

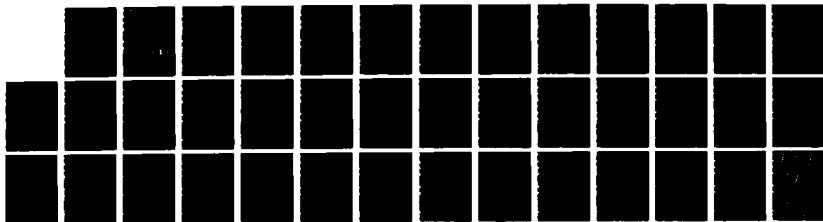
AD-A186 615

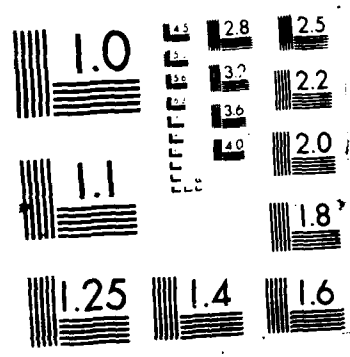
SAHAYOG: A TESTBED FOR LOAD SHARING UNDER FAILURE(U)
MARYLAND UNIV COLLEGE PARK INST FOR ADVANCED COMPUTER
STUDIES P DIKSHIT ET AL JUL 87 UMIACS-TR-87-34
UNCLASSIFIED N00014-87-K-0124

1/1

F/G 12/7

ML





DTIC FILE COPY

(12)

AD-A186 615

UMLACS-TR-87-34, ✓
CS-TR-1891 ✓

July, 1987

**SAHAYOG: A Testbed for Load Sharing Under
Failure¹**

Piyush Dikshit

Department of Computer Science

Satish K. Tripathi

Department of Computer Science and
Institute for Advanced Computer Studies

University of Maryland
College Park, MD 20742

**COMPUTER SCIENCE
TECHNICAL REPORT SERIES**



DTIC
ELECTE
S OCT 26 1987 **D**
CO D

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

87 10 12 00

UMIACS-TR-87-34, ✓
CS-TR-1891 ✓

July, 1987

SAHAYOG: A Testbed for Load Sharing Under Failure¹

Piyush Dikshit

Department of Computer Science

Satish K. Tripathi

Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

ABSTRACT

This paper describes the implementation of a testbed for distributed load sharing in a broadcast local area network consisting of AT&T 3B2 minicomputers. The testbed is designed to evaluate dynamic load sharing policies which use the process migration approach. It contains an optional fault-tolerance feature to handle single-node failures. Five load sharing algorithms are implemented and evaluated using the testbed under different loading conditions and for various parameter values. The experiments done using these algorithms confirm some earlier results about load sharing and also provide some new insights.

DTIC
ELECTE
OCT 26 1987
S
D

1 This research was supported in part by a contract (N00014-87-K-0124) from the Office of Naval Research to the Department of Computer Science, University of Maryland

2 The computer equipment for the testbed was obtained through a grant from AT&T

1
DISTRIBUTION STATEMENT
Approved for public release
Distribution Unlimited

Contents

1	Introduction	1
2	Load sharing: survey and classification	1
2.1	Characteristics of distributed load sharing algorithms	1
2.2	A hierarchy of load sharing	2
2.3	Discussion of previous work	3
3	A testbed for load sharing	3
3.1	Goals	3
3.2	Resources	4
3.3	Constraints	5
3.4	Assumptions	5
3.5	Testbed description	6
3.5.1	Overview	6
3.5.2	Fault tolerance in the testbed	6
3.5.3	Communication interfaces	8
3.5.4	Messages	8
3.5.5	Control Host	11
3.5.6	Local Host	12
3.5.7	Message flows	15
3.5.8	User commands	17
3.6	A typical experiment in load sharing	18
3.7	Matching goals to implementation	20
3.8	Future enhancements	20
4	Load sharing algorithms	21
4.1	Selection of algorithms	21
4.2	Implementation issues	21
4.3	Description of selected algorithms	22
4.3.1	Algorithm 1	22
4.3.2	Algorithm 2	22
4.3.3	Algorithm 3	22
4.3.4	Algorithm 4	22
4.3.5	Algorithm 5	23
5	Experimental evaluation	23
5.1	Environment	23
5.2	Format of result tables	24
5.3	Experiment 1: Performance of all the algorithms	25
5.4	Experiment 2: Timer-driven scheduling in receiver-initiated algorithms	29
5.5	Experiment 3: Single-node failure	29
5.6	Experiment 4: Effect of service-time distribution	31
6	Summary and conclusions	31
	References	32

