A Load Sharing Algorithm for a Workstation Environment

Richard Korry

Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

Technical Report 86-10-03
October 1986

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Abstract: This thesis describes research showing that load sharing is possible and desirable in the workstation environment. It points out some important aspects of the workstation environment and how they affect the design of a load sharing algorithm. A new load sharing algorithm that has been implemented and tested is presented.

Table of Contents

List of Figures

ACKNOWLEDGEMENTS

First, my thanks goes to my advisor, John Zahorjan, and my "other" advisor, Ed Lazowska, for their advise, suggestions, and prodding (when it was necessary). I also want to thank the Eden project for providing the necessary financial support throughout my stay at the University of Washington. In particular, my hat goes off to Andrew Black, Norm Hutchinson, Calton Pu, and Mike Schwartz.

I must thank Roslyn for her support and love through these troubled times. Finally, I want to dedicate this thesis to my father, who will never be able to see this work.

Chapter 1

Introduction

Imagine that you are sitting at your workstation, using all the processor cycles available, while next to you another workstation lies idle. It makes sense that some of your work should be run on the idle machine. Current systems, however, require that you explicitly send your work to the other computer. Ideally, the system you are using should automatically move your work from your machine to the idle one. This idea is known as *load sharing.*

Load sharing can be viewed as a global instance of the scheduling problem. Load sharing automatically and transparently assigns tasks to processors to reduce response time. Thus, in the above scenario, load sharing would increase your productivity by decreasing response time through moving work from your busy machine to the idle one in an automatic and efficient way.

Load sharing seems easy at the two extremes of machine utilization. Almost any load sharing algorithm will work well when only one machine is busy and many are idle. Alternatively, almost all algorithms are ineffective when every workstation is busy; a poorly conceived algorithm could easily increase response time. The important research issues in load sharing today consider how best to handle the large area in between these extremes.

This thesis presents the results of implementing a new load sharing algorithm at the University of Washington. This work shows that load sharing for the workstation environment is possible, desirable, affordable, and can be effective using simple policies. In the course of this work, new insights about load sharing in the workstation environment were discovered.

Chapter 2

The Environment: Distributed Systems

Over the past decade, *distributed systems* have been a focal point of research, development and debate. Liebowitz and Carson [26] offer a very general definition: "A distributed system is one in which the computing functions are dispersed among several physical computing elements." Enslow [17] narrows this to those with a system-wide operating system so that users are unaware that their work may be distributed throughout the system as needed. Such a system would share computing resources to a high degree. While some commercially available systems satisfy Liebowitz and Carson's definition, all fall short of Enslow's goal. Some research systems, however, are now approaching this ideal.

2.1 The Hardware

The hardware required for implementing a system such as Enslow's is currently available. Processor and memory price to performance ratios have fallen to the point that powerful single user workstations are now widely available. Meanwhile, the last decade has seen the rapid emergence of computer networks. These provide the communication links needed for sharing the expanding computing resources.

2.1.1 Computer Networks

Two types of networks currently dominate computer communication. Long haul networks, such as the ARPANET, connect computers over thousands of miles via dedicated lines and satellite and radio links. These types of networks typically have a low data transmission rate (on the order of 56 kbps) and high latency.

Local Area Networks (*LANs*), have evolved for short haul connections. Popular examples of LANs include contention buses, of which the best known is the Ethernet [30], and token rings [19]. Using coaxial cable, twisted pair, and fiber optic technology, LANs offer high data rates, on the order of 10 to 100 Mbps. This high bandwidth enables machines to use data remotely that previously they could only access locally. The combination of high bandwidth and low latency allows the sharing of resources over the network. Resources such as printers and secondary memory (typically hard disks) are now commonly shared.

2.1.2 Workstations

The *raison d'etre* of workstations is to provide highly interactive computing. They offer their individual users more processing power and main memory than the timeshared superminis of a few years back. A typical workstation today provides computing resources in the 1 MIPS range, a large main memory (4MB is not uncommon), and a high resolution, bit-mapped display with a pointing device such as a mouse. The workstation software allows multi-tasking and is often a port of a traditional time-sharing system (e.g., Unix or VMS [1]).

Combining LANs and workstations has produced a new style of architecture, the diskless workstation (see Figure 2.1). Diskless workstations (i.e., those lacking local secondary memory) use the LAN to get disk pages from a file server. A file server is a node on the network with local secondary memory (e.g., a hard disk) that provides secondary memory for the diskless workstation. The diskless workstation makes requests to the file server

[1] Unix is the trademark of AT&T Bell Laboratories. VMS is the trademark of Digital Equipment Corporation.

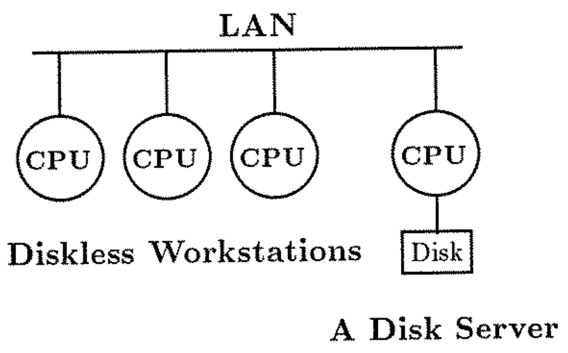

Figure 2.1: Architecture of Diskless Workstations.

via a network message. The file server responds with one or more network messages containing the information requested. The benefit of this architecture is that a single fast, large capacity disk is much more cost effective than many smaller, slower disks. The cost advantage for secondary memory allows additional purchases of relatively inexpensive workstations, which results in more resources for the same total system price. In addition, file sharing is easily provided by distributed file systems that allow a workstation to access any file on the network rather than limiting it to those on its file server.

2.2 The Software

Software technology has lagged behind hardware technology in the area of distributed systems. While some available systems share files, secondary memory and special peripherals, none share processors. A major reason is that most of the software running on workstations has been ported from the timesharing environment and was not designed for use in distributed systems. These older systems have relied heavily on pioneering research and development done at the Xerox Palo Alto Research Center. PARC developed much of the fundamental hardware and software for distributed systems: the Ethernet; the first workstation (the Alto); the bitmap display; the laser printer; the mouse; window managers; program development environments; Smalltalk; InterLisp and more.

However, some significant research has been done on achieving distributed systems that satisfy Enslow's definition. Some of the notable projects include Argus [27] at MIT, Clouds [1] at Georgia Tech, Eden [2] [8] at the University of Washington, and LOCUS [43] at UCLA. Eden was instrumental in my research and merits a brief review which will be useful for later reference.

The Eden project's goal was to build and use an integrated distributed computing system. Eden views the world as consisting of Eden Objects or *Objects*. Each Object is identified by a *capability* consisting of a unique identifier and a set of rights. An Object supports a set of operations that it exports to the world. For example, a Directory Object would support operations such as Insert, Delete, and LookUp. One Object asks another to perform an operation by means of an *invocation*. Invocations are Eden's Remote Procedure Call that includes the target Object's capability, value and result parameters, and a status parameter. Network transparency is supported; an Object's capability is all that is needed for an invocation. Because Objects are conceptually always active, explicit activation is not required.

The Eden Kernel supervises all Object activity. It has the responsibility for locating and possibly activating a target Object. Invocations pass through the Kernel to prevent Objects from forging capabilities or modifying their rights fields. The Kernel provides Objects with many primitives, including the ability to create new Objects of a given type, e.g., a new instance of the Directory Object.

The Eden Programming Language (EPL) [7] is an extension of Concurrent Euclid (CE) [21]. EPL provides language support for invocations so that they appear as intermodule procedure calls. Stub procedures that pack and unpack parameters for incoming and outgoing invocations are generated automatically. Internal concurrency using lightweight processes is provided along with monitors for concurrency control.

The Eden system has facilitated the writing of numerous distributed applications.

These include, among others, a mail system [3], personal calendar [20], nested transactions [34] and data replication [32].

2.3 Summary

According to Enslow, distributed systems should share all their resources in such a transparent manner that users remain unaware of it. The hardware technology required for such systems is now available but the software technology is lagging. Current software permits the sharing of files and peripherals but not processors. However, research systems that are capable of sharing processors are now emerging. Thus research into how best to share processors is timely and relevant.

Chapter 3

Overview of Load Sharing

The goal of this chapter is to provide the reader with a basic understanding of load sharing. It first defines load sharing and explains why it is useful. Then it presents an overview of load sharing concepts and criteria for evaluation. Finally, a review and comparison of the previous work in the field sets the stage for the research presented in Chapters 4 and 5.

3.1 What is Load Sharing?

In a network of autonomous computers with independent sources of work, the load on one computer will generally differ from that on the others. This gives rise to the situation where some machines have excess work while others remain idle [28]. A more efficient use of resources would be to distribute the load among all the computers. Load sharing[1] is a general term for efficiently sharing computing resources (processors) in a network. An ideal load sharing system would automatically and transparently migrate work as required from node to node so as to minimize response time. Users would not know whether their work was executing locally or remotely.

[1]The term load *balancing* has also been used to describe these ideas, drawing on the intuitive idea of balancing or equalizing the load among nodes. Load *sharing* is preferred because it does not specify anything about how the algorithm should work. That is, the load could be shared without being balanced.

3.2 The Need for Load Sharing in Distributed Systems

The cost of computers continues to drop while their performance continues to rise; users have more and more processing power available. Why go to the trouble of load sharing when users already have more power than they can use? This question has two answers.

The first looks at the traditional leapfrogging of hardware power and software needs. Software is written to the capabilities of the available hardware. As more cycles or memory become available, software demand for hardware resources is bound to increase. Much of the current software running on workstations was designed and developed on overburdened timesharing systems. Consequently, it does not take advantage of the increased capabilities of workstations. Programming environments that have been designed for use on workstations, such as Cedar [41], require large amounts of processing power to run well.

The second reason is that load sharing can provide an inexpensive way to increase the potential amount of processing power available to each user. In a network of twenty nodes, for example, each user could potentially use all twenty computers. Consider what could happen when a user compiles a program. With load sharing, the individual files or even the separate passes of the compiler could be executing concurrently on separate workstations. Although most of the time some of the other workstations will be busy, there are indications that they are often idle [42].

