference, pp.73-85.*

[Comer1997] Comer, Douglas E. *Computer Networks and Internets*, 1997 Prentice-Hall, Inc. Upper Saddle River, NJ 07458

[Coulouris1994] Coulouris, George and Dollimore, Jean and Kindberg, Tim. *Distributed Systems-Concepts and Design*, 1994 Addison-Wesley Publishing Company Inc.

[Cristian 1994] Cristian, F., "Fault-Tolerant Internal Clock Synchronization", 1994, *Proceedings of the Thirteen Symposium on Reliable Distributed System.*

[Dasgupta et al. 1991] Dasgupta, P., LeBlanc Jr., R. J. Ahamad, M. and Ramachandran, U., "The Clouds Distributed Operating System", 1991, *IEEE Computer, Vol.24, No.11, pp.34-44.*

[Dolev1996] Dolev, D., "The Transis Approach to High Availability Cluster Communication", 1996, *Communications of the ACM 39:4:64-70.*

[Donahu 1985] Donahu, J., "Integration mechanisms in Cedar", 1985, *Proc ACM SIGPLAN 85 Sym. On Programming Languages and Environments.*

[Duan 1994] Duan, J., "An object-oriented parallel simulation of TR-machine architecture", 1994, Oklahoma State University, MS thesis.

[Elbert1994] Elbert, Bruce and Martyna, Bobby. *Client/Server Computing -Architecture, Applications, and Distributed Systems Management*, 1994 Artech House, Inc. 685 Canton Street, Norwood, MA 02062.

[Hsiao 1992] Hsiao, T., "An interactive parallel program slicer for C programs on the ipsc/2 system", 1992, Oklahoma State University, MS thesis.

[Kochan1989] Kochan, Stephen G. and Wood, Patrick H., *Unix Networking*, 1989, Pipeline Associates, Inc.

[Leach et al.1983] Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L. and Stumpf, B. L. "The architecture of an integrated local network", *IEEE J.*

Selected Areas in Communications, Vol. SAC-1, No.5, pp.842-56.

[Leffler1989] Leffler, Samuel J. and Mckusick, Marshall Kirk et al, *The Design and Implementation of the 4.3BSD Unix Operating System*, 1989, Addison-Wesley Publishing Company.

[Liskov & Scheifler 1984] Liskov, B. and Scheifler, R. W., "Guardians and actions: linguistic support for robust, distributed programs", 1982, *ACM Trans. Programming Languages and Systems, Vol.5, No.3, pp.381-404.*

[Loepere 1991] Loepere, K., "Mach 3 Kernel Principles", 1991, *Open Software Foundation and Carnegie-Mellon University.*

[Malloth1995] Malloth, C. P., "Phoenix: A Toolkit for Building Fault-Tolerant Distributed Applications in Large-Scale Networks.", 1995, *New York: IEEE Computer Society Press.*

[Mishra et al 1991] Mishra, S., "A Membership Protocol Based on Partial Order.", 1991, *Proceedings of the IEEE International Working Conference on Dependable Computing for Critical Applications, 137-145.*

[Mitchell1985] Mitchell, J. G., "File servers In Local Area Networks: an Advanced Course, Lecture Notes in Computer Science", 1985, *Springer-Verlag, pp.221-59.*

[Mitchell & Dion 1982] Mitchell, J. G. and Dion, J. "A comparison of two network-based file servers", 1982, *Comm. ACM, Vol.25, No.4, pp.233-45.*

[Needham & Herbert 1982] Needham, R. M. and Herbert, A.J., *The Cambridge Distributed Computing System*, 1982, Wokingham: Addison-Wesley.

[Pike et al. 1990] Pike, R., Presotto, K., Thompson, K. and Trickey, H., "Plan 9 from Bell Labs", 1990, *Proc. UK Unix Users Group, Summer 1990, Conference, London.*

[Popek & Walker 1985] Popek, G. and Walker, B., *The LOCUS Distributed System Architecture*, 1985, Cambridge MA: MIT Press.

[Powell 1991] Powell, D., "Delta-4: A Generic Architecture for Dependable Distributed Computing", 1991, *Springer-Verlag ESPRIT Research Reports, Vol. 1, Project 818/2252.*

[Prohmbhadra 1995] Prohmbhadra, S., "Distributed license server", 1995, Oklahoma State University, MS thesis.

[Rashid 1986] Rashid, R. R., "From RIG to Accent to Mach: the evolution of a network operating system", 1986, *Proceedings of the ACM/IEEE Computer Society Fall Joint Conference, ACM, November.*

[Reiter 1996] Reiter, M. K., "Distributing Trust with the Rampart Toolkit", 1996,

Communications of the ACM 39:4, 71-75.

[Renaud1996] Renaud, Paul E., *Introduction to Client/Server Systems-A Practical Guide for Systems Professionals*, Second Edition, 1996, by John Wiley & Sons, Inc.

[Ritchie1974] Ritchie, D. M. and Thompson, K., " The UNIX time-sharing system", 1974, *Comms. ACM, Vol.17, No.7, pp.365-75.*

[Rozier et al.1988] Rozier, M., Armand, F. and Neuhauser, W., "Chorus Distributed Operating Systems", 1988, *Computing Systems Journal, Vol.1, No.4, pp.305-70.*

[Rozier et al.1990] Rozier, M., Abrossimov, V., and Neuhauser, W., "Overview of the Chorus Distributed Operating System", 1990, *Tech. Report CS/TR-90-25.1, Chorus Systems, France.*

[Singh1999] Singh, Harry, *Progressing to Distributed Multiprocessing*, 1999, Prentice Hall PTR, Upper Saddle River, NJ 07458.

[Stevens1990] Stevens, W. Richard, *UNIX Network Programming*, 1990, Prentice-Hall, Englewood Cliffs, NJ.

[Tanenbaum 1992] Tanenbaum, A. S., *Modern Operating Systems*, 1992, Prentice-Hall, Englewood Cliffs, NJ.