List of Tables

Table 1: Performance of all the algorithms (Load = VHML)	26
Table 2: Performance of all the algorithms (Load = LLLL)	26
Table 3: Performance of all the algorithms (Load = MMMM)	27
Table 4: Performance of all the algorithms (Load = HHHH)	27
Table 5: Performance of all the algorithms (Load = VVVV)	28
Table 6: Effect of timer-driven scheduling on receiver-initiated algorithms (Load = VHML)	28
Table 7: Effect of single-node failure (Load = VHML)	30
Table 8: Effect of variation in the service-time distribution of jobs (Load = VHML) . .	30

List of Figures

Figure 1: Overall Configuration	7
Figure 2: Control Host	9
Figure 3: Local Host	9
Figure 4: Control Message	10
Figure 5: NI packet-header	10
Figure 6: Job Duplication	16
Figure 7: Job Transfer	16



ACCOUNT FOR	
NTS GRA21	☒
END LAB	☐
Completed	☐
BY	
DATE	
REMARKS	
A-1	

1 Introduction

In recent times, tremendous improvements in communication technology have led to the evolution of *distributed systems*, which consist of a number of autonomous computers interacting through a communication network. Distributed systems offer a multiplicity of resources. If the resources can be shared effectively, users can get better performance and improved reliability. A job need not wait in queue at a host to get service from a resource which is temporarily unavailable or overloaded. If a copy of the same resource is idle at some other host, it can be used instead.

Resource sharing in a distributed system can be achieved using algorithms for *distributed resource scheduling*. The algorithms should have the following desirable features:

- transparency — the user should not have to worry about the physical location and accessing method of the resources.
- fault-tolerance — if a resource fails, jobs at that resource should be taken care of.

When resource sharing is done to improve the performance of a distributed system, it is also called *load sharing*. The load at each host is represented by a load index, which is used by the load sharing algorithms to make scheduling decisions. The hosts exchange their load indices using status messages. The algorithms have two types of costs: a communication cost and an execution cost. The communication cost is due to the exchange of status messages across the network. The execution cost is due to the computations of an algorithm. Load is balanced by transferring jobs from over-loaded hosts to under-loaded hosts. A job can be transferred only if:

- a job transfer facility is provided by the distributed system.
- the job can be executed at the host to which it is transferred.

A lot of load sharing algorithms have been proposed in literature. This paper describes a testbed for evaluating a given load sharing algorithm and determining the conditions under which it performs best. The testbed provides a common yardstick for comparing load sharing algorithms. It contains an optional fault-tolerance feature to handle single-node failures. To illustrate the usefulness of the testbed, five load sharing algorithms from [LIVN82], [STA84B], [EAG86B] and [MIRC87] are implemented and evaluated.

2 Load sharing: survey and classification

2.1 Characteristics of distributed load sharing algorithms

An environment for a distributed load sharing algorithm is described in [STA84B]. Each host acts as a controller and has an identical copy of the algorithm. These copies run asynchronously and concurrently, continually making scheduling decisions over time. The decisions are based on the global state of the system as seen by the local controller. There is no *master controller* in the system at any time.

A distributed load sharing algorithm consists of two basic elements [LIVN82]: an *information element*, and a *control element*. The information element maintains information about the state of the distributed system, which is used by the control element in making scheduling decisions. This information characterizes the load at other hosts in the system. The load at a host can be represented in several ways — queue-lengths of the jobs waiting for resources, utilization of the resources, estimated average response time for the jobs, etc. In a study on load indicators

[FERR86], a linear combination of resource queue lengths is proposed as a reasonable measure of load.

The information element can work in a periodic or asynchronous manner. Asynchronous information update can be done after every change in the state of the system, or upon request from another information element. In certain cases, the information may be piggybacked with other messages (see [NI 85]). The information may be broadcast to all hosts, or sent to only a subset of hosts.

The frequency of information exchange is critical. Frequent exchange usually implies up-to-date state information. This helps the control element to make better scheduling decisions. The benefit from frequent information exchange is not without the associated cost of communication. The trade-off depends upon several factors, including the properties of the communication medium.

The control element uses the information provided by the information element to decide whether one or more processes at a local host need to be transferred to another. The transfer can be of two types: *sender-initiated*, in which congested hosts search for lightly-loaded hosts to which processes may be transferred, or *receiver-initiated*, in which lightly loaded hosts search for congested hosts from which jobs may be transferred. If a transfer is necessary, the control element has to make two decisions:

- which process (processes) to transfer.
- which host to transfer it (them) to.