3.3 A Load Sharing Taxonomy

When only one node is busy and the rest lie idle, almost any plan to migrate work from the overloaded node to the rest will result in better response time. Conversely, if all the nodes have more work than they can handle, moving work around will not help. Between these extremes, the questions of where and when to migrate work, and what work to select, remain open. As the number of proposals have increased, taxonomies have emerged to organize and compare them (e.g., Wang and Morris [44]).

My own taxonomy looks at five issues in load sharing: static versus dynamic algorithms, information policies, load assessment policies, migration policies, and preemption versus nonpreemption.

- Static versus Dynamic:

  A *static* algorithm decides the migration path *a priori*, based on the average rather than the current state of the system. A simple example of a static algorithm is for node A to always send work to node B whenever A is busy. Static algorithms are useful when the workload is well defined in advance because they allow calculation of an optimal assignment of tasks to processors before execution begins. The daily transactions of a bank or a factory might fit this description. Static policies are appealing because they do not require communication of state information throughout the system and any required migration calculations take place off line. Thus the cost of computing a new workload distribution does not contribute to the system load.

  The problem with static policies is that they perform poorly when the actual system state is different from what was expected. Using the above example, if nodes A and B both are busy but node C is idle, then node C should receive work from node A, rather than node B. Unfortunately, most environments are incapable of providing an accurate profile of their average system state in advance.

  Dynamic algorithms use current state information when making load sharing decisions. The flexibility of dynamic algorithms, due to their ability to react to the state of the system, permits them to work in almost every environment. This flexibility, however, brings with it the disadvantage of having a higher cost than static algorithms.

  Dynamic algorithms can either be *distributed* or *centralized.* In a distributed algorithm, each node autonomously makes the migration decision in cooperation with the recipient node. This contrasts a centralized algorithm where a node must contact a central authority which provides the load sharing decision.

- Information Policy:

  An information policy answers the question: "What do I know about my load and what do I know about the loads on other machines?" The information policy selects the data to be gathered locally, decides what will be passed to other nodes, and chooses the manner in which to communicate it. For example, a simple policy could measure the queue length of runnable processes and broadcast its value to all other nodes every second.

  Information policies must take into account the impact on a workstation from taking local measurements. The more often we meter our node, the more timely our information; if we meter too often, however, we affect the very load we are trying to measure. A similar situation arises in transmitting our node's state to the rest of the network. Frequent broadcasts, especially as the size of the network grows, begin to clog the communication medium and force other nodes to spend time processing the messages. There is not one "best" information policy; each algorithm will have different requirements. Every policy, however, must balance the benefits of timely information with the costs to the nodes and the network.

- Load Assessment Policy:

  The load assessment policy, closely associated with the information policy, asks the question: "Should I load share now, and if so, which job?" It uses the data provided by the information policy to calculate the load existing on a node. Using this load level along with other information (e.g., the type of program, special resources required, etc.), the load assessment policy decides if this process should be moved. For example, a simple load assessment policy could classify a node as busy whenever the length of its run queue exceeded one and move jobs off the machine as dictated by the algorithm. Answering the question of which process to move can be as simple as "move the next one created" or as complicated as considering resource usage of all jobs in light of the other jobs on their processors. This question is dependent on many other aspects of the load sharing algorithm, such as the type of program (interactive or batch), and is discussed further.

- Migration Policy The migration policy asks the question: "Which node should receive (or send) the migrated process?" It decides this by using data from the information policy and/or by obtaining other information at that moment.

  A migration policy can be either sender or receiver initiated. Sender initiated means that the node dispatching work is responsible for locating a willing recipient. An example of a sender initiated policy is the "Shortest Runnable Queue" policy, which selects as the destination the node with the shortest queue of runnable processes. Another example is "Random", which selects a node at random regardless of the node's current state.

  In a receiver initiated policy, a potential host asks for work from another node. A busy node then migrates extra work to the requesting node. Examples of receiver initiated policies include "Longest Runnable Queue" and "Random".

  Sender and receiver initiated policies perform best at opposite ends of the machine utilization spectrum. Compare what happens to each type when all of the nodes in the network are busy. In the sender initiated version, each busy node wastes time futilely seeking an idle node to accept its work, although none exists. In contrast, the receiver initiated version does not attempt to load share because there are not any idle nodes seeking work. The converse occurs when the network is lightly loaded; idle "receivers" spend time looking for the small amount of work that does exists. Although the waste is "free" in this case, the efficiency of the algorithm still suffers.

  The performance difference between the two schemes has been explored by Eager, Lazowska, and Zahorjan [15]. They found that sender outperformed receiver initiated at light to medium loads while receiver proved better at heavy loads. Under their assumptions, although individual nodes might be heavily utilized quite often, high levels of total system use would be rare. Thus they concluded that sender initiated offers a distinct advantage. Wang and Morris [44], however, came to the opposite conclusion by arguing that efficiency counted more when system utilization was high rather than low.

Other initiation strategies are possible. For example, every node could periodically reevaluate itself and then contact other nodes to either look for more work or send off excess work. The algorithm presented in Section 5.2 is a hybrid combining sender initiated and receiver initiated components.

- Preemption versus Nonpreemption:

  The final point asks the question: "At what point can a process be migrated?" The two apparent possibilities are: at any time and only when a task is created. Process *preemption* means that a process can be migrated between nodes throughout its lifetime. *Nonpreemption* limits migration to the moment of process creation. Preemption allows greater flexibility and potentially more efficiency in the load sharing algorithm than one limited to nonpreemption. For example, if both nodes A and B have two jobs and both jobs on A finish, preemption allows for one of B's jobs to be moved to A. Without preemption, A will remain idle until the next process is created. Some algorithms require preemption and so are limited in that respect while others can use either.

  Implementing process migration during execution is nontrivial. Process state must be saved, any current operations suspended, and the forwarding of messages to the process at its new site must be done. Preemption has recently been achieved in a few systems: Demos/MP [33], Locus [11] and the V-system [42] .

3.4 Basic Criteria for All Load Sharing Algorithms

Evaluation of any load sharing algorithm should consider three criteria: stability, flexibility, and effectiveness. The algorithm must be stable at all load levels so that the system does not come to a stop or cease to perform useful work. The flexibility of the algorithm in handling computers of differing power, size, and architecture must be considered. Finally, the load sharing algorithm must be effective; the response time of a program must be reduced or, at a minimum, never increased.

At least two types of instability can occur in an algorithm. One type, called process thrashing, can occur in algorithms that allow both process preemption and multiple migrations of a process. At high levels of machine utilization, processes constantly move from one busy node to another without ever executing; each node spends all of its time sending and receiving jobs. The second type involves the overloading of idle nodes. When a node becomes idle, work should be migrated in an orderly fashion preventing all of the busy nodes from suddenly swamping the idle node. For example, suppose 20 nodes each have one excess process. If one node becomes idle, it would be a mistake for all other nodes to simultaneously send their excess process to the idle node.

The flexibility of a load sharing algorithm must be considered since the last thing remaining constant these days is the type of computers attached to a network. An algorithm that works with only one model of computer on a network of fixed size would soon be discarded. Any algorithm should be able to accommodate computers of differing power and size. Accommodating differences in architecture (heterogeneity) is a more difficult challenge since it involves juggling binary programs for the various types of machines.

Other questions about an algorithm's flexibility should be asked. Is the algorithm restricted only to workstations or timesharing systems? What happens when special resources (e.g., a main frame, vector processor, or processor pool) are added to the network? Can they be easily integrated into the load sharing model? Does the algorithm contain parameters that allow for fine tuning or modification of the work flow? How does the algorithm resolve the conflict between personal ownership and communal use of workstations?

The effectiveness of an algorithm involves whether its cost exceeds its benefit. An algorithm's benefit comes from a reduction in a program response time while the algorithm's cost can be broken down into three different parts. The first is the cost of the information policy. Policies such as "Shortest Queue First" (mentioned in Section 3.3) require state information about the entire network. The expense from frequent broadcasts of this information can dramatically increase as the network grows in size [28]. In general, the more timely the information and the wider its distribution, the greater its cost. The second cost involves the additional time required to locate an acceptable host for a task in the migration policy. This must consider both the time for success and that of failure. Clearly each must be reasonable with respect to total execution time. The third is the cost of migration and remote execution. This includes the transferance of binary programs, any required data, and the higher expense of communications between the remote program and the local user's display. In this respect, diskless workstations operating with a distributed file system have an advantage (or disadvantage). They never experience any added cost for remote execution because all the files must be transmitted over the network regardless of the locations of the user and the executing node. They are subject, however, to the higher cost of communications.

3.5 Review of Previous Work

3.5.1 Static Algorithms

Static policies drew most of the initial attention in load sharing for two reasons. The first is that the model a static algorithm requires is of a well behaved, well understood system with a good characterization of the workload. This simple model invited an analytical solution while more complex models did not. The second was that during the late 1970's, people began to attach minicomputers to large main frames in a star network configuration. Consequently, these people were interested in the problem of partitioning work between the central and the satellite processors in the most efficient manner.

Stone [39] used network flow models to deterministically find the optimal placement of processes so as to minimize the total response time. This technique was expanded and enhanced by others [9] [13]. Additional efforts to solve the problem included Chu [14] using linear integer programming and Ma [29] with a branch and bound method. Each solution required significant computational effort. More recently, Tantawi and Townsley rejected deterministic procedures in favor of probabilistic methods in a new static algorithm [40].

3.5.2 Dynamic Algorithms

Simulations

The model required by static algorithms did not correspond to the distributed systems emerging in the 1980's. These systems needed algorithms based on the model that the system's state changes dynamically. Although many dynamic algorithms have been proposed, most have only been simulated while few have been implemented.

Livny and Melman [28] looked at three different node information policies: broadcast state when a process starts or finishes; broadcast when idle, and poll when idle. Using simulation, they concluded that each works best under certain conditions. They also noted the rising communication costs of using network broadcasts as the number of nodes increases. In addition, they found that there is a high probability that there will be an idle node when other nodes have excess work.