[Tanenbaum et al. 1990] Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G., Mullender, S., Jansen, J. and van Rossum, G., "Experiences with the Amoeba Distributed Operating System", 1990, *Comm. ACM, Vol.33, No.12, pp.46-63.*

[Teitelman 1984] Teitelman, W., "A tour through Cedar", 1984, *IEEE Software, Vol. 1, No. 2, pp.44-73.*

[Umar 1993] Umar, Amjad, *Distributed Computing – A Practical Synthesis*, 1993, PTR Prentice-Hall, NJ 07632.

[Valliere 1996] Valliere, C. E., "Distributed reporting management system", 1996, Oklahoma State University, MS thesis.

[Verissimo1995] Verissimo, P., *Real-Time Communication*, 1995, Distributed Systems, 2d ed., Addison-Wesley/ACM Press.

[Wu1999] Wu, Jie, *Distributed System Design*, 1999, CRC Press LLC, Florida.

APPENDIX

```
/*******************************************************************
* Thesis Project:
* Designing and Building an Efficient Application On A Unix Network File System
* Author: Jian Liu
* Date: December, 1999      (Fall, 1999)
*
* The purpose of this project is to design and build an application, which provides a
* facility for users to run their jobs at idle or under-utilized workstations on a Unix
* Network File System. It can automatically take advantage of the shared resources to
* achieve fast, reliable, and cost effective performance.
* Users just need to load this application on their own machine and run this program.
* There should have 3 files come with this program. One is the hosts file. It contains the
* name of the hosts which will run the program. One file contains the list of the
* program's name on the user's machine and will be distributed to the other host's
* machine to run. One file contains the parameters which user set for the application to
* justify and to distribute the jobs.
* When the users run the program, they should input the file which contains the list of
* programs' name and the output file will be generated automatically.(P-log-d)
*
*******************************************************************/

#define NAME_LEN          80
#define LINE_LEN          160
#define SYS_CALL_LEN      250

#define MODE_MASK         0777   /* octal */
#define DONE_MODE         0777   /* octal */

#define WAITING           0
#define RUNNING           1
#define DONE              2
#define ABORTED           -1

#define P_OFF             0
#define P_ON              1
#define DOWN              -1

#define FILE_FAIL_LIMIT          4       /* have to change later !!!! */
#define HOST_FAIL_LIMIT          4       /* have to change later !!!! */
```

```
struct host {                          /* structure for host */
  char name[NAME_LEN];                 /* host name */
  long up;                             /* uptime in terms of minutes */
  int users;                  /* number of users on host */
  float load_1;               /* load average in last 1 minute */
  float load_5;               /* load average in last 5 minutes */
  float load_15;              /* load average in last 15 minutes */
  int status;                 /* -1: DOWN; 1: MCNP running; 0: MCNP off */
  long last_check_time;       /* time of last checking */
  int fails;                  /* continuous abortion or unsuccess of start */
  struct jobs {               /* statistics about jobs run on the host */
    int done;                 /* no of jobs done */
    int abort;                /* no of jobs aborted */
    long sum_time;            /* sum of time (seconds) for jobs done */
    long last_time;           /* time of last job done */
  } jobs;
};
typedef struct host host;

struct record {                        /* record for each MCNP job */
  char host_name[NAME_LEN];            /* host name */
  int host_id;                         /* host id: index to host table */
  struct rpss {                        /* remote processes' information */
    int no_ps;                         /* number of processes */
    int pid;                           /* the process id */
    int cpu;                           /* cpu time used by the process so far */
  } rpss;
  int start_hist;                      /* starting history number */
  struct f_names {
    char in[NAME_LEN];                 /* input file name */
    char out[NAME_LEN];                /* output file name */
  } f_names;
  int f_fail_cnts;                     /* output file inquery failures */
  long start_time;                     /* start time, seconds since Jan. 1, 1970 */
  long end_time;                       /* end time, seconds since ... */
  int status;                          /* -1: aborted;  0: waiting;  1: running; 2: done; */
  struct record * next;                /* pointer to the next record */
};
typedef struct record record;

float IDLE_LOAD;
float BUSY_LOAD;
int CPU_PERCENT;
int PRIORITY;
int DEBUG;
```

```
char LOG_FILE_PREFIX[NAME_LEN];        /* log files' prefix */



#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <signal.h>
#include <setjmp.h>
#include "global.h"

jmp_buf env_buf;

main(int argc, char *argv[])
{
  int no_hosts, *h_order;       /* order of hosts */
  host *hosts;                  /* a list of hosts */
  record *wait_q, *run_q = NULL, *done_q = NULL;
  long init;

  if (argc != 2){
    printf("-------- Usage: %s & Input -------- \n", argv[0]);
    exit(1);
  }
  init_user_parms();    /* initial the user's paramaters */