The transfer is called *pre-emptive* if it requires pre-empting a process in the middle of its execution. The control element estimates the cost of the transfer and if it exceeds the cost of executing it locally, then no performance improvement can be obtained by transferring it. The transfer cost is the sum of the cost of removing the process from its current environment and the cost of migrating it through the network. The cost of removal is more in the case of pre-emptive transfer. The cost of migration depends on the characteristics of the network. In [MIRC87], queueing analysis has been used to show that network delays have a significant effect on the performance of the algorithms.

As in the case of the information element, the frequency with which the control element is activated is critical. Ideally, the frequency of activation should be a function of the load. Frequent activation at low loads can result in too much overhead. At high loads, if the activation is not done often enough, the algorithm cannot react quickly to changing conditions.

2.2 A hierarchy of load sharing

In [LELA86], Leland and Ott describe a natural hierarchy of load sharing schemes to classify approaches to load sharing. The proposed hierarchy contains two mechanisms for assigning jobs to hosts: *static assignment* and *dynamic assignment*.

Load sharing by static assignment restricts the number of hosts on which a given job can execute. It does not take into account the state of the system. Transfer decisions are made either in a deterministic or stochastic manner. The techniques to achieve static load sharing are simple since there is no need to maintain system state information, but they are not as effective as the more informed dynamic assignment techniques.

Dynamic assignment techniques use information about the system state to assign jobs to hosts. Dynamic assignment can be done in the following ways:

1. Assignment by user: Based on an informal estimate of the expected response time, the user chooses the host where a given job should be executed.
2. Initial Placement: When a job is initiated, the operating system uses information about the loads at the hosts to decide where it should be executed. If the chosen host is different from the one where the job originates, the job is moved to that host.
3. Process Migration: The operating system reassigns jobs to different hosts while they are executing.

Dynamic assignment can be done in a centralized or distributed manner. In the case of distributed dynamic assignment, load sharing can be sender-initiated (where job transfer is initiated by overloaded hosts) or receiver-initiated (where idle hosts initiate job transfer).

2.3 Discussion of previous work

A lot of load sharing studies have been analytic in nature ([EAG86A], [EAG86B], [LIVN82], [MIRC87], [NI 82], [WANG85]), where queueing analysis has been used. Various simulation studies are reported in ([STA84B], [WANG85], [ZHOU86], [LELA86]) and implementation studies are discussed in [BARA85], [BERS85], [EZZA86], [HAC 86], [HWAN82], [KORR86], and [ZHOU86].

Both analytic and simulation studies suffer from the drawbacks of modeling techniques in general. It is difficult to represent a complex real system in a single model. An attempt to do so makes the model extremely complex. To keep the model tractable, it is necessary to make simplifying assumptions about the behavior of the system. Due to these assumptions, the analysis may not be accurate.

Only a limited number of the implementation studies use the process migration approach. One implementation with process migration is presented in [BARA85]. However, the load sharing policy used there has a drawback: load sharing is not transparent to the process which is migrated. The process has to make an explicit request to be considered for load sharing.

Very few implementations use receiver-initiated algorithms. It is difficult to implement receiver-initiated algorithms because they often require pre-empting a job in the middle of its execution for transferring it. In receiver-initiated algorithms, idle hosts request jobs from heavily loaded hosts. Load sharing attempts to reduce the probability of a host being idle while there are jobs waiting at another host. So, receiver-initiated algorithms can be expected to provide good performance improvement.

The issue of fault tolerance in load sharing has not received much attention in the literature. Some of the algorithms that have been proposed use centralized components. Thus they do not conform to the basic structure of truly distributed load sharing algorithms. The fault tolerance studies described in [TRIP85], [CHOU83] and [PATN86] consider load redistribution *after* a failure occurs in the system. To use load sharing for performance improvement, load has to be distributed not only after a failure, but also when all the hosts are operational.

The testbed [DIKS87] described in this paper can be used to evaluate dynamic load sharing policies which use process migration. It does not allow pre-emptive job transfer, but it can be used to evaluate receiver-initiated algorithms. It can provide fault tolerance during load sharing: a load sharing algorithm that is not fault-tolerant can be implemented in the testbed with very little modification to make it fault-tolerant.

3 A testbed for load sharing

3.1 Goals

In our implementation, we set our goal to have a testbed which:

1. permits the evaluation of a wide range of distributed load sharing policies.
2. is modular enough to allow the "insertion" of a load sharing algorithm without too much effort.
3. allows parametric execution of the algorithms. This should enable the tuning of the parameters of an algorithm without modifying its code.
4. provides basic fault-tolerance so that the advantage in performance from load sharing is not at the cost of poor reliability.
5. minimizes the impact of fault-tolerance on the load sharing algorithm. It should not be necessary to make too many modifications to the load sharing algorithm to make it fault-tolerant.
6. contains a facility to generate statistics from data produced during the execution of the algorithm. These statistics should provide the costs and benefits of using the algorithm.
7. provides a single interface to the user. The distributed system consists of several hosts. While doing experiments, the user should be able to monitor the whole system by interacting with a single host.