Bryant and Finkel [10] proposed a preemptive algorithm based on the observation that the amount of time a job has already run provides a good estimate of a job's remaining amount of computation. They used this fact to select the best node for a task on the basis of the total amount of processing time remaining per node. More recently this idea was verified by Leland and Ott [25]. Analyzing three years worth of data from Unix systems revealed three types of processes: "CPU hogs", "I/O hogs", and everything else. This third category accounted for 98% of the number of processes but only 35% of the total amount of CPU consumed. After three seconds of CPU time, a linear relationship emerged between the amount of CPU used and the mean amount of CPU eventually needed. Their algorithm used this relationship in deciding to migrate only older tasks.

Their information policy consisted of broadcasting messages that contained the number of processes and amount of CPU time each process had used on each node. This global information allowed the migration of the CPU hogs to computers without CPU hogs. Simulation of this algorithm showed a great benefit in response time for CPU hogs with some minor improvement for the other type of processes. They also noted problems with their algorithm as the number of nodes in the system increased. It remains to be seen if the findings of Leland and Ott are duplicated in other environments.

Bidding algorithms have been proposed that would have nodes bid for work whenever new processes were created; the winner would be the highest bidder [18]. A receiver initiated variation of bidding called drafting was proposed by Ni, Xu, and Gendreau [31]. The information policy specified three types of load classes: light (can accept work), normal (can not accept work but does not need to migrate work), and heavy (needs to migrate work). Each node kept a table consisting of hosts and their state. To minimize state broadcasts, the authors considered two alternatives to always broadcasting: piggybacking information about transitions from normal to heavy on other messages and a mixed updating approach. The load assessment policy used the number of processes per node. Simulation of the algorithm showed a significant improvement in response time over no load sharing at all for a five processor network.

Eager, Lazowska, and Zahorjan [16] argued that simple policies performed adequately and scaled well. They noted that even complex algorithms would sometimes make mistakes, especially as they responded to minor variations in load. They felt that a simple policy could reduce response times substantially (although not optimally) with very low cost. Since the use of broadcasts does not scale well to very large networks, they proposed information and migration policies that operate without them. To verify the utility of their analysis, they simulated three migration policies: randomly selected nodes; randomly selected with probe; and shortest queue. Each policy used the length of the queue of runnable processes as its load factor. The load assessment policy used the idea of setting a threshold; a node would attempt to migrate work whenever the length of the run queue exceeded one. In the first version, when a node had more than one runnable process it randomly selected another node and sent it the new process. A process would be migrated only once to prevent process thrashing. The second policy selected some small number of nodes at random (this number being referred to as the probe limit) and asked (probed) each selected node if it was under threshold. If one was, it would receive the work. The final policy assumed global knowledge of the queue lengths for all the nodes (at no cost) and would send work to the node with the shortest queue length. This excellent but normally expensive policy was included for comparison against the other two. In addition, models of the system without load sharing (M/M/1 model) and with perfect load sharing (M/M/N model for N nodes) were used as asymptotic limits. The analysis showed that even the Random policy did much better than no load sharing at all and Random with Probe did almost as well as the Shortest Queue policy. This study pointed out that load sharing did not have to be complicated to help at almost all levels of network loads.

Implementations

Research based on implementations rather than simulations has begun to surface as the state-of-the-art in distributed systems has advanced. LOCUS, a distributed version of Unix, has been in operation at UCLA for more than two years. Although no paper on automatic load sharing has been published, the tools for process preemption have been implemented and ideas on load sharing have been outlined [11]. A user initiated mechanism is provided in the command interpreter on a per command basis [43].

Barak and Shiloh [4] designed a preemptive algorithm using an information policy that consisted of nodes frequently exchanging a small vector describing workstation loads. The exchange took place between nodes selected randomly; each node took the information received from the other and updated that currently held. The authors prove that this method allows load information to be kept current in the network with high probability at low cost. This algorithm was implemented in a small network running the MOS system and performed very well when the workload remained constant.

The Maitrd system [6], done at the University of California at Berkeley, allowed for Unix applications (properly modified) to migrate between Unix hosts. This nonpreemptive system used the length of the run queue exponentially smoothed over time (the Unix load average) for assessing load. The node had a threshold value for the load average. The value of the load average with respect to the threshold determined if the node would or could not accept work from other nodes. Any change of state was broadcast to the network. Each node maintained a list of hosts along with their current state. When migration was required, this list was traversed in a round-robin fashion so that the previously selected node was the last checked in the next search. Currently only the C and Pascal compilers have been included in the Maitrd system. Bershad reports improvements from 78% for interactive programs (due to less competition from compilations) to 69% for compilations.

Another system from UCB investigated use of a mixture of load factors in load assessment. Zatti [45] did preliminary research that showed no one load factor adequately covered both CPU and I/O intensive work. For CPU bound work, the CPU utilization fared poorly as a predictor of run time. The large variations in response time seen were attributed to the policies of the Unix scheduler. Zatti concluded that a vector containing various load factors could best predict response time. His nonpreemptive algorithm specified an information policy similar to many above. The nodes kept a list of the other nodes that included their availability for CPU and I/O. A node was in one of four states: not available; CPU available; I/O available; and both available. All state changes were broadcast to the network. The load assessment policy was a set of thresholds for CPU and I/O loads. Zatti experimented with two methods of storing state information. A binary vector simply stated the availability of the resource while a real valued vector contained actual load levels for each resource. When a process required more resources than locally available, the node scanned the host list looking for an appropriate node. The recipient had the chance to migrate the new process if it could no longer accept it. A process could migrate a maximum number of times to prevent process thrashing. Zatti found that binary and real schemes worked comparably and, in general, his algorithm reduced response time for all classes of jobs.

3.6 Comparison of Algorithms

A complete comparison of the above algorithms is beyond the scope of this thesis, but some things do stand out. The information policies of most algorithms rely on network broadcasts to share information, although at least two [28] [25] note that the cost becomes prohibitive as the network grows. Alternatives to broadcasts become more attractive [16] [4] if the current trend of bigger and bigger networks continues.

Many algorithms use the length of the run queue (e.g., the Unix load average) in assessing the load. The alternatives to this include the per process CPU usage [10] [25] and the rate of I/O pages (along with the run queue) [45]. No evidence of how these different load factors compare has been presented.

The majority of algorithms presented are sender initiated. Taking the conclusions reached by Eager, Lazowska, and Zahorjan [15] this makes sense for networks that do not normally expect to be operating at high utilization.

Preemption appears to be a popular idea since almost all algorithms call for its use. It gives an algorithm the opportunity to exploit every imbalance in a systems load; this improves performance but also carrys the risk of instability. Nonpreemption seems to be used only when the host system does not offer preemption.

3.7 Summary

Load sharing assigns tasks to processors so as to reduce their response time. Although more powerful computers are continually becoming available, load sharing will remain a useful tool. The previous research in load sharing has investigated many of its fundamental aspects and some important points have become clear. Frequent use of network broadcasts incurs costs proportional to the size of the network. Even the simplest policies perform better than no load sharing at all, indicating that complex algorithms are not necessarily required. However, many other questions remain to be answered.

Chapter 4

Research on the Accuracy of Load Indicators

Load assessment is an important part of a load sharing algorithm. Incorrect or poor judgements undermine the effectiveness of any algorithm. My hypothesis was that an accurate indication of load should produce an accurate estimate of a job's expected response time. A migration policy would have an easy decision if given the expected response time of a job on each potential recipient.

Very little research has been published about the accuracy of various factors in indicating machine load. Many published load sharing algorithms used the number of tasks waiting for service as their load metric, but the evidence of the accuracy of this measure as a load indicator had not been conclusively presented. Zatti [45] did some work in this area. Personal experience with the Unix "load average", a time averaged value of the queue length, led me to believe that it did not always accurately measure the true load of the system. Zatti showed that the Unix load average did not correlate well for I/O bound jobs.

Thus the initial research goal was to study which load factors could accurately predict response time.

4.1 Design of the Experiment

To correlate load factors with response time, clearly one must measure both and then analyze the results. This was accomplished by recording various aspects of load in an operating system just before running a specific test (probe) job, and then measuring the response (elapsed) time of the test program. A log file received the values of the load factors and then the job's response time.

The experiment took place on Sun2 [5] workstations running the Eden system (see Section 2.2) on top of Sun Unix. The probe jobs were Objects running under a modified version of the Eden Kernel. The Kernel measured the various Unix load factors at five second intervals, which is the rate at which the Unix kernel updates these values. The values for the previous five intervals were written into a log file just before an Object began execution. Recording five intervals rather than just one allowed correlation of time averaged as well as instantaneous load.

Selection of the the load factors used in the experiment was based on my intuition about how well each measured load. The factors examined were:

- Utilization of
  - main memory
  - swap space
  - CPU (i.e., per cent idle)
- The number of
  - context switches
  - device interrupts,
  - page faults
  - swap faults

- The number of processes waiting for
    - the disk
    - for a page
    - in the run queue.

These measures are available from the Unix system by reading the device /dev/kmem. The code for this was adapted from Unix utilities that use this information (e.g., *ps*). Before and after a program ran, the probe job added a line with the current time into a log file. Postprocessing of this file gave the response time of the program.

There were three types of probe jobs: CPU intensive, I/O intensive, and balanced. The CPU intensive task calculated various mathematical functions such as square, square root, logarithm, etc.. The I/O intensive program sequentially read through four very large files while writing each page into a file consisting of a single page, overwriting the previous page on each write. The balanced process used the CE compiler to compile the Dispatcher module from the Eden library.

The experiment ran on workstations during their normal daily use. A command script controlled the experiment by executing the CPU, IO, and compile Objects sequentially, five minutes apart. This loop was repeated every 30 minutes over approximately an eight hour period. The logfile was then interpreted by a program that printed the type of program, its response time, and the load factors recorded before execution. This output was read by another program which organized the data by load factor. It then wrote the response time, the value of the load factor and program type into a file, one file per load factor. A typical line in the file for the CPU Utilization might be: `69 101 c`. This meant that the CPU was initially 69% idle before executing the CPU intensive job, which had a 101 second response time. These load factor files were then graphed as scatter plots using the Unix *graph* utility and displayed on the Sun. A total of 176 runs were recorded per type of probe job.