  hosts = init_host_table(&no_hosts);  /* initial host table */
  h_order = init_host_order(no_hosts);  /* initial host order */
  log_file_init("h", "parms");
  wait_q = construct(argv[1]);    /* build up a waiting queue */
  init = get_clock();
  log_file_init("r", "records");
  run_q = get_a_record("dummy", RUNNING);
  done_q = get_a_record("dummy", DONE);
  parallel_run(wait_q,run_q,done_q,init,no_hosts,hosts,h_order);
  log_file_hosts(no_hosts, hosts, h_order, init);
  print_q(stdout, done_q, "DONE Queue");
  log_file_init("d", "records");
  log_file_d(done_q->next);
}

/* file for the statistics of hosts */
void log_file_hosts(no_hosts, hosts, h_order, init)
int no_hosts, * h_order;
host * hosts;
```

```
long init;
{
  FILE * fp;
  host * h;
  long curtime, total_time, lag, latest = 0;
  float avg, delay, percent;
  char log_f[NAME_LEN];
  int i, used_hosts = 0, dones = 0, aborts = 0;
  long total_lag = 0;
  float total_avg = 0, total_delay = 0;
  float total_avg2 = 0, total_delay2 = 0, total_percent2 = 0;

  strcpy(log_f, LOG_FILE_PREFIX);
  strcat(log_f, "-log-h");
  fp = fopen(log_f, "a");

  curtime = get_clock();
  total_time = curtime - init;

  fprintf(fp, "\n\n---------- Job statistics ----------\n\n");
  fprintf(fp, "Total time: %.2f minutes.\n", total_time/60.0);
  fprintf(fp, "    From: %s\n", cctime(init));
  fprintf(fp, "      To: %s\n", cctime(curtime));
  fprintf(fp, "\n");
  fprintf(fp, " Total hosts:  %d\n", no_hosts);
  for (i = 0; i < no_hosts; i++) {
    if ((hosts+i)->jobs.done != 0)
      used_hosts++;
    if ((hosts+i)->jobs.last_time > latest)
      latest = (hosts+i)->jobs.last_time;
  }
  fprintf(fp, "  Used hosts:  %d\n", used_hosts);
  fprintf(fp, "Unused hosts:  %d\n", no_hosts - used_hosts);
  fprintf(fp, "\n----- Itemized statistics for each host --------\n");
  fprintf(fp, "--------------------------------------------------------\n");
  fprintf(fp, "   Host    Done  Abort  Avg.     Last done ");
  fprintf(fp, "    Lag    Delay  D/D+A\n");
  fprintf(fp, "--------------------------------------------------------\n");
  for (i = 0; i < no_hosts; i++) {
    h = hosts + h_order[i];
    dones += h->jobs.done;
    aborts += h->jobs.abort;
    lag = (h->jobs.last_time == 0)? 0 : latest - h->jobs.last_time;
    avg = (h->jobs.done == 0)? 0.0 : h->jobs.sum_time*1.0/h->jobs.done;
    delay = (h->jobs.done == 0)? 0.0 :
      (total_time - h->jobs.sum_time - lag)*1.0/h->jobs.done;
```

```
    percent = (delay == 0)? 0 : delay * 100.0/(delay + avg);

    total_avg += avg * h->jobs.done;
    total_avg2 += avg;
    total_delay += delay * h->jobs.done;
    total_delay2 += delay;
    total_lag += lag;
    total_percent2 += percent;

    fprintf(fp, "%10s %5d %5d %7.2fm  %-15s %6.2fm %6.2fm %5.1f%%\n",
        h->name, h->jobs.done, h->jobs.abort, avg /60.0,
        (h->jobs.last_time==0)? "-----": cc_short_time(h->jobs.last_time),
        lag / 60.0,  delay /60.0, percent);
  }
  fprintf(fp, "---------------------------------------------------------\n");
  fprintf(fp, "%10s %5d %5d %7.2fm  %15s %6.2fm %6.2fm %5.1f%%\n",
     "Total:", dones, aborts, total_avg /60 / dones, "",
     total_lag/60.0 /used_hosts, total_delay /60 /dones,
     total_delay*100 /(total_delay+total_avg)  );

  fprintf(fp, "%22s %7.2fm  %23s %6.2fm %5.1f%%\n",
          "", total_avg2/60/used_hosts, "",
          total_delay2/60 /used_hosts,
          total_percent2/used_hosts);
  fclose(fp);
}

void parallel_run(wait_q, run_q, done_q, init, no_hosts,hosts,h_order)
record * wait_q, * run_q, * done_q;
long init;
int no_hosts, *h_order;
host * hosts;
{
  int freed_hosts;

  freed_hosts = update_hosts_stat(no_hosts, hosts);
  printf("Freed_Hosts Before Executor & Controller is %d\n",freed_hosts);

  while ( wait_q->next != NULL  || run_q->next != NULL ) {
    executor(wait_q, run_q, no_hosts, hosts, h_order, init);
    freed_hosts = controller(wait_q, run_q, done_q, no_hosts, hosts);
    update_host_order(no_hosts, hosts, h_order);

    if (down_hosts(no_hosts, hosts) == no_hosts){
      printf("No more hosts to use.\n");
      exit(0);
```

```
    }
  }
}

/* for every running job on the run_q, checks and sees if it is finished or it should be
removed for various reasons */
int controller(wait_q, run_q, done_q, no_hosts, hosts)
record * wait_q, * run_q, * done_q;
int no_hosts;
host * hosts;
{
  record * p = run_q->next, * q = run_q;
  int freed_hosts;
  host * h;
  int * no_of_running_records = &(run_q->start_hist);

  printf("============== in Controller ===========================\n");
  freed_hosts = update_hosts_stat(no_hosts, hosts);

  while (p != NULL) {
    h = hosts + p->host_id;
    if (p_done(p)) {
      q->next = p->next;              /* remove p from run_q */
      (*no_of_running_records)--;
      p->end_time = get_f_mtime(p->f_names.out);
      p->status = DONE;
      insert_to_queue(p, done_q);          /* move to done_q */
      h->jobs.done++;                 /* update host's info */
      h->jobs.sum_time += p->end_time - p->start_time;
      if (h->jobs.last_time < p->end_time)
        h->jobs.last_time = p->end_time;
      h->fails = 0;
      if (h->status != DOWN)   /* P done */
        h->status = P_OFF;
      if (host_idle(h))
        freed_hosts++;

      p = q->next;
    }
    else if (high_load_or_down(h) || f_fail_too_many(p) ||