3.2 Resources

The distributed system used for the implementation consists of five AT&T 3B2 minicomputers connected in a bus topology using a 10 Mbit/sec coaxial cable. The Local Area Network formed by these hosts is called 3BNET. It is compatible with ETHERNET and it uses the standard CSMA/CD (Carrier Sense Multiple Access/Collision Detection) communication protocol.

All the 3B2s have a 10 Mbyte hard disk and use the AT&T *UNIX*^R operating system. The system software [ATT84C] on the machines provides facilities for communication among processes running on the same host or on different hosts on the 3BNET.

Each 3B2 has a unique physical address. The 3BNET software [ATT84A] running at each host contains a Network Interface facility which permits user processes to create and use ports for sending/receiving messages to/from the network. There are two types of ports — a local address port and a broadcast port. The broadcast port is used exclusively for sending and receiving broadcast messages. The attributes of these ports (for example, buffer size, number of buffers, etc.) have default values that can be changed depending on requirements.

Processes running on the same host can use the Inter-Process Communication (IPC) facility for communication. The System V IPC package [ATT84B] provides three types of communication services — messages, shared memory and semaphores.

To communicate using messages, processes create message queues using system-provided primitives. The message exchange is done through send and receive operations on these message queues. For both these operations, processes have the option to wait till a message can be sent/received. If the process that does a send/receive does not wish to wait, it can get back control immediately with an error-code specifying whether the operation was successful or not.

The shared memory feature allows different processes to read from/write into the same memory area. The size of the memory that is to be shared is decided by the process when it makes a request for memory allocation.

The semaphore option of IPC can be used to prevent two or more concurrent processes from executing their critical sections at the same time. Semaphores must be used when the processes use shared memory and the input-output operations on the common memory area are done inside a critical section.

Another useful feature of the 3BNET software is the facility for transferring files among the hosts. Using a primitive called *nisend*, it is possible to transfer among the hosts on the 3BNET. This feature has been used extensively in the implementation of the testbed to transfer files for fault tolerance, load sharing and statistics generation.

AT&T *UNIX*^R provides a software-interrupt facility through a system call called *signal*. A signal-handling routine can be specified by the user process that wants to *catch* a signal. Signals have been used in the testbed extensively for setting alarms and for asynchronous communication among processes.

3.3 Constraints

Unlike Berkeley *UNIX*^R, AT&T *UNIX*^R does not have an *uptime* command to provide the load average of the local host. Therefore, the load is computed and maintained by the testbed software. AT&T *UNIX*^R does not have a long-term CPU scheduler. A user process gets a time-slice very quickly after being submitted for execution. If load sharing is done using the process migration approach (See Section 2.2), pre-emptive job transfer is necessary. Also, in the initial placement approach, it is not possible to implement receiver-initiated policies without pre-empting a job. Pre-emptive job transfer is expensive and causes a considerable overhead. The need for pre-emptive job transfer is obviated in the testbed by using a long-term scheduler. The scheduler maintains a queue of jobs before they are submitted to the *UNIX*^R operating system for execution. The degree of multiprogramming (which is the maximum number of jobs that can be active at any time) can be specified by the user when initiating a load sharing experiment . It is used by the long-term scheduler to control the number of jobs that are being executed under the *UNIX*^R operating system at any given time. In the absence of the source code for AT&T *UNIX*^R, the problems mentioned above cannot be solved by modifying the *kernel*.

3.4 Assumptions

We make the following assumptions in our implementation:

1. the communication protocol in the 3BNET is robust. No explicit acknowledgement mechanism to prevent messages from getting lost is implemented in the testbed.
2. the testbed is meant for testing distributed load sharing policies. If a centralized load sharing policy is tested, the testbed has to be modified.
3. the jobs that simulate the workload at each local host are independent and statistically identical.
4. each job can be executed at any host.

5. fault-tolerance is provided for single-node failure only. When a host fails, all the jobs that get aborted are re-initiated at another host. There is no check-pointing mechanism implemented to restart a job from the point of failure.
6. there is no inter-process communication between jobs running at different hosts. (If this is allowed, the fault tolerance implementation does not work).
7. job transfers are not done in a pre-emptive manner i.e. during job transfer, a job is not pre-empted in the middle of its execution.