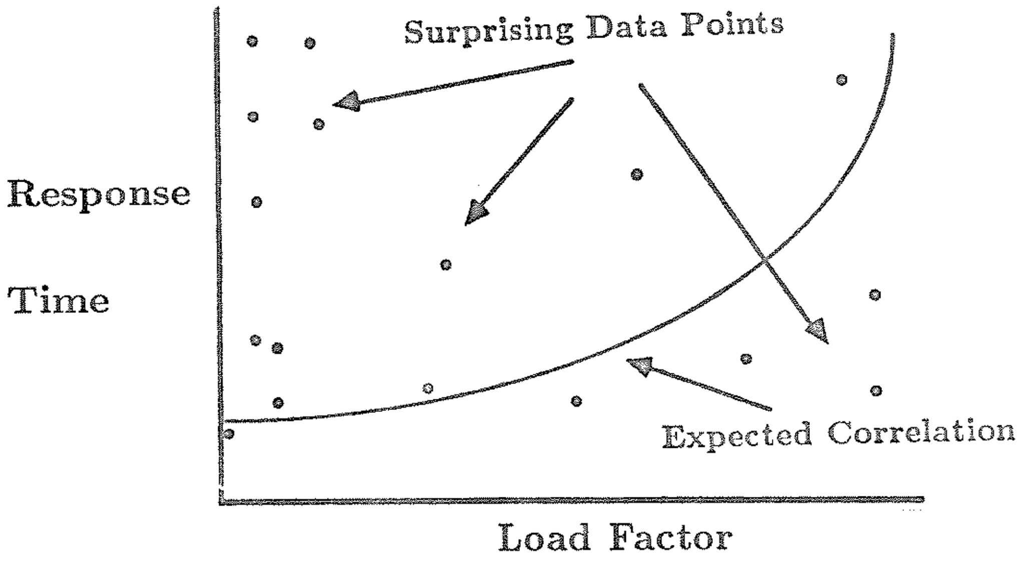

Figure 4.1: Expected Correlation and Surprising Results.

4.2 Unexpected Findings

The hypothesis that some combination of load factors would provide an accurate prediction of response time was not supported by the results of the experiment. As Figure 4.1 shows, rather than exhibiting a well defined relationship between load factor and response time, the data for both instantaneous and time averaged values of load factors did not display any obvious correlation with response time at all. This surprising result prompted further investigation into the experimental design and the workstation environment.

After considering both the design and the environment, two significant aspects of our workstation environment were clarified: workstations load is basically binary (idle or busy), and their source of work is a single person. The former occurs because our workstations are primarily used for program development. This consists of the cycle: edit a program, compile it, run the new version, and debug it. This cycle consumes resources in a very erratic manner. The CPU remains idle during editing, while compiling completely consumes all available cycles. Running and debugging a program require varying amounts of resources depending on the application, although debugging is usually interactive (i.e.,

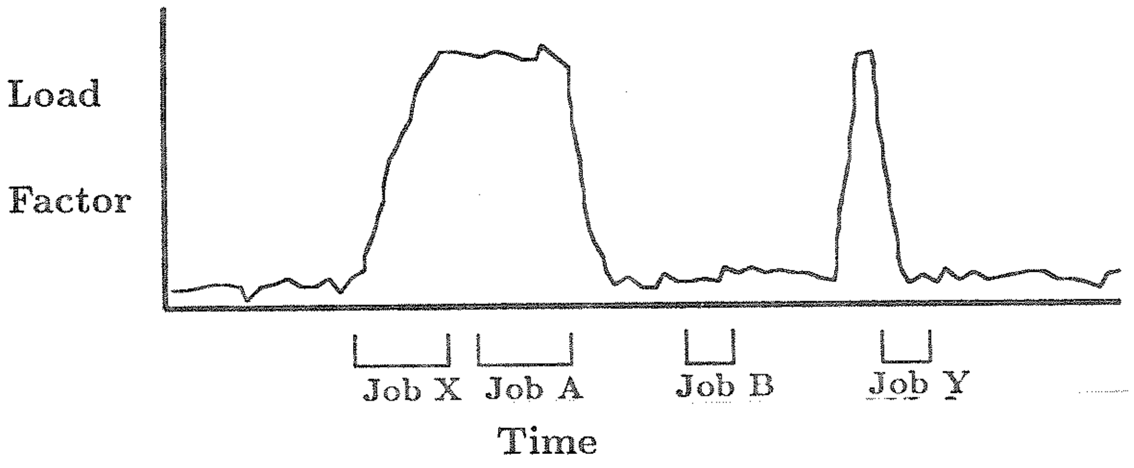

Figure 4.2: Effects of Bursty Workload on Response Times.

not CPU intensive). This usage pattern results in workstation being idle a large percentage of the time [42], but being swamped at other times as shown in Figure 4.2. Users of timesharing systems also exhibit this pattern but when their patterns are superimposed, the individual variations merge together resulting in a more stable load (see Figure 4.3). Since workstations only have a single user they always see the individual fluctuations.

Combining these two aspects of a workstation yields a *bursty* pattern for the workload and explains the unusual experimental results. Given the length of response time for the probe jobs, it is unlikely that a workstation remains at a constant level until a probe job finishes. Figure 4.2 illustrates this problem for four probe jobs. Jobs A and B experience a constant level of load and their response times correlate well with the initial level of load. Job A initially sees a light heavy load and its response time is long while Job A encounters a light load and correspondingly takes less time to execute. The opposite occurs during jobs X and Y. The workstation is initially idle just before Job X begins but becomes heavily loaded moments later. The result is that job X takes longer to finish than would be expected from the initial level of load.

This explanation was verified by artificially raising the load to a constant level by running numerous jobs in the background on a continuous basis. Under this constant

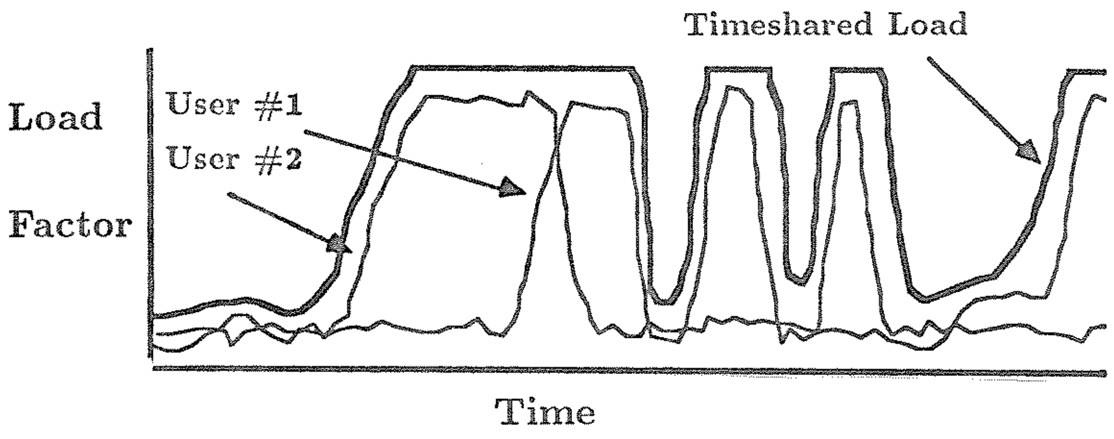

Figure 4.3: Merging of Two Individual Workloads Under Timesharing.

workload, some of the load factors exhibited the expected correlation. Work by Zatti [45] offered further support of this explanation. His graphs indicated that at light load levels program response time displayed a large variance. Better correlation occurred only at relatively high levels of load.

Due to the bursty workload, future load cannot be predicted with much accuracy. The lesson to be learned is that rather than attempting to make highly accurate predictions of response time, one should simply look for other workstations with a significantly lower level of activity. The inability to accurately predict future load continues but the problems that it causes are mitigated by the ability to load share. Consider what happens when an idle workstation, after receiving work from other nodes, begins producing additional work itself. Under load sharing, there are attempts to migrate some of the work to keep the overall load level down. This helps to keep the load on each workstation closer to the level predicted. Could load sharing have solved this problem in the above experiment? It could have helped to some extent, but not sufficiently well to allow accurate predictions of response time.

4.3 Summary

My hypothesis failed not because the load level of a workstation can not be measured, but because it assumed that using the current load one could predict response time. This assumption proved invalid due to the bursty nature of workstation workload. Timesharing systems suffer less from this problem since the bursty behavior from individual users combines to give a more stable load. The lesson learned is that it makes more sense to look for *gross* differences in load rather than trying to accurately predict response time for each node.

Chapter 5

A New Load Sharing Algorithm

The experimental results of the previous chapter changed the direction of my research. Instead of looking at the ability of different load indices to predict response time, I shifted my focus to examining the ways in which the workstation environment affects the design of a load sharing algorithm. This chapter is devoted to outlining these effects, explaining the design and implementation of the algorithm, describing the experiment to test the algorithm's effectiveness, and interpreting the experimental results.

5.1 The Effects of Workstations on Load Sharing

The experiment described in Chapter 4 highlighted a difference between a workstation environment and a collection of networked timesharing systems. This difference prompted me to consider what other distinctions exist between the two types of systems. Two other points soon emerged that caused a fundamental rethinking of how an effective load sharing algorithm should be constructed.

5.1.1 The Goal of a Workstation

The first difference that came to light involved the purpose of each system. The primary goal of a workstation is to provide a responsive, highly interactive work environment; users of workstations expect and demand immediate interactive response. In contrast, a timesharing system attempts to supply a moderate number of computing cycles to a large number of users; it does not offer the type of CPU intensive interfaces possible with workstations. Since workstations strive to reduce interactive response, it seemed logical that load sharing in this environment should share this goal.

This new goal of reducing interactive response time conflicts, however, with the traditional goal of load sharing. Traditionally, load sharing sought to minimize the *average* response time per node or globally, regardless of the type of the work being done[44].

Such a point of view disregards research on users perceptions [36]. Users can perceive relatively small delays in interactive response, such as a slow or erratic repaint of a page in a screen editor. They find these delays very annoying and irritating; more importantly, the delays adversely affect user productivity. One reason why an interactive delay leaves a user frustrated is that people are more sensitive to delays as the expected response time grows shorter. Since the expected response time for interactive processing (e.g., moving a cursor in an editor) tends to be much shorter than batch processing (e.g., compiling a program), interactive response is more susceptible to negative user perceptions. During program development, a user spends a large amount of time using interactive programs such as screen editors, symbolic debuggers, electronic mail handlers, etc.. Minimizing the response time for these programs should increase programmer productivity by reducing interactive response time and frustration.