                                          p_terminated_or_little_cpu(h,p) ) {
      if (h->status == P_ON)
        h->status = P_OFF;
      h->fails++;
      h->jobs.abort++;
      q->next = p->next;              /* remove p from run_q */
```

```
      (*no_of_running_records)--;
      resubmit(wait_q, p, h);
      p = q->next;
    }
    else {
      q = q->next;
      p = p->next;
    }
  }
  printf("============== End of Controller ==========================\n");
  return freed_hosts;
}

void executor(wait_q, run_q, no_hosts, hosts, h_order, init)
record * wait_q, * run_q;
long init;
int no_hosts, *h_order;
host * hosts;
{
  int i, newly_runs = 0, no_waits = wait_q->start_hist;
  record *run_q_tail_1 = run_q, * run_q_tail_2, * p, * r;

  printf("!!!!!!!!!!!! Begin of Executor --------\n");

  while (run_q_tail_1->next != NULL)  /* find the run_q tail */
    run_q_tail_1 = run_q_tail_1->next;
  run_q_tail_2 = run_q_tail_1;        /* store it in run_q_tail_2 */

  while (wait_q->next != NULL &&
     (i=find_idle_host(no_hosts,hosts,h_order,run_q,no_waits,init))!=-1){
    dispatch(wait_q, run_q, &run_q_tail_2, hosts+i, i);
    no_waits--;
    newly_runs++;
  }
  if (newly_runs > 0 && newly_runs < 5)
    sleep(6 - newly_runs);

/* the reason not to check rsh() success in dispatch(): have to wait for 2 or 3 seconds after
each dispatch; which adds up to minutes if no. of hosts is large; thus wasting too much
time there. */
  for (p = run_q_tail_1->next; newly_runs > 0; newly_runs--, p = r) {
    r = p->next;          /* save p's next due to side-effect */
    check_rsh_success(wait_q, run_q, p, hosts+p->host_id);
    printf("------ Check rsh success finished!!!! ------\n");
  }
  printf("!!!!!!!!!!!! End of Executor ----------\n");
```

```
}

void dispatch(wait_q, run_q, run_q_tail, h, host_id)
record * wait_q, * run_q, ** run_q_tail;
host * h;
int host_id;
{
  char sys_call[SYS_CALL_LEN];
  record *p = wait_q->next;

  strcpy(p->host_name, h->name);
  p->host_id = host_id;
  make_out_fnames(p);

  /* sh() way (1> or 2>/dev/null) to redirect output or error to /dev/null */
  sprintf(sys_call, "rsh %s cd /net/nmrserv/data3/jliu 1>/dev/null 2>/dev/null \";\"
/usr/bin/nice -%d %s>%s \";\" chmod %o %s 2>/dev/null &",p->host_name, PRIORITY,
p->f_names.in, p->f_names.out, DONE_MODE, p->f_names.out);
  system(sys_call);

  p->start_time = p->end_time = get_clock();   /* end_time for checking */
  p->status = RUNNING;
  h->status = P_ON;
  wait_q->next = p->next;     /* remove p from wait_q */
  wait_q->start_hist--;       /* no of records in the queue */
  run_q->start_hist++;
  (*run_q_tail)->next = p;
  p->next = NULL;
  *run_q_tail = p;
}

int down_hosts(no_hosts, hosts)
int no_hosts;
host * hosts;
{
  int downs = 0;

  for (no_hosts-- ; no_hosts >= 0; no_hosts--)
    if ((hosts+no_hosts)->status == DOWN)
      downs++;
  return downs;
}

/* reorder the hosts according to their past performance so that executor() will choose the
next one in this order */
void update_host_order(no_hosts, hosts, h_order)
```

```
int no_hosts, * h_order;
host * hosts;
{
  int i, j, tmp;

  for ( i = 0; i < no_hosts; i++)
    for (j = i+1; j < no_hosts; j++)
      if (higher_order(hosts+h_order[j], hosts+h_order[i])) {
        tmp = h_order[i];
        h_order[i] = h_order[j];
        h_order[j] = tmp;
      }
}

/* if j has not finished any job: j is not better than i (no info) if j has finished some jobs
and i has not: j is better than i if j has finished some jobs and i has also: compare their
speeds */
int higher_order(j, i)
host * j, *i;
{
  return (j->jobs.done != 0) && (i->jobs.done == 0 || j->jobs.sum_time*1.0/j->jobs.done
                                  < i->jobs.sum_time*1.0/i->jobs.done);
}

void resubmit(wait_q, p, h)
record * wait_q, * p;
host * h;
{
  printf(">>>>>>>>> Begin of Resubmit ..........................\n");
  if (h->status != DOWN){
    kill_processes(p);      /* kill the running process if any */
  }
  remove_out_files(p);
  p->f_fail_cnts = 0;
  p->status = ABORTED;
  insert_to_queue(p, wait_q);
  log_file_r(p);
  printf(">>>>>>>>> End of  Resubmit ......................\n");
}

char * cctime(time)
long time;
{
  char * t_buf;

  t_buf = ctime(&time);
```

```
  t_buf[strlen(t_buf)-1] = '\0';      /* remove \n at the end */
  return t_buf;
}

char * cc_short_time(time)
long time;
{
  char * t_buf;

  t_buf = ctime(&time);
  t_buf[strlen(t_buf)-6] = '\0';      /* remove ' 1991\n' at the end */
  return t_buf + 4;                   /* remove 'Sat' in the front */
}

void kill_processes(r)
record * r;
{
  char sys_call[SYS_CALL_LEN];

  sprintf(sys_call, "rsh %s kill %d 1>/dev/null 2>/dev/null &", r->host_name,
                                                                   r->rpss.pid);
  system(sys_call);
}

int high_load_or_down(h)
host * h;
{
  return h->status == DOWN || h->load_1 > BUSY_LOAD;
}

int f_fail_too_many(r)
record * r;
{
  return r->f_fail_cnts > FILE_FAIL_LIMIT;
}

int p_terminated_or_little_cpu(h, p)
host * h;
record * p;
{
  int pid, terminated = 0;

  /* use p->end_time as the last check time variable before it's done */
  if (get_clock() - p->end_time < 90)  /* to reduce the network traffic */
    return 0;                  /* not check if < 90 seconds */
```

```
  if (get_pid(h, p, &pid, &p->rpss.cpu) == 0)
    terminated = 1;
  else if (p->rpss.pid > 0)