3.5 Testbed description

3.5.1 Overview

The overall configuration of the testbed is depicted in Figure 1. One of the 3B2s has been chosen to act as the "control host", while the others have been designated to be "local hosts". Every host has a logical name and a unique network-wide physical address. A unique host-index is assigned to each host as shown in the figure. The control host has a host index of 0.

The control host is used to initiate and monitor controlled experiments that run at the local hosts of the system. It has a user interface which prompts users for inputs and displays messages on the 3B2 screen at the control host. It is the single point for user interaction to the testbed. A Central Manager (CM) is responsible for executing the commands entered by the user by issuing control messages to its agent at every local host — the Local Manager(LM). A statistics generation facility is also provided at the control host.

All local hosts run exactly the same software, which consists of:

- the Local Manager.
- a module to generate jobs based on parameters specified by the user. The characteristics of job generation determine the load at each local host.
- long-term scheduler. The scheduler contains a queue into which it places jobs as they arrive on a first-come first-served basis.
- a module that implements the control and information components of the load sharing policy being studied.
- a module to handle the fault-tolerance feature.

3.5.2 Fault tolerance in the testbed

The testbed is designed to evaluate load sharing policies which use the process migration approach; the fault tolerance feature of the testbed is also based on process migration. The local hosts are configured in pairs by the user. Each host monitors its cohost using health signals. When a job arrives at a host, a copy of the job is migrated to the cohost. If a host fails, its cohost re-initiates all its unfinished jobs by using their duplicate copies.

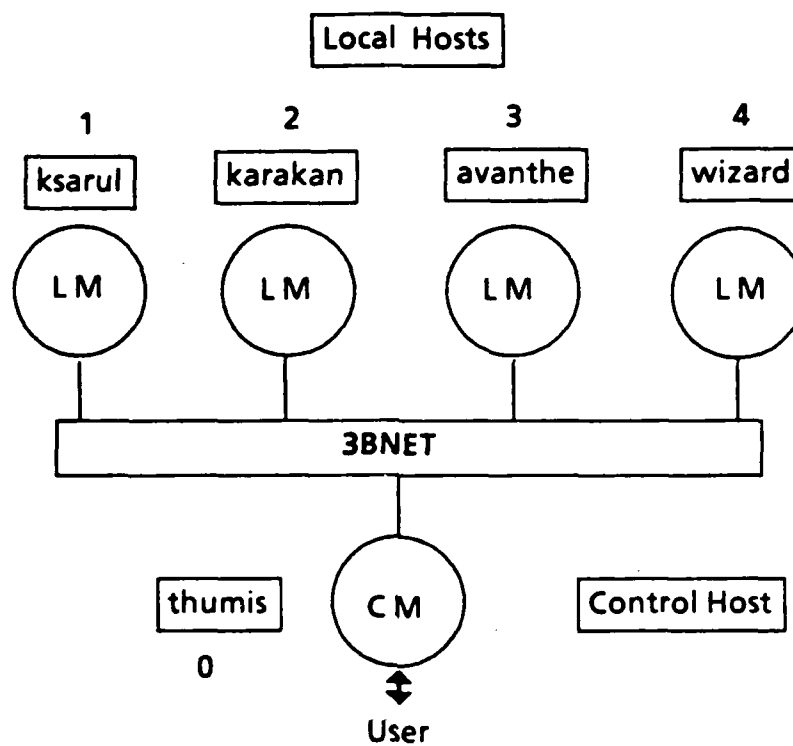


Figure 1 : Overall Configuration

3.5.3 Communication interfaces

Communication among the processes that constitute the testbed is asynchronous. The communication interfaces discussed below are illustrated in Figures 2 and 3. Communication is used only for the exchange of control information. For transferring jobs, the *nisend* primitive is used. Processes running at the same host use IPC queues for communication while those running at different hosts communicate through Network Interface ports.

There are two IPC queues at each host shared by all the processes that are active there — *talkq* and *hearq*. All processes receive their messages through the *hearq*. They use the *talkq* to send messages to remote processes. To send a local message, a process can put it directly in the local *hearq*. *Talkq* and *hearq* are named after the daemons TALK and HEAR that run at every host. They hide the details of the NI ports from other processes.

TALK waits on the *talkq* in an infinite loop. When a message arrives, it picks it up, translates logical host names to physical host addresses, and writes it to the local or broadcast NI port depending on the destination of the message.

HEAR is the exact opposite of TALK. It waits on the local NI port. It picks up messages from the port and inserts it into the *hearq* for being received by one of the processes which are waiting on the *hearq*.

There is another daemon called WILD, which works exactly like HEAR except that it reads the broadcast port instead of the local port. It handles only broadcast messages. The control host does not receive any broadcast messages so it does not use the WILD daemon.