Programs encounter delays when they compete with other programs for resources. But since workstations have a single user, the only competitors of an interactive program must be noninteractive programs (e.g., compilations and text formatting). Thus, load sharing should favor interactive programs over noninteractive programs in order to minimize interactive response. Considering how people react to delays, it makes sense to reduce the interactive delays even at the expensive of overall longer batch delays. A one time 20% to 50% increase in the time for a noninteractive program such as a compilation will not be as annoying as many shorter interactive delays. Note that while people are less susceptible to batch delays, they will not accept them if the delays become unreasonably long. Giving interactive programs higher priority, however, still enables the user to run background work concurrently while not degrading interactive response time.

5.1.2 Workstation Ownership

The second important difference between workstations and timesharing systems is how users of each system perceive ownership of computing resources. A workstation has a single user who often considers the resources to be personal property. This contrasts to the more communal attitude required when using a timesharing system. These users may grumble about not getting enough of the resources, but they do not consider the machine to be their personal fief. Consequently, load sharing must take into account the impact of migrated work on a local user's response time. If users perceive that outsiders are consuming too much of their *personal* workstation, they will rebel against load sharing. Thus, locally generated work should have a priority over foreign generated work.

5.1.3 Summary

Three unique characteristics of the workstation environment affected the design of my load sharing algorithm. Workstations have a bursty workload that prevents using fine grained indicators of current load to predict response times. As a result, a load sharing algorithm must instead exploit *gross* load differences between nodes. Workstations seek to provide highly interactive computing so that the algorithm should emphasize reducing interactive response time while still offering reasonable batch response. Finally, the algorithm should respect the concept of ownership of personal workstations.

5.2 Description of the Load Sharing Algorithm

This algorithm is based on one published by Eager, Lazowksa, and Zahorjan [16] called "Random with Probe". A number of significant modifications to the original algorithm were incorporated. Eight separate load factors, rather than just the length of the run queue length, are used. The use of equivalence classes simplifies the comparison of load between workstations. Introducing feedback into the sender initiated component while adding a receiver initiated component enhances the algorithm's performance and stability at high levels of utilization. Yet the algorithm retains the simplicity and low cost of the original algorithm.

5.2.1 Using Equivalence Classes

Assessing levels of workstation load in the migration policy can be a difficult task. Comparing raw data such as CPU utilization and swap rate will not always produce an obvious choice. The use of equivalence classes to classify the load level of a node simplifies this problem. The algorithm also becomes more modular so that changing the tests for inclusion into a class does not affect other aspects of the algorithm. The classes are ranked by the "<" relation with its usual properties. A workstation's load assessment policy considers its load and places it into a class. The migration policy then uses this class for purposes of comparison with other nodes when locating a host for migration.

Deciding how many classes suffice becomes the next question. This involves examining the amount of activity on a node and whether or not it has an active user. Analysis of the previous experiment indicated that workstations were either idle or busy implying that only two levels of activity are needed. But another load problem sometimes occurs in daily experience. Operating systems, such as Unix, provide a fixed amount of resources. For example, there is a maximum number of processes allowed or a finite amount of disk space available for swapping. Attempts to exceed these limits result in the death of the offending process. The importance of avoiding such a drastic response argues for the addition of a third category. This gives three levels of load: idle, busy, and full.

On a workstation without an active user, the load must be, by definition, noninteractive. With a goal of reducing interactive response time, it seems sensible to prefer using workstations without active users to ones with them. Given that two nodes have equivalent loads but only one has an active user, selecting the workstation without the user reduces the impact on the active user's response time. This distinction results in two operating modes: interactive and noninteractive.

The combination of the three load levels and two operating modes produces five load Classes. *Quiet* and *Quiet Noninteractive* means a node can comfortably handle the current level of activity. *Busy* and *Busy Noninteractive* indicates more work exists than resources. The two noninteractive Classes indicate that the workstation does not have an active user. When a node is *Full*, crucial resources are close to depletion. Note that the state of interaction is irrelevant for the Full class because a node in such a situation should attempt to migrate its tasks and refuse any foreign ones regardless of the presence of a user. Classes are ordered: Quiet Noninteractive < Quiet < Busy NonInteractive < Busy < Full.

5.2.2 The Information Policy

The load factors selected for the information policy came out of experimental work of Chapter 4 and continued observations during the experiments that followed. The final set consisted of:

- Utilization of the
  - CPU (percent idle)
  - Swap area
  - Main memory
- Number of
  - Context switches
  - Swapped pages
  - Empty process slots
- Length of the run queue
- The keyboard idle time.

These factors are gathered after specific time intervals or *ticks*[1].

Unlike many other algorithms, the nodes do not exchange state information on a regular basis, but only in the process of migration (see Section 5.2.4) or when a node has been idle for a long time. The latter involves each node checking for the number of consecutive ticks it has been Quiet or Quiet Noninteractive. If the number of ticks exceeds `ConsecQuietTime` [2] ticks, an *IamQuiet* message is sent to four randomly selected nodes. The recipient workstations add the node to a list of idle nodes called the *Quiet List*. (The Quiet List contains the names of workstations known to have been Quiet or Quiet Noninteractive. Its use is described in Section 5.2.4.) This mechanism allows idle nodes to seek out work and adds a Receiver Initiated component to the algorithm. This component improves the algorithm's performance at higher levels of utilization. The message is repeated after every `ConsecQuietResend` ticks. The flag indicating that an IamQuiet message should be sent is reset after `ConsecBusyTicks` ticks. This prevents messages going out after ephemeral Busy periods.

5.2.3 The Load Assessment Policy

After each tick, every node obtains the latest values for its own load factors and reevaluates its Class. If the load during the tick satisfies the criteria described below, the tick is designated as a *spike*. The activity level of the node is found by looking at the recent spike history.

A spike is found by checking four of the load factors: the length of the run queue; the number of context switches; the number of swapped pages and the CPU utilization. These four were used because they showed good correlation with response time at higher load levels during the previous experiment. Each load factor has a corresponding threshold. For example, the threshold for the number of context switches (`ContextSwitchThreshold`) is

[1]Each tick is five seconds, since this is the rate at which the Unix kernel updates this information.

[2]Algorithm parameters are printed in typewriter font. A list of their values during in the experiment is provided in Appendix A.

50 . If the number of factors exceeding their threshold values totals `SpikeThreshold` or greater, the tick is recorded as a spike in a history kept of the previous ticks. The spike history maintains `TickHistory` entries.

The workstation activity level is next determined. The activity level is labeled Busy if any of four criteria are met:

- The previous tick is a spike. This detects a potential increase in node activity.
- The spike history contains more than `SpikeRatio` spikes. This permits an idle node that has been continually Busy a small period of reduced activity. It also prevents ephemeral light loads from fooling the node into believing that it should accept more work.
- Memory utilization exceeds `MemThreshold`, implying that swapping is soon expected.
- A new process commenced during this tick. This prevents a node from accepting too many processes before they consume enough of the resources to alter the node's Class.

A node is Full when any of the following occur:

- Utilization of swap space exceeded `SwapFull`.
- Less than `ProcSlotsFull` process slots are available.
- The length of the run queue is more than `RunQFull`.

The first two factors are observed to be the limits most likely to abort a process. The last one is included from an intuitive feeling that there should be an limit to the number of jobs that a workstation should ever be attempting to execute. It differs from the first two in that it detects a situation where the process will run very slowly rather than aborting. If none of these tests are satisfied, activity level is declared Quiet.

The Class is finally determined by the inspecting the interactive state. If the keyboard or mouse had not been used within `KeyboardIdleTime` minutes, the workstation mode becomes NonInteractive. Combining the activity level and interactive mode produces the Class unless the activity level is Full, in which case it remains Full.

With the introduction of Classes, the decision about when to migrate becomes straightforward. If the system offers only nonpreemption of processes, the workstation checks its Class when creating a new process. If the Class is Quiet NonInteractive or Quiet, the creation occurs locally. Otherwise the node attempts to find another host with a more preferable Class. If preemption is available, much more flexible policies could be considered. This is discussed in Section 6.1.

5.2.4 The Migration Policy

Since the information policy does not produce data on the state of other workstations, the migration policy must rely on querying potential recipients at migration time. The migration policy selects a small number of nodes (`ProbeLimit`) to be sent query (probe) messages. These potential nodes are chosen by first using those found in the Quiet List (described later in this section) and then randomly selecting as many more nodes as needed. The probe message, consisting of the sender's Class, is sent to each of the probed workstations using multicast, or by sending each message asynchronously. Probed machines ignore the query message if their Class is greater than or equal to the sender's Class. The work is sent to the first node to reply on the assumption that the least loaded node will reply first [28] . If no workstation replies to the query, migration fails and the process is created locally. One could attempt another round of selections and probes but research has shown that this will not generally improve the results [16].

The migration policy uses a feedback mechanism to limit the probing of other workstations to times when a success query is likely. This mechanism adds stability and efficiency to the algorithm when most other nodes are also Busy. Each node uses a counter to keep track of the number of failures, i.e., no node replied to the query messages. Each failure increments the counter while each success decrements it. The counter is checked before attempting a probe of other nodes. When it exceeds `MaxFail`, the node does not attempt to contact other nodes but just creates the job locally. The node sets a timeout of length `ResumeProbeTMO` which decrements the counter when it expires. This permits the node to periodically attempt to load share. The counter is also decremented whenever an IamQuiet message is received.

The Quiet List contains the names of workstations known to have been Quiet or Quiet Noninteractive. When a Quiet or Quiet Noninteractive node replies to a probe message, it is added to the list. Nodes of other Classes and those that do not reply are removed from the list.

5.2.5 Priority of Interactive and Foreign Work

To help fulfill the goal of reducing the response time of interactive programs, work is divided into Interactive and Everything Else. Interactive work executes at a higher processor priority than Everything Else. For example, a text editor would have higher priority than a compilation. If a user simply does nothing while waiting for a compilation to finish, then the compilation's reduced priority won't have any effect. Because many interactive programs access the user's display very frequently, local execution of interactive programs is very desirable. The communications overhead for these programs could be prohibitively expensive if they execute remotely. Thus, Interactive work is only migrated when a node is Full. This view could change as extremely efficient graphical protocols become available [23].