    terminated = (pid != p->rpss.pid);
  else {            /* failed last time, but okay now */
    terminated = (pid != - p->rpss.pid);
    if (terminated == 0)     /* back to normal */
      p->rpss.pid = - p->rpss.pid;
  }
  p->end_time = get_clock();  /* update the last check time */

  if (terminated == 0)        /* not terminated, check cpu percent */
    return p->rpss.cpu *100.0/(p->end_time - p->start_time) < CPU_PERCENT;

/* the job may be just finished between the calls of p_done() and this function in
controller(); if so, don't remove the job; the next round of checking p_done() will move
the job to done_q. On the other hand, the job may be permanently terminated due to some
errors before it is done; if so, the next time the following code catches it */
  if (p->rpss.pid > 0) {      /* 1st time detect terminated; not to remove */
    p->rpss.pid = - p->rpss.pid;
    return 0;
  }
  else {                      /* 2nd time detect terminated; report to remove */
    p->rpss.pid = - p->rpss.pid;
    return 1;
  }
}

void check_rsh_success(wait_q, run_q, p, h)
record * wait_q, * run_q, * p;
host * h;
{
  if ((p->rpss.no_ps = get_pid(h, p, &p->rpss.pid, &p->rpss.cpu)) > 0) {
    p->end_time = get_clock();      /* update the last checking time */
    return;                         /* rsh successfully started */
  }
  if (p_done(p))   /* finished already before checking */
    return;

  printf(">>>>>> Unsuccessful start on %s <<<<<<\n", p->host_name);
  run_q->start_hist--;        /* no of records in the queue */
  while (run_q->next != p)    /* locate p in run_q */
    run_q = run_q->next;
  run_q->next = p->next;      /* remove p from run_q */
  remove_out_files(p);
  p->start_time = p->end_time = 0;
```

```
    p->status = WAITING;
    insert_to_queue(p, wait_q);
    h->status = P_OFF;
    h->fails++;
  }

/* get process ids and cpu time used on the processes on remote machines */
int get_pid(h, r, pid, cpu)
host * h;
record * r;
int *pid, * cpu;
{
  char sys_call[SYS_CALL_LEN], s[SYS_CALL_LEN], tty[4], prgm[NAME_LEN];
  char tmp_f[NAME_LEN];
  FILE * fp;
  int no_ps = 0, minute, second;

  generate_unique_fname("pid", tmp_f);

  sprintf(sys_call, "rsh %s ps -e|grep %s>%s",r->host_name,r->f_names.in,tmp_f);

  get_host_stat(h);          /* to prevent a rare case */
  if (h->status == DOWN)
    return no_ps;

  signal(SIGALRM, alarm_hd);
  alarm(15);          /* treat as down if no response in 15 seconds */

  if (setjmp(env_buf)) {       /* save the stack positions */
    remove_file(tmp_f);
    printf("After 15 seconds ...: get_pid() failed on %s.\n", h->name);
    return 0;         /* treat as non-success: no_ps = 0 */
  }
  else {
    /* code after normal setjmp()  */
    system(sys_call);

    alarm(0);             /* cancel alarms */
    signal(SIGALRM, SIG_IGN);

    if ((fp = fopen(tmp_f, "r")) == NULL) {
      printf("Cannot open file %s\n", tmp_f);
    }
    else {
      printf("-------- Processes IDs and commands ------------\n");
      while (fgets(s, SYS_CALL_LEN, fp) != NULL) {
```

```
        sscanf(s, "%d %s %d:%d %s", pid, tty, &minute, &second, prgm);
        if (!strcmp(tty, "?")){        /* tty = "?" means no control terminal  */
          no_ps = 1;
          *cpu = minute * 60 + second;
          break;
        }
      }
      fclose(fp);
    }
    remove_file(tmp_f);
    return no_ps;
  }
}

int find_idle_host(no_hosts, hosts, h_order, run_q, no_waits, init)
int no_hosts, * h_order, no_waits;
host * hosts;
record * run_q;
long init;
{
  int i, j;

  for (j = 0; j < no_hosts; j++) {
    i = h_order[j];
    if (host_idle(hosts+i)){
      return stop_use(hosts+i,j,hosts,h_order,run_q,no_waits,init)? -1 : i;
    }
    if ((hosts+i)->fails >= HOST_FAIL_LIMIT)
      (hosts+i)->fails = ((hosts+i)->fails + 1) % (2 * HOST_FAIL_LIMIT);
  }
  return -1;
}

/* this functions implements a better load balancing algorithm: if machine is slow enough
(from past performance), then it is better not to use it, instead wait for a faster machine to
finish its current job and to take the remaining job(s)
note: this only occurs when the number of waiting jobs are small relative to the hosts'
speed differences. */
int stop_use(h, no_hosts_faster, hosts, h_order, run_q, no_waits, init)
host * h, * hosts;
int no_hosts_faster, * h_order, no_waits;
record * run_q;
long init;
{
  int i, j, stop_point = 0;
  record * p;
```

```
  host * h_i;
  float epsilon, avg_time_h_i;

  if (h->jobs.done == 0){
    return 0;       /* 1st time to use h; no info about its speed yet */
  }

  for (j = 0; j < no_hosts_faster; j++) {
    i = h_order[j];
    h_i = hosts + i;
    if ((p = search_record_by_host_id(run_q, i)) == NULL)
      continue;             /* not running a job; don't count on it */
    epsilon = 1 - 1.0 * (get_clock() - p->start_time)/
                              (h_i->jobs.sum_time / h_i->jobs.done);
                                /* use h_i's real avg. run time to calculate epsilon */
    if (epsilon < 0)  /* h_i already finished its current job */
      epsilon = 0;
    avg_time_h_i = 1.0*(h_i->jobs.last_time - init)/h_i->jobs.done;
                                /* include both avg. run time and avg. delay */
    stop_point += (int) ( 1.0 * (h->jobs.sum_time/h->jobs.done) /
             avg_time_h_i - epsilon);
  }
  return no_waits <= stop_point;
}