3.5.4 Messages

All processes use the same message format (Figure 4) for exchanging control messages. It consists of the following fields:

- **desthost**(8 bytes) : Logical name of the destination host.
- **destmtyp**(4 bytes) : Integer identifying destination process.
- **srchost** (8 bytes) : Logical name of the source host.
- **srcmtyp** (4 bytes) : Integer identifying source process.
- **control** (1 byte) : Identifies the type of message.
- **datalen** (4 bytes) : Integer containing length of mtext.
- **mtext** (157 bytes): Contains control parameters.

When a process puts a control message in the *talkq* or the *hearq*, it attaches a 4-byte header to it. This header is an integer which identifies the process which will remove it from the queue.

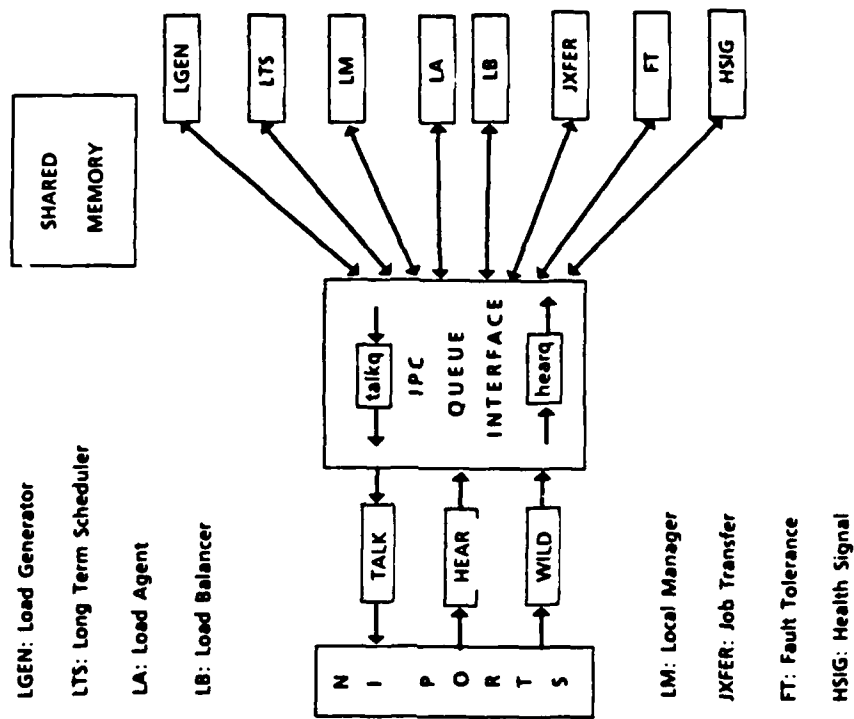


Figure 2 : Control Host

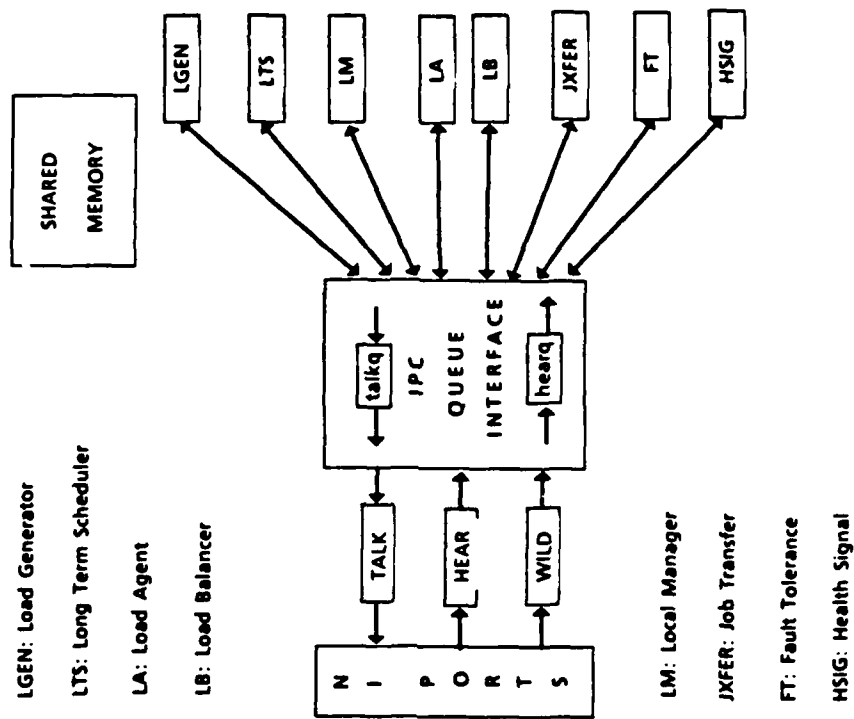


Figure 3 : Local Host

8	4	8	4	1	4	157
desthost	destmtyp	srchost	srcmtyp	ctrl	datalen	mtext

(Numbers represent length in bytes)