Deciding whether a program is Interactive or not remains subjective. Some choices such as animation, text editing, debuggers, and iconic interfaces stand out among the many types of Interactive programs. Compilations, text processing, and VLSI design rule checkers could clearly be excluded. Program characterization is further discussed in Section 6.2.

Reducing the negative impact of migrated work on a user of a personal workstation is also important. Load sharing will not be accepted by users if unacceptable local delays occur. To avoid this, the algorithm executes work received from other nodes at a lower processor priority so that the local user does not perceive an increased delay for local work. Consequently, all work is classified as either Local or Foreign. All subsequently created work would retain its parent's heritage.

Combining these two concepts produces four levels of processor priority in the order: Local Interactive > Foreign Interactive > Local Noninteractive > Foreign Noninteractive.

5.3 Description of the Experiment

This load sharing policy was tested on the Eden system. The experiment was designed to answer the following questions. Does this algorithm work at all? Does it reduce the response time for interactive programs? What are the effects of lowering the priority of noninteractive and foreign programs? What are the costs of load sharing?

5.3.1 The Experimental Environment

Two environments were combined in the experiment. A network of eight diskless Sun2 workstations running 4.2 Berkeley Unix [35] connected by a 10 MB Ethernet [30] provided the physical computing facilities. Each Sun2 had 4Mb of main memory but lacked a local swapping disk. The Suns were configured with two other Sun2 nodes acting as disk servers for clusters of four workstations. Sun's distributed file system, NFS, was not installed.

The Eden operating system (see Section 2.2) provided the distributed operating system capabilities for the experiment since Unix does not provide the necessary distribution primitives. Eden provides *CreateAt* and *ActivateAt* primitives that allow the Eden client to specify a destination for an Object. Eden did not provide automatic load sharing so that some modifications to the Eden Kernel were required. New message types and handlers for Probe, Probe Reply, and IamQuiet messages were added. Code for accessing the Unix kernel's load data as well as a random node selection routine was required. The addition of 800 lines of code increased the size of the Eden Kernel by 9%.

Eden does not provide preemption of Objects. As a result, migrations only occurred when an Object was being created by the Kernel.

5.3.2 Experimental Controls

To minimize experimental error, no other users were allowed access during the experiment. Furthermore, file servers were not used as hosts during the experiment. This is due to the demand on the disk server CPU when satisfying a disk request from a diskless workstation [24]. Not using the disk servers as hosts prevented intermittent delays for diskless I/O due to congestion at the server CPU. Identical Suns were used to minimize processor or memory differences affecting measured response time. Note that this is not a requirement of the load sharing algorithm but simplified analyzing the results. Machines of differing power and size can be easily accommodated by changing the values of the algorithm's parameters.

5.3.3 Description of the Workload

My interest in program development environments prompted me to use a workload that reflected those requirements. Eden currently does not provide full program development facilities. Consequently, I used Unix programs to simulate this workload. Since Unix does not offer distribution primitives, Eden became the distribution agent for these Unix programs. While the experimental workload provides only a single data point with respect to all possible workloads, it was developed after discussion and review with the members of the local user community and presents a typical interaction in that community.

Types of Programs

The simulated program development cycle workload consists of reading mail, editing a program with a screen editor, compiling the program, running the program, and debugging it with a symbolic debugger. Using the standard Unix mail program, *mail* [37], four messages are read and replies are sent for two. The *emacs* screen editor [38] edits the sources for *more*, a Unix file display utility. Compilation uses the *cc* C compiler while the symbolic debugger *dbx* acts as the debugging agent.

Running each program involves using aspects of both Eden and Unix. The workload simulation Object is created on the node selected by the load sharing algorithm. It then spawns the Unix program to be executed. The interactive programs, such as *emacs*, expect input from a user typing at the keyboard. The Object acts as the user by sending characters to the interactive programs so that they are received as keyboard input. This is accomplished by redirecting the input of the interactive program to receive output from the Object using a mechanism called a pseudo-teletype or pty. The source of the keyboard input comes from a script file read by the Object. It reads the script and sends the keystrokes at the rate of 5 per second to the interactive program via one of the ptys.

As previously mentioned, a single user presents a bursty workload. This workload involves batch type programs, such as a compiler, and interactive programs, such as an editor. Interactive programs, while not being CPU intensive, also are subjected to a user "think time". During this is period of time the user does not submit new work and is not waiting for the completion of previous work. To simulate this, the scripts specify user think time and the Object waits for this amount of time before sending the next set of input.

One potential problem is the interactive program falling behind while reading the keyboard input. Suppose that the editor cannot keep up with its input and the input buffer begins to fill up. When the Object stops for the specified amount of think time, the editor will still process its input as fast as it can until the input buffer is empty. Because the response time for the editor is found by subtracting the timestamp taken before it starts from one taken after it completes, the actual amount of delay would not be accurately measured. To avoid this, an explicit "think" command was added to each interactive program. The Object sends the think command along with its duration to the interactive program before it "thinks" for itself. In the above example, the editor will read the input including the think command. It then stops reading for the specified amount of time, even after the Object has continued to send it input. "Think" was implemented using the Unix *sleep* library routine. Extensive testing showed sleep as very accurate with no more than 3% error being observed for the times involved.

Levels of Load

The initial research showed that the load due to the program development workload did not overwhelm the workstations. However, background processes and modern window managers facilitate adding noninteractive work to a node. This situation is represented by creating a background workload. Compilations with *cc* and *cec*, the Concurrent Euclid compiler, along with text formatting using *latex* [22] provide noninteractive load. These programs run sequentially in a loop every one or two minutes. This workload operates concurrently with the program development workload.

Three levels of activity were tested. A Light load consisted only of the program development workload. A Medium load added one background workload while a Heavy load used two background workloads. The background workloads ran concurrently with the interactive workload.

5.3.4 Design of the Experiment

To answer the questions posed in Section 5.3 the experiment varied the dependent variables so as to isolate their effects. The dependent variables of the experiment were:

- Whether or not load sharing was used
- Utilization level (the number of nodes generating work under load sharing).
- The level of load activity on each node
- Distinguishing between Interactive and NonInteractive program
- Distinguishing between Local and Foreign programs when load sharing

When load sharing was not in use, distinguishing between Foreign and Local work as well as varying the number of nodes generating work became irrelevant. The four values for the utilization levels under load sharing were one, two, four, and six active nodes out of a total eight workstations. Three levels of load activity (detailed in Section 5.3.3) were used. The values of the other variables were binary. All other aspects of the experiment remained constant (e.g., the type of computer).

The primary metric for the experiment was the response time for the program development workload components. Response time was calculated by subtracting a timestamp taken before execution from one taken afterwards. Since the user think time was a constant value for each type of program, it was subtracted from the measured response time leaving only the response time from processing. The throughput and response time of the batch programs in the background workload were also recorded. This allowed for analysis of the effects of the various aspects of the load sharing algorithm on batch programs.

Auxiliary Programs

A number of auxiliary programs and command scripts were essential in running the experiment. A command script, called TestMonitor, controlled each run of the experiment. After selecting the experimental values, I then ran TestMonitor and supplied it these values on the command line. TestMonitor first checked the experimental values for consistency and then ensured that only the experimental version of the Eden Kernel was running on the network by first killing off all the Eden Kernels and then starting the experimental versions. Subsequently, TestMonitor set the kernel parameters for load sharing and Foreign program priority on each experimental node. It then executed another command script, called RunWindows, on each of the nodes to run the Sun window manager using three windows for program output and a console window for system messages. In addition, RunWindows started a graphical monitor (described below) of the load factors being measured by the algorithm's information policy. TestMonitor finally created an Object, called NodeWork, on each node that was to be a source of work and waited for all of them to finish. It cleaned up by removing the window managers and reinstalling the normal Eden Kernels in the network.

The NodeWork Object created another Object, called DoProgram, for each Unix program to be run. Each part of the workload had a concurrent process inside the NodeWork Object. Thus, when a NodeWork Object was running the Heavy workload, three concurrent processes were active. Without load sharing, DoProgram Objects executed locally; with load sharing, they could execute on any node. After creating a DoProgram Object, NodeWork sent it a *Start* invocation containing all the dependent variables and the type of Unix program to be begun. A *Status* invocation immediately followed, which blocked this concurrent process in NodeWork until the DoProgram Object replied with the program's response time. NodeWork wrote this response time into a logfile using a monitor to prevent simultaneous writes by other concurrent processes.

A graphical monitor of the load factors proved valuable in determining what was going on. Changes were made to the Sun utility *perfmon* to look at different load factors as well as keeping a history of the output in a file. The new program, *myperfmon*, ran on each node to record the effects of a load level on a workstation. To interpret the history file, more changes were made to *perfmon* so that instead of actively monitoring current load, it displayed the input from the *myperfmon* history file. This program, *showperf*, read input from the history file allowing one to roll forwards and back through the data. Figure 5.1 is an example of *showperf* output. The horizontal lines with the short vertical lines is the time line and each short vertical line is a tick. The other horizontal lines are thresholds for the various load factors. This figure shows the node experiencing a series of heavy activity and node's Class responding. Note that for the measurement of MyClass, Quiet Noninteractive corresponds to 0 and Full to 4.