record * search_record_by_host_id(queue, host_id)
record * queue;
int host_id;
{
  for (queue = queue->next; queue != NULL; queue = queue->next)
    if (queue->host_id == host_id)
      return queue;
  return NULL;
}

int update_hosts_stat(no_hosts, hosts)
int no_hosts;
host * hosts;
{
  static long last_check_time;  /* control status inquery interval */
  int i, freed_hosts = 0;

  if (get_clock() - last_check_time < 5)  /* < 5 seconds */
    return freed_hosts;                     /* not check */

  for (i = 0; i < no_hosts; i++) {
```

```
    get_host_stat(hosts + i);
    print_hosts(stdout, hosts+i, 1);
    if (host_idle(hosts + i))        /* newly become idle machine */
      freed_hosts++;
  }
  last_check_time = get_clock();      /* update the last check time */
  printf("_________________ End of stat report _______\n");
  return freed_hosts;
}

int host_idle(h)
host * h;
{
  return (h->status == P_OFF) &&(h->fails == 0? h->load_1 < BUSY_LOAD :
                      h->load_1 <= IDLE_LOAD) && (h->fails < HOST_FAIL_LIMIT);
}

void log_file_r(p)
record * p;
{
  FILE * fp;
  char log_f[NAME_LEN], sys_call[SYS_CALL_LEN];

  strcpy(log_f, LOG_FILE_PREFIX);
  strcat(log_f, "-log-r");
  fp = fopen(log_f, "a");
  fprintf(fp,"---The following processes were killed and resubmitted---\n");
  print_record(fp, p);        /* print the record to fp */
  fclose(fp);
}

void log_file_d(p)
record * p;
{
  FILE * fp;
  char log_f[NAME_LEN], sys_call[SYS_CALL_LEN];

  strcpy(log_f, LOG_FILE_PREFIX);
  strcat(log_f, "-log-d");

  while (p != NULL) {
    fp = fopen(log_f, "a");
    print_record(fp, p);        /* print the record to fp */
    fclose(fp);
    sprintf(sys_call," cat %s>>%s 2>/dev/null", p->f_names.out,"P-log-d");
    system(sys_call);
```

```
    remove_out_files(p);
    p = p->next;
  }
}

/* get the last modified time of a file or the current clock time */
time_t get_f_mtime(file)
char * file;
{
  struct stat sbuf;
  extern int errno;

  if (stat(file, &sbuf) != -1)
    return sbuf.st_mtime;
  else
    return get_clock();
}

void remove_out_files(p)
record * p;
{
  remove_file(p->f_names.out);
}

/* checks if job is done according to an predefined agreement on the modes of output
files */
int p_done(p)
record * p;
{
  struct stat sbuf;
  extern int errno;

  if (stat(p->f_names.out, &sbuf) == -1) {
    perror("error stat():.............----->>>");
    if (errno == ENOENT) {    /* no file yet */
      p->f_fail_cnts++;
      printf("Inquery %s: Fail: Counts = %d\n",p->f_names.out,p->f_fail_cnts);
    }
    else
      perror("Fatal error occurred in stat(): ");
    return 0;
  }
  else {
    p->f_fail_cnts = 0;      /* reinitialize to 0 whenever success */
    if ((sbuf.st_mode & MODE_MASK) == DONE_MODE)
      printf("------->>> %s finished ......!!!!!!!!!!!\n", p->f_names.out);
```

```
    return ((sbuf.st_mode & MODE_MASK) == DONE_MODE);
  }
}

/* generate a group of unique file names for o, f. */
void make_out_fnames(p)
record * p;
{
  static int fname_generator;

  do {
    fname_generator = fname_generator % 999 + 1;
    name_constructor("tmp", "o", fname_generator, p->f_names.out);
  } while (check_f_exist(p));
}

/* if any output file already exists, don't use them */
int check_f_exist(p)
record * p;
{
  return f_exists(p->f_names.out);
}

/* search host id by host name */
int search_host_id(host_name, no_hosts, hosts)
char * host_name;
int no_hosts;
host * hosts;
{
  int i;

  for (i = 0; i < no_hosts; i++)
    if (strcmp(host_name, (hosts + i)->name) == 0)
      return i;
  return -1;
}

void print_record(fp, r)
FILE * fp;
record * r;
{
  fprintf(fp, "-----------------------------------------------\n");
  fprintf(fp, "host name: %s\n", r->host_name);
  fprintf(fp, "host id: %d\n", r->host_id);
  fprintf(fp, "remote processes:: no_ps = %d\n", r->rpss.no_ps);
  fprintf(fp, "pid = %d; cpu%% = %.1f", r->rpss.pid, r->rpss.cpu == 0? 0:
```

```
                                        r->rpss.cpu*100.0/(r->end_time - r->start_time));
  fprintf(fp, "\n");
  fprintf(fp, "program name: %s\n", r->f_names.in);
  fprintf(fp, "file fail counts = %d\n", r->f_fail_cnts);
  fprintf(fp, "start time = %s", ctime(&(r->start_time)));
  fprintf(fp, "end time = %s", ctime(&(r->end_time)));
  fprintf(fp, "elapsed time = %.2f min.\n",(r->end_time - r->start_time)/60.0);
  fprintf(fp, "status:  ");
  switch (r->status) {
     case -1:  fprintf(fp, "Aborted");
           break;
     case 0:  fprintf(fp, "Waiting");
           break;
     case 1:  fprintf(fp, "Running");
           break;
     case 2:  fprintf(fp, "Done");
           break;
  }
  fprintf(fp, "\n------ The Result of the Program As Follow ------\n\n");
}

void print_q(fp, q, q_name)
FILE * fp;
record * q;
char * q_name;
{
  if (!DEBUG)        /* debug off */
    return;
  fprintf(fp, "\n===== There Are %d Jobs On %s =====\n",q->start_hist,q_name);
  q = q->next;       /* skip the dummy head */
  fprintf(fp, "-------------- Beginning of %s ------------\n", q_name);
  while (q != NULL) {
    print_record(fp, q);
    q = q->next;
  }
  fprintf(fp, "-------------- End Of %s ------------\n\n", q_name);
}

/* builds up a waiting queue */
record * construct(INIT)
char *INIT;
{
  FILE *p_fp;
  record *wait_q, *r;
  char junk_name[NAME_LEN];
  int no_prog;
```

```
  if ((p_fp = fopen(INIT, "r")) == NULL) {