Figure 4 : Control Message

6	6	2
destaddr	srcaddr	ptype

(Numbers represent length in bytes)

Figure 5 : NI packet-header

When a control message leaves a host through the NI port interface, it has a 14-byte packet-header (Figure 5) which consists of:

- **destaddr** (6 bytes): Physical address of destination host.
- **srcaddr** (6 bytes): Physical address of source host.
- **pctype** (2 bytes): Protocol type.

This header is used by the 3BNET software for message routing in the network. The size of a packet in the testbed is 200 bytes.

The TALK daemon uses the **desthost** and **srchost** fields in the control message to build the packet-header. The mapping from logical to physical addresses is done using a file called "localhosts" resident at all hosts. The HEAR and WILD daemons remove the packet-header upon receiving a message through the NI ports.

3.5.5 Control Host

The processes that run at the Control Host, along with the communication interfaces, are shown in Figure 2. The TALK and HEAR daemons have been described in the previous section. The functions of the other processes are given below:

- **Central Manager(CM)**: This is the central point of control in the testbed. All the commands entered by the user are initiated and monitored here. For most commands, a control message is sent to one or more Local Managers, which act as agents of the CM at the local hosts. The Local Managers respond to these control messages, indicating whether the requested service was completed successfully or not. The CM uses the messages from the LMs to format a response for the user.

The most frequently used function of the CM is monitoring a load sharing experiment. The user can specify the duration of an experiment at the time of initiating an experiment. The CM is responsible for carrying out the experiment and informing the user when it is over. During an experiment, each local host creates its own job-stream. The generation of jobs is stopped when the user-specified duration of the experiment is over. An experiment is complete only after all the jobs in the system are finished. The CM can know that an experiment is over in one of the following ways:

- by polling the local hosts for the number of jobs remaining to be executed after the expiry of the duration of the experiment.
- by being informed by the local hosts when their job-streams are over.

The first alternative is implemented for the following reason: jobs can be transferred during load sharing, so they may not get executed at the local host where they originate. Thus, even after job-generation stops at a local host, it may have jobs waiting to be executed.

After initiating the experiment at the local hosts, the CM sets a built-in alarm for the duration of the experiment. When the alarm expires, the CM polls all the local hosts to determine if the experiment is over. If it is not, it sets an alarm once more. This time the alarm is set for a short period only (2 minutes, for example). When the alarm expires, the CM polls all the hosts once again. This polling sequence continues at 2 minute intervals until all the local hosts indicate that the experiment is over. Thereafter, the CM generates a message for the user to tell him that the experiment is done.

- **Load Balancing Shell (LBSHELL):** This is the only user-interface to the testbed. If a command requires parameters, LBSHELL prompts the user for each parameter with a short description of what is expected as input. (In the current implementation, LBSHELL expects all inputs in upper-case.)

The success or failure of a user's request is indicated by the outputs from LBSHELL to the user. User commands are acknowledged immediately by the LBSHELL as soon as they are received. Invalid commands are rejected with an error message. The result of the execution of the command is displayed later as a response.

A useful feature of LBSHELL is that the user can exit from it without the fear of losing any messages that come in while not interacting with it. When a long experiment is running at the local hosts, the user can utilize the control host for some other purpose, for example, to edit or compile programs. Experiments run only at the local hosts, so using the control host while an experiment is running does not affect the results of the experiment at all.

- **Statistics Generator (GENSTAT):** This is initiated by the CM when a user requests statistics for an experiment. Upon receiving the request, the CM sends a broadcast message to all local hosts asking for the log-files that were generated during the experiment. GENSTAT uses the log-files sent by the local hosts to generate performance statistics. After generating these statistics, it sends a message to the user through LBSHELL. The message identifies the local hosts whose log-files were used to generate statistics.

3.5.6 Local Host

The same set of processes run at each local host in the testbed. Figure 3 shows all of them with the communication interfaces. Besides TALK and HEAR, each local host has a WILD daemon to receive broadcast messages from the network.

The information that is common to all the processes is maintained in shared memory. Access to data in shared memory is controlled by semaphores. There are two main structures in shared memory that are used in all experiments. One is a table containing the logical names of all hosts in the network along with their status. The table also has an entry containing the load at each host. The other structure contains a queue of user jobs. Jobs are inserted into the queue upon arrival and removed when they are to be executed by UNIX or transferred to another host for load sharing.