Tuning of the numerous parameters in the algorithm required running the experiment many times in various different modes. A significant number of changes and additions to

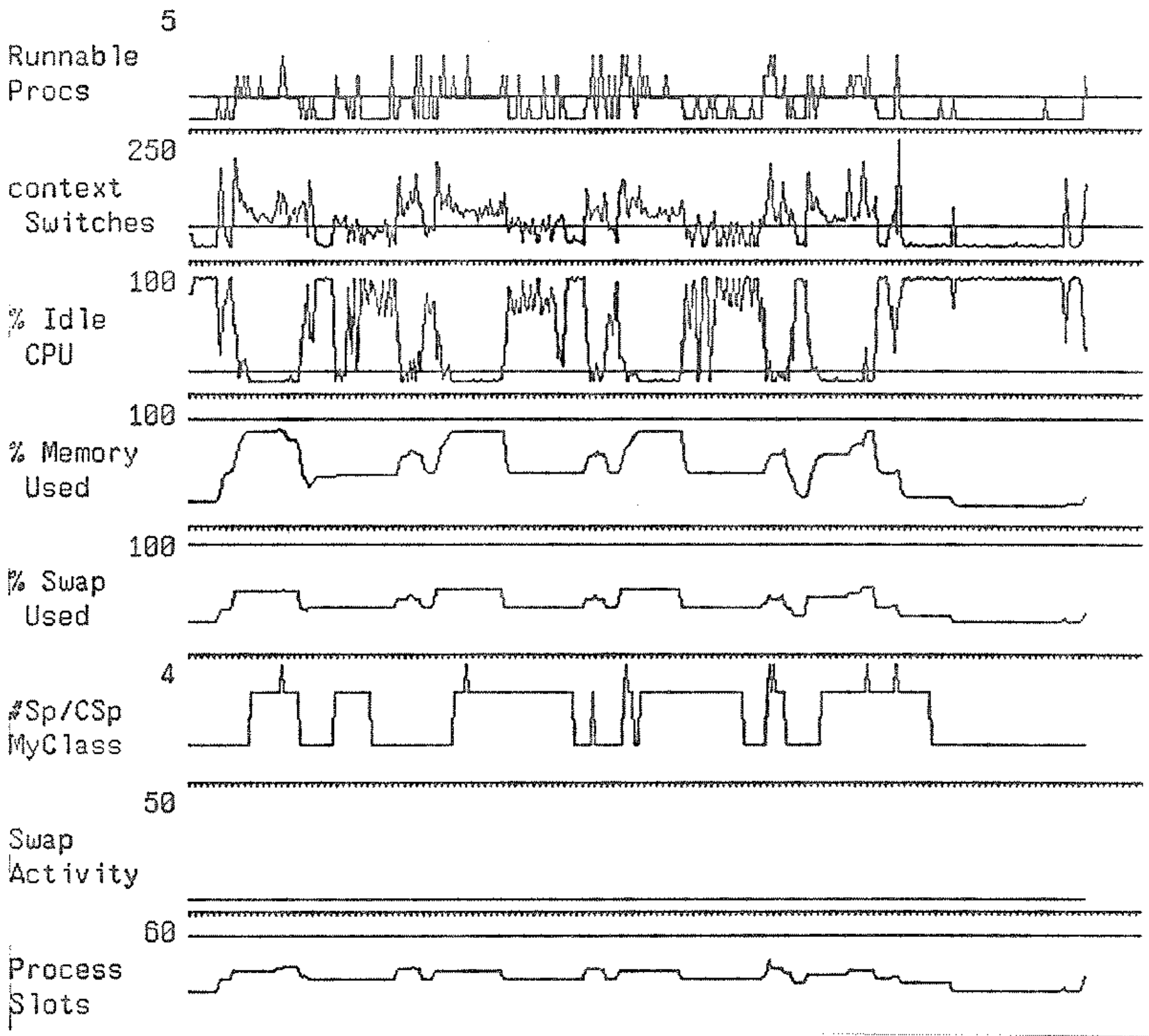

Figure 5.1: Sample Output From the Showperf Graphical Monitor.

the algorithm resulted from observations during this pretest phase. Measuring the swap rate was added to the load factors after it became clear that almost any swapping resulted in very poor response. The Full Class was included after having experiments fail due to Objects exceeding resources on a workstation. Two of the four tests for detecting the Busy activity level were added after noticing the problems from a node responding too slowly to an increase of work. Using *myperfmon* and *showperf* eased the task of deciphering the causes of poor response or strange load sharing decisions.

The logfiles from the NodeWork Objects containing the results of the each experimental run were located in a well known directory on each participating node. After running the experiments, all the log files were moved to a single node for analysis. A program called *stat* read in the contents of all the log files. Certain consistency checks were performed on the data to ensure its validity. For example, results from an incomplete experiment due to the failure of a workstation would not be used. *Stat* then produced a table of results with mean response time and standard deviation for interactive programs and the mean response time and mean throughput for batch programs.

5.3.5 Running the Experiment

The values of the dependent variables were chosen so as to provide the most information in the fewest number of experimental runs. Since batch programs were not vying with interactive programs for the processor under Light loads, load sharing did not affect these interactive response times. Thus, the results from all the Light runs, regardless of the values of the other dependent variables, should be basically the same. Consequently, the experiment was run with a Light load only a few times in order to find the base response times for the interactive programs and to verify that the experiment was working properly. The values for the other runs were selected so that effects from the dependent variables would be most visible.

5.4 Results of the Experiment

Load Sharing and Interactive Priority Reduce Response Time

The results from the experiment clearly show that using load sharing and running interactive programs at higher process priority both significantly reduce interactive response time. Table 5.1 presents a small portion of the results that best illustrate the effects of using load sharing and higher interactive priority. The numbers given in the table indicate the *increase* in response time over a base (i.e., nonloaded) response time for the editing session. The worst response time occurred with the combination of not having load sharing and not favoring interactive programs. Using either load sharing or a higher interactive priority produced similar improvements while using them together gave the best response. It is interesting to note that load sharing also improved the response time for batch programs. This is surprising since improving batch response time was not a design goal and every choice in the algorithm favored interactive programs. The reason for the batch improvement is that load sharing uses of all of the processors. Batch programs are moved to idle processors so that even batch programs receive more cycles since there are now more cycles to go around. This shows that load sharing, even at high levels of system utilization, more efficiently uses available resources than not load sharing.

Helping Local Processes is Inconclusive

The results are inconclusive regarding the hypothesis that running Local processes at a higher priority decreases the impact on local interactive response time. One would expect to find improvement only at high levels of utilization because only then do processes migrate to other nodes in large numbers. But no significant effects were seen. However, this experiment does not conclusively rule out that running Foreign tasks at a lower priority benefits local interactive response due to a flaw in the experimental design. In this experiment, each active node produced work at the same rate with the effect that

Increase of Response Time Over An Unloaded Response With 6 of 8 Nodes Producing a Heavy Activity Load				
Increase for:	Editing		Compiling	
What Type	No L.S.	L.S.	No L.S.	L.S.
No Interactive Priority	113%	57%	125%	85%
Interactive Priority	52%	19%	125%	85%
Important Result:	L.S. & Prio help equally well. Best when together.		Only L.S. helps batch response.	

Table 5.1: Sample Results of Using Load Sharing and/or Processor Priority to Reduce Interactive Response Time for 6 of 8 Nodes Running 1 Interactive Job Stream and 2 Batch Job Streams.

each node was coping with locally generated noninteractive programs as well as migrated work. As a result, no difference should be expected in the interactive response times on each node. A more effective test would run nodes at varying levels of activity, with and without higher priority for Local programs. This permits a comparison between nodes running at high and low activity levels and highlights any effects of migrating Foreign tasks. If the hypothesis is correct, then the response times on lightly loaded workstations should remain stable as the number of other busy workstations increases.

Selected Load Factors Work Well

The selected load factors worked well. More work is needed to identify the minimal set of factors required and to see in what context each is important. For example, for a diskless workstation with insufficient main memory, the swap rate is a vital sign. The degree to which these factors are operating system or architecture dependent remains to be seen.

Two interesting lessons were learned regarding load factors. The first is that load factors are not always independent. Using *myperfmon* to graphically display the load factor output revealed that CPU Utilization and Context Switches mirrored each other and either one of these would suffice (see Figure 5.1). The second is that CPU Utilization acts as a poor load factor when used alone because it does not give any indication of what percentage of the CPU would be available to a new process. This value can be determined in conjunction with the length of the run queue. For example, when the CPU utilization registers 100% busy, a new process can receive up to 50% or as little as 10% of the CPU cycles whether one or nine other processes respectively also require service.

The Problem with Nonpreemption

A key problem of nonpreemption emerged during the course of the experiment. The problem is that nonpreemption does not permit an algorithm to recover from making a poor load sharing decision; a mistake can be corrected only by waiting for processes to finish executing. One example of such an error occurs when too many processes are accepted by a node before any of them use enough resources to have the node classified as Busy. The root of the problem is that the algorithm must be able to accurately predict short term load levels. This problem can be eased by using heuristics to predict potential increases in a workstation's load. Since accurate predictions of load are not possible, the heuristics must be paranoid and always correctly predict a load increase even at the expense of many false alarms.

Costs are Affordable

The costs of load sharing appear very affordable. The time for a successful probe is 61 milliseconds. Each probe involves two Unix IPC messages which require about 18 milliseconds or 30% of this total. Other systems have reported IPC times of 1 millisecond [12] thus this total cost could easily be reduced by one third. Even without this reduction, the time required is comparable to that needed to create a Unix process. One expects that if the load sharing policy were part of the operating system rather than running as an application this cost would drop significantly.

The price for failing to find a node equals the length of the timeout for the IPC message. During this experiment, the timeout was three seconds but this value could be reduced to twice the expected length of time for a successful probe. In any case, this relatively rare event still is a fraction of the execution time for most processes worth migrating.

The cost of measuring the load factors appears insignificant. For this implementation, reading the Unix load factors for the information policy took 81 milliseconds every five minutes. This amount could be reduced to nearly zero if load sharing were part of the operating system and the needed measurements kept as part of normal operations. For example, the Unix kernel currently does not keep a count of the number of processes in use. During the experiment, the Eden Kernel read the entire process table to calculate this simple value.

5.5 Conclusions

The conclusions of this thesis are clear. Load sharing is possible, effective, stable and affordable in a workstation environment using a relatively simple algorithm. An important point was the extent that the environment affected the implementation of the algorithm. This point stresses the need to analyze the target environment for its unique characteristics before considering the implementation. Each environment offers its share of advantages and problems that must be understood in the context of solving a problem. In this case, the goal of load sharing was modified to stress reducing interactive response time in light of the similar purpose of workstations to provide highly interactive computing. In addition, the bursty workstation workload altered the way the algorithm was implemented.

The results clearly show that load sharing and favoring interactive programs improve interactive response time to about the same extent while combining them provides the greatest improvement. One somewhat surprising difference between the two changes is that load sharing also improves batch response time. This point underlines the fact that load sharing more efficiently uses available resources.