    printf("There Must Be File Which Contain The Name Of The Programs\n");
    exit(1);
  }

  while (skip_space_or_comments(p_fp) != EOF){
    fscanf(p_fp, "%s", junk_name);      /* counting */
    no_prog++;
  }
  printf("Total Programs Are %d\n", no_prog);
  if (no_prog == 0){
    printf("There Is No Program In %s!!!\n", INIT);
    exit(0);
  }

  wait_q = get_a_record("dummy", WAITING);    /* dummy head */
  fseek(p_fp, 0, 0);            /* rewind the file */
  while (skip_space_or_comments(p_fp) != EOF){
    r = get_a_record("", WAITING);
    r->start_hist = no_prog;
    fscanf(p_fp, "%s", r->f_names.in);
    insert_to_queue(r, wait_q);
  }
  fclose(p_fp);
  return wait_q;
}

/* insert p to queue which has a dummy head! */
void insert_to_queue(p, queue)
record * p, *queue;
{
  p->next = queue->next;
  queue->next = p;
  queue->start_hist++;  /* save as no of records in the queue */
}

record * get_a_record(host_name, status)
char * host_name;
int status;
{
  record * r;

  r = (record *) malloc(sizeof(record));
  strcpy(r->host_name, host_name);
  r->host_id = -1;
```

```
  r->rpss.no_ps = 0;
  r->rpss.pid = 0;
  r->rpss.cpu = 0;
  r->start_hist = 0;
  r->f_names.in[0] = '\0';
  r->f_names.out[0] = '\0';
  r->f_fail_cnts = 0;
  r->start_time = r->end_time = 0;
  r->status = status;
  r->next = NULL;
  return r;
}

void log_file_init(suffix, flag)
char * suffix, * flag;
{
  FILE * fp;
  char log_f[NAME_LEN], hostname[NAME_LEN];

  strcpy(log_f, LOG_FILE_PREFIX);
  strcat(log_f, "-log-");
  strcat(log_f, suffix);
  fp = fopen(log_f, "a");
  fprintf(fp, "\n----------------------------------------------\n");
  fprintf(fp, "File: %s\n", log_f);
  fprintf(fp, "    (automatically generated log file)\n");
  gethostname(hostname, NAME_LEN);
  fprintf(fp, "Host running Program:  %s\n", hostname);
  fprintf(fp, "----------------------------------------------\n");

  if (strcmp(flag, "parms") == 0)
    print_user_parms(fp, 1);
  fclose(fp);
}

int * init_host_order(no_hosts)
int no_hosts;
{
  int * h_order, i;

  h_order = (int *) malloc(sizeof(int) * no_hosts);
  for (i = 0; i < no_hosts; i++)     /* init to given order in .pmcnp_hosts */
    *(h_order + i) = i;
  return h_order;
}
```

```
/* gets the number of hosts from file .hosts which must exist in the current working
directory in order for program to run;   then sets up a host table */
host * init_host_table(no_hosts)
int * no_hosts;
{
  FILE *host_fp;
  host *hosts, *tmp;
  char junk_name[NAME_LEN];

  if ((host_fp = fopen(".hosts", "r")) == NULL) {
    printf("There Must Be File .hosts In The Current Directory,\n");
    printf("Which Contains A List Of Hosts For Program To Use.\n");
    exit(1);
  }

  *no_hosts = 0;
  while (fscanf(host_fp, "%s", junk_name) != EOF)      /* counting */
    (*no_hosts)++;

  printf("Total Hosts Is %d\n", *no_hosts);
  if (*no_hosts == 0){
    printf("%s", "There Is No Host In .hosts!\n");
    exit(1);
  }

  hosts = tmp = (host *) malloc(sizeof(host) * *no_hosts);
  fseek(host_fp, 0, 0);              /* rewind the file */
  while (fscanf(host_fp, "%s", tmp++->name) != EOF)
    ;                          /* get machine names */
  fclose(host_fp);
  init_hosts_stat(hosts, *no_hosts);
  print_hosts(stdout, hosts, *no_hosts);
  return hosts;
}

void print_hosts(fp, hosts, no_hosts)
FILE * fp;
host * hosts;
int no_hosts;
{
  host * h = hosts;

  if (!DEBUG)
    return;
  printf("-------- Host State --------\n");
  for (; no_hosts > 0; no_hosts--, h++) {
```

```
    fprintf(fp, "%10s %8d %6.2f %6.2f %6.2f %3d %3d\n",
                    h->name,h->up,h->load_1,h->load_5,h->load_15,h->status,h->fails);
    fprintf(fp, "\t\t%4d %4d %8.1f min. %8.1f\n", h->jobs.done, h->jobs.abort,
       (h->jobs.done == 0)? 0: h->jobs.sum_time/60.0/h->jobs.done,h->jobs.last_time);
  }
  printf("-------- End of Host State --------\n");
}

void init_hosts_stat(hosts, no_hosts)
host * hosts;
int no_hosts;
{
  for (; no_hosts > 0; no_hosts--) {

    hosts->last_check_time = 0;
    hosts->status = P_OFF; /* init to up */
    get_host_stat(hosts);    /* set to DOWN if down */
    hosts->fails = 1;        /* set fails to 1 first so that for the first time, host_idle() will use
                                                        IDLE_LOAD to test */
    hosts->jobs.done = 0;
    hosts->jobs.abort = 0;
    hosts->jobs.sum_time = 0;
    hosts->jobs.last_time = 0;
    hosts++;
  }
}

void alarm_hd()
{
  longjmp(env_buf, 0);        /* alarm off, return to saved state */
}

void get_host_stat(h)
host *h;
{
  FILE * fp;
  char sys_call[SYS_CALL_LEN], tmp[NAME_LEN], tmp_f[NAME_LEN];

  if (get_clock() - h->last_check_time <= 5){   /* at least 5 seconds */
    return;
  }

  signal(SIGALRM, alarm_hd);
  alarm(15);
  if (setjmp(env_buf)) {
    printf(" %s Down:(rsh host uptime)\n", h->name);
```

```
    down_setup(h);
  }
  else {
    generate_unique_fname("s", tmp_f);
    sprintf(sys_call, "rsh %s uptime > %s 2>/dev/null", h->name, tmp_f);
    system(sys_call);

    alarm(0);
    signal(SIGALRM, SIG_IGN);

    if ((fp = fopen(tmp_f, "r")) == NULL)
      printf("Cannot open %s\n", tmp_f);
    else {
    /* read in the information in the following format */
    /* 3:19m  up 8 days, 4:24, 1 user, load average: 0.13, 0.02, 0.01 */

      while (1) {
        fscanf(fp, "%s", tmp);
        if (strcmp(tmp, "average:") == 0)
          break;
      }
      h->up = -1; /* dummy number, skip the time part */
      fscanf(fp, "%f,%f,%f", &h->load_1, &h->load_5, &h->load_15);
      fclose(fp);
      if (h->status == DOWN)        /* if DOWN before, up now */
        h->status = P_OFF;          /* otherwise, don't alter */
    }
  }
  remove_file(tmp_f);
  h->last_check_time = get_clock();    /* update last checking time */
}

/* get the clock time in terms of seconds since Jan. 1, 1970 */
time_t get_clock()
{
  struct timeval t;
  struct timezone tz;

  if (gettimeofday(&t, &tz) != 0)
    perror("Error Occurred In gettimeofday(): ");
  else
    return t.tv_sec;        /* seconds since Jan. 1, 1970 */
}

void remove_file(fname)
char * fname;
```

```
{
  char sys_call[SYS_CALL_LEN];

  sprintf(sys_call, "rm %s 2>/dev/null", fname);
  system(sys_call);
}

void generate_unique_fname(suffix, fname)
char * suffix, * fname;
{
  static int fname_generator;

  do {
    fname_generator = fname_generator % 999 + 1;
    name_constructor("tmp", suffix, fname_generator, fname);
  } while (f_exists(fname));
}

char * name_constructor(prefix, suffix, f_count, name)
char * prefix, * suffix, * name;
int f_count;
{
  char f_count_chars[NAME_LEN];

  if (f_count < 10)
    sprintf(f_count_chars, "00%d", f_count);
  else if (f_count < 100)
    sprintf(f_count_chars, "0%d", f_count);
  else
    sprintf(f_count_chars, "%d", f_count);
  sprintf(name, "%s%s%s", prefix, f_count_chars, suffix);
  return name;
}

void down_setup(h)
host * h;
{
  h->status = DOWN;
  h->up = 0;
  h->load_1 = h->load_5 = h->load_15 = BUSY_LOAD + 1;
  h->users = 0;
  printf("Machine %s Is Down....!!!\n", h->name);
}

/* initializes the user defined parameters according to file .parms in the current working
directory; otherwise, sets them to defaults. */
```

```
void init_user_parms()
{
  FILE *parm_fp;

  set_default_parms();        /* set default first */

  if ((parm_fp = fopen(".parms", "r")) == NULL) {
    printf("Because .parms Doesn't Exist In The Current Directory,\n");
    printf("All The Parameters Are Set To Their Default Values:\n\n");
    print_user_parms(stdout, 1);     /* always print */
  }
  else {
    while (skip_space_or_comments(parm_fp) != EOF)
      read_and_parse_a_parm(parm_fp);            /* parse parms */

    if (PRIORITY < 10) {
      printf("\nYour Priority Changed From %d To 10.  Be Nice!\n", PRIORITY);
      PRIORITY = 10;
    }
    print_user_parms(stdout, DEBUG);
    fclose(parm_fp);
  }
}

void read_and_parse_a_parm(fp)
FILE * fp;
{
  char p_line[LINE_LEN], p_name[NAME_LEN], p_value[NAME_LEN];
  double atof();             /* have to declare it */

  fgets(p_line, LINE_LEN, fp);
  sscanf(p_line, "%s %s", p_name, p_value);   /* syntax: p_name p_value */
          /* remaider of the line is ignored */

  if (strcmp(p_name, "IDLE_LOAD") == 0)
    IDLE_LOAD = atof(p_value);
  else if (strcmp(p_name, "BUSY_LOAD") == 0)
    BUSY_LOAD = atof(p_value);
  else if (strcmp(p_name, "CPU_PERCENT") == 0)
    CPU_PERCENT = atoi(p_value);
  else if (strcmp(p_name, "PRIORITY") == 0)
    PRIORITY = atoi(p_value);
  else if (strcmp(p_name, "DEBUG") == 0)
    DEBUG = atoi(p_value);
  else if (strcmp(p_name, "LOG_FILE_PREFIX") == 0)
    strcpy(LOG_FILE_PREFIX, p_value);
```

```
  else
    printf("Parameter Name: %s Not Defined", p_name);
}

int skip_space_or_comments(fp)
FILE * fp;
{
  int c;
  char comments[LINE_LEN];

  while ((c=fgetc(fp)) == ' ' || c == '\t' || c=='\n' || c=='#')
    if (c == '#')
      fgets(comments, LINE_LEN, fp);        /* skip comments */
  ungetc(c, fp);      /* not comments; push c back onto fp */
  return c;
}

void print_user_parms(fp, msg_flag)
FILE *fp;
int msg_flag;
{
  if (!msg_flag)
    return;   /* not print */

  fprintf(fp, "\n------ Parameter setting ------\n");
  fprintf(fp, "IDLE_LOAD = %.2f\n", IDLE_LOAD);
  fprintf(fp, "BUSY_LOAD = %.2f\n", BUSY_LOAD);
  fprintf(fp, "CPU_PERCENT = %d\n", CPU_PERCENT);
  fprintf(fp, "PRIORITY = %d\n", PRIORITY);
  fprintf(fp, "DEBUG = %d\n", DEBUG);
  fprintf(fp, "LOG_FILE_PREFIX = %s\n", LOG_FILE_PREFIX);
  fprintf(fp, "------ End of Parameter setting ------\n");
}

/* preset default values for various parameters; can be overriden by the corresponding
parameter in .parms */
void set_default_parms()
{
  IDLE_LOAD = 1.5;
  BUSY_LOAD = 4.0;
  CPU_PERCENT = 20;
  PRIORITY = 10;
  DEBUG = 0;
  strcpy(LOG_FILE_PREFIX, "p");
}
```

```
int f_exists(f_name)
char * f_name;
{
  struct stat sbuf;
  extern int errno;

  if (stat(f_name, &sbuf) == -1)
    return (errno != ENOENT);        /* not exist */
  else
    return 1;                        /* exists */
}
```

VITA

Jian Liu

Candidate for the Degree of

Master of Science

Thesis: DESIGNING AND BUILDING AN EFFICIENT APPLICATION ON A UNIX NETWORK FILE SYSTEM

Major Field: Computer Science

Biographical:

Personal Data: Born in Beijing, P. R. China, March 6, 1966, the son of Maotang Liu and Shujun Yang

Education: Graduated from Beijing No. 35 High School, Beijing, P. R. China in 1984; received Bachelor of Medicine degree from Capital University of Medicine, Beijing, P. R. China, in July 1989. Completed the requirements for the Master of Science degree at Oklahoma State University in May, 2000.

Experience: Employed as a physician in charge at Beijing Goodwill Hospital, 1989-1996.

Professional Membership: China Medical Society.