There are other interesting results from the experiments. Load factors should be used to indicate general levels of load and not be expected to yield precise results. This is a consequence of the bursty workloads experienced in the workstation environment; multi-user environments might not suffer from this problem. The use of equivalence classes solves the problem of comparing numerous load factors from different nodes and increase the modularity of the algorithm. The inability of nonpreemption to correct mistakes became apparent as a critical problem during the course of the experiment. When using nonpreemption, nodes must quickly respond to increases in load, even at the expense of numerous short-lived Busy peaks. Overall, striving for the simple yet useful seems to work extremely well.

The algorithm presented is easy to implement since it does not require special protocols or unusual information. It provides stability at all load levels by including a hybrid of sender and receiver initiated components and feedback for high utilization levels. Because it does not maintain state information on other nodes it avoids problems with stale information. Lacking the need to broadcast information, it can easily scale to any size network. Finally, it provides tunable parameters for flexibility in light of changing systems capabilities.

Chapter 6

Further Work

6.1 Future Algorithm Research

Many aspects of these load sharing results deserve further attention. How many classes should be used? Should some be added or deleted? Should the load level be a combination of activity level and keyboard idle time or should they be considered separately? What are the best combination of load factors? Should thresholds be adjusted dynamically? The experiment should be run again with differing sizes of the Probe Limit to see its effects. Clearly much work on tuning the numerous parameters would be required before using this algorithm in a production system. The load assessment policy and Class selection criteria also merit further study.

If process preemption were available, how would the algorithm be changed to take advantage of this? A number of possibilities arise. A node could attempt to migrate work whenever its Class increases. Selecting the process to migrate should use research done on process runtimes and their eventual requirements [25]. Thresholds could be changed so that a node did not go to Busy at the first hint of work increase. Quiet nodes could request work with their *IamQuiet* status messages.

6.2 Program Characterization

The question of what work to migrate always arises. Some programs should never be migrated, either due to their nature or the short time required to execute. One example of the former informs the user of the name or IP address of the user's machine. Other programs such as printing the date or listing the contents of a directory simply take an insubstantial amount of time to execute. If a node knew the expected requirements of a program, it could decide whether to bother load sharing or not. Characterizing a program has been done in Maitrd [6], Locus [11], and other systems to a limited degree. But a more general mechanism is needed.

One possibility is registering programs with a characterization service (analogous to a naming service). Complete characterization would not be needed; rather general guidelines would suffice. Important factors include when to migrate (e.g. always, never, after N seconds of CPU usage), whether it is highly interactive, required resources (e.g. array processor), etc..

Many methods for entering or changing a classification would be needed. A system administrator could perform it for well known programs such as formatters, compilers, editors and debuggers. Another would allow users to register their own programs. This could be done directly with the classification server or using language support. Authors would include a hint to the language translator giving the program's migration requirements, level of interaction, etc.. Users should be able to override a default except when it would cause an error. For example, a user's request for migration of a specific program would be overridden if the program could only run locally or if no available nodes were found. Before executing a program, the operating system would query the service about the program's class and take into account any other requests. One problem with this method is detecting and fixing incorrect classifications.

Another method looks to the history of a program to decide what it will do in the future. The operating system would gather statistics on programs after each execution and update the program's history. The location of this history could be with a classification service, an auxiliary file, or in a header within the executable itself. The problem with this method is deciding what statistics are important and drawing the correct conclusions from them.

6.3 Load Sharing With Computation Servers

If workstations shared their network with main frames computers then another type of load sharing could be used. As mentioned in Section 3.5.2, two interesting aspects of general computing are that most programs have a very short lifetime and a small fraction of all programs consume the majority of processing cycles. Using these characteristics, a very simple load sharing algorithm is possible.

The load sharing algorithm only intervenes after a program has spent a certain amount of time on a workstation (e.g., 3 seconds) and exceeds some resource consumption to time alive ratio (e.g., 75 % busy). This solves the problem of load sharing short-lived processes noted in the previous section. These programs are then sent to the main frames, which act as dedicated computation servers. This simple algorithm has many attractive properties and deserve further research as the cost of hardware continues to drop.

Bibliography

[1] James E. Allchin and Martin S. McKendry. Synchronization and recovery of actions. *Proceedings of the 2nd Annual ACM Symposium of Distributed Computing*, 31–44, August 1983.

[2] G. Almes, A. Black, E. Lazowska, and J. Noe. The eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11:43–59, 1985.

[3] Guy T. Almes and Cara L. Holman. *Edmas: An Object-Oriented Locally Distributed Mail System.* Technical Report 84-08-03, Department of Computer Science, University of Washington, August 1984.

[4] A. Barak and A. Shiloh. *A Distributed Load Balancing Policy for a Multicomputer.* Technical Report, Hebrew University of Jerusalem, 1984.

[5] A. Bechtolsheim, F. Baskett, and V. Pratt. *The SUN Workstation Architecture.* Technical Report Tech Report 229, Comp. Sci. Lab, Stanford University, March 1982.

[6] B. Bershad. *Load Balancing With Maitrd.* Technical Report UCB/CSD 82/276, University of California, Berkeley, 1985.

[7] Andrew P. Black. *Eden Programming Language.* Technical Report 85-09-01, Eden Project, University of Washington, September 1985.

[8] Andrew P. Black. Supporting Distributed Applications: Experience with Eden. *Proceedings of the 10th SOSP*, 19(5):181–193, 1985.

[9] S. Bokhari. Dual processor scheduling with dynamic reassignment. *IEEE Transactions on Software Engineering*, SE-5(4):341–349, July 1979.

[10] R. Bryant and R. Finkel. A stable distributed scheduling algorithm. *Proc. 2nd Int Conf. Dist. Comput. Sys.*, 314–323, 1981.

[11] D. Butterfield and G. Popek. Network tasking in the LOCUS distributed unix system. *Proc. of the 1984 Summer USENIX Conf.*, 1984.

[12] D. Cheriton and W. Zwaenepeol. The distributed v kernel and its performance for diskless workstations. *Proceedings of the 9th SOSP*, 17(5):128–139, October 1983.

[13] T. Chou and J. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, SE-8(4):401–412, July 1982.

[14] W. Chu, L. Holloway, M. Lan, and K. Efe. Task allocation in distributed processing. *IEEE Computer*, 557–70, November 1980.

[15] D. Eager, E. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated dynamic load sharing. *Performance Evaluation*, to appear.

[16] D. Eager, E. Lazowska, and J. Zahorjan. Dynamic load sharing in homogenous distributed systems. *IEEE Trans. on Software Eng.*, to appear.

[17] P. Enslow Jr. What is a 'distributed' data processing system? *Computer*, 11:13–21, Jan. 1978.

[18] D. Farber. The distributed computing system. *Proc. of Compcon Spring 73*, 31–34, 1973.

[19] D. Farber and K. Larson. The system architecture of the distributed computer system - the communications system. *Symp. on Comput. Networks*, April 1972.

[20] Cara L. Holman. *The Eden Calendar.* Technical Report, Eden Project, University of Washington, March 1985.

[21] R. Holt. *Concurrent Euclid, the Unix System, and Tunis.* Addison-Wesley, 1983.

[22] L. Lamport. LaTeX *A Document Preparation System.* Addison-Wesley, 1986.

[23] K. Lantz, Nowicki. W., and M.. Theimer. An empirical study of distributed application performance. *IEEE Transactions on Software Engineering*, SE-11:1162–1174, 1985.

[24] E. Lazowska, J. Zahorjan, D. Cheriton, and W. Zwaenepoel. *File Access Performance of Diskless Workstations.* Technical Report, University of Washington, 1984.

[25] W. Leland and T. Ott. Load balancing among loosely-coupled computers. *Proc. of the 1986 Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, May 1986.

[26] B. Liebowitz and J. Carson. *Distributed Processing.* IEEE, 1978.

[27] Barbara Liskov. On linguistic support for distributed programs. *IEEE Transactions on Software Engineering*, 203–210, May 1982.

[28] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. *Proceedings of the Computer Network Performance Symposium*, 47–55, September 1982.

[29] P. Ma, E. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C-31:41–47, January 1982.

[30] R. Metcalfe and D. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19:395–404, July 1976.

[31] L. Ni, C. Xu, and T. Gendreau. Distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, SE-11:1153–1161, 1985.

[32] J. Noe, A. Proudfoot, and C. Pu. *Replication in Distributed Systems: The Edeb Experience.* Technical Report TR-85-08-06, Eden Project, University of Washington, September 1985.

[33] M. Powell and B. Miller. Process migration in DEMOS/MP. *Proceedings of the 9th SOSP*, 17(5):110–119, October 1983.

[34] C. Pu and J. Noe. *Design and Implementation of Nested Transactions in Eden.* Technical Report TR-85-12-03, Eden Project, University of Washington, February 1986.

[35] D. Ritchie and K. Thompson. The unix time-sharing system. *Bell System Journal*, 56(6), July-Aug. 1982.

[36] B. Shneiderman. Response time and display rate in human performance with computers. *Computing Surveys*, 16(3):265–285, September 1984.

[37] K. Shoens and C. Leres. Mail reference manual. Berkeley Unix Manual.

[38] R. Stallman. EMACS manual for TWENEX users. 1980.

[39] Harold Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 85–93, 1977.

[40] A. Tantawi and D. Towsley. Optimal load balancing in distributed computer ststems. *Journal of the ACM*, 32(2):445–465, April 1985.

[41] W. Teitelman. *The Cedar Programming Environment: A Midterm Report and Examination.* Technical Report CSL-83-11, Xerox Palo Alto Research Center, 1984.

[42] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the v-system. *Proceedings of the 10th SOSP*, 19(5):2–12, 1985.

[43] B. Walker, G. Popek, R. English, et al. The LOCUS distributed operating system. *Procs. of the 9th SOSP*, 17(5):49–70, October 1983.

[44] Y. Wang and R. Morris. Load sharing in distributed systems. *IEEE Transactions on Software Engineering*, C-34:204–217, 1985.

[45] S Zatti. *A Multivariable Information Scheme To Balance The Load In A Distributed System.* Technical Report UCB/CSD 85/234, University of California, Berkeley, May 1985.