The functions of the processes at each local host are:

- **Local Manager (LM):** As mentioned earlier, the Local Manager is the agent of the Central Manager at each local host. It receives control messages from CM and interprets them based on the **control** field in the message. After servicing the request in the control message, LM sends a response to the CM.

The LM is responsible for initiating all the processes at the local host, based upon requests received from the CM. The LM also terminates all the processes (except the Load Generator, which terminates by itself) using a *signal*. The LM keeps track of the process-ids of all active processes in a table which is updated every time a process is initiated or terminated by the LM.

In addition to doing process management, the LM answers status requests about its local host. Status requests are received from processes at other hosts. During an experiment on the fault-tolerance aspect of the testbed, the failure of a local host is simulated by the

termination of all processes running there *except* the LM. The LM continues to respond to status requests from processes at other local hosts telling them that its local host has failed.

- **Load Generator (LGEN):** The Load Generator creates the a stream of user jobs that constitute the load during a load sharing experiment. The load can be light(L), moderate(M), heavy(H) or very heavy(V), depending on the parameters of the LGEN.

Each job is CPU bound and it does some dummy computation in a loop. The number of iterations of the loop is used as an estimate of the service time required by a job. Another characteristic feature of a job is its physical size. Both these attributes are assigned to each job by LGEN.

The mean service time of all jobs is the same at all the hosts. Variations in load are produced by changing the inter-arrival time of the jobs. Both arrival and service times are generated randomly using a probability distribution with the appropriate mean. Service times can be deterministic, or follow a uniform or exponential distribution. Inter-arrival times follow a uniform or exponential distribution. The choice of the distributions to be used for arrival and service is based upon the parameters passed to LGEN when it is initiated by the LM. The LM, in turn, derives these parameters from the control message it receives from the CM.

At the time of its initiation, LGEN receives a parameter which indicates the duration of the experiment. LGEN keeps generating jobs until this duration expires. Then it sends a message to the LM saying that no more jobs will be generated, and terminates. LM uses this message to remove LGEN from its table of active processes.

- **Long Term Scheduler (LTS):** This process has been introduced to fill the gap of a long term scheduler in *UNIX*^R. It receives jobs from the LGEN and schedules them for execution. The number of jobs that can be active at any time is determined by the degree of multiprogramming which is passed to the LTS as a parameter at the time of its initiation.

Most jobs arrive at the LTS after being generated locally by LGEN. There are two ways in which jobs can arrive from another host — due to a job transfer from another local host (done for load sharing), or upon the failure of the cohost (all jobs that get aborted due to the failure have to be executed all over again).

When a job arrives, the LTS sets some of its attributes. The attributes of a job are :

- physical size (always assigned by LGEN)
- service-time (also assigned by LGEN)
- initiation-host (set by the LTS at the host where job was generated)
- initiation-time (set by the LTS at the host where job was generated)
- number-of-transfers (incremented by 1 for each job transfer)
- job-number (a local number assigned to each job received by LTS)

If the fault tolerance option is used, each job is duplicated at the cohost after its attributes are set, so that the cohost has all the information required to re-initiate the job if the host where the job originates has a failure. If job duplication cannot be done, the job is not accepted by LTS. After successful job duplication, it is guaranteed that the job will be executed at some local host in the network.

A job is either initiated immediately or placed in a queue, depending on the degree of multiprogramming and the number of active jobs at the local host. When a job terminates, the LTS gets a *signal* and it logs the initiation-time and end-time of the job in a log-file (used later for statistics generation). The cohost is also informed about the successful termination of the job.

The instantaneous load at a local host is given by the number of active jobs plus the number of jobs in queue. Due to the arrival and termination of jobs this load keeps changing. LTS is responsible for reflecting these changes in the shared memory table in which the load at all the local hosts is maintained.

For some load sharing policies, the LTS also initiates the Load Agent and/or the Load Balancer when the load at the local host crosses a certain threshold.

- **Job Transfer (JXFER):** This process handles the movement of jobs to its local host. It is used for job duplication (for fault-tolerance) as well as job transfer (for load sharing). In both cases, it gets a message containing information about the size of the job that is being moved. so that it can estimate the amount of time it must wait for the job. The successful receipt of a job is acknowledged to the sender. If the job is not received, the sender is notified with a negative acknowledgement.
- **Health Signal (HSIG):** At regular intervals, the HSIG processes at cohosts exchange health signals which consist of "I am okay" messages. If an HSIG fails to receive an expected health signal from its peer HSIG, it sends a status request to the cohost's LM . If the response to the request indicates that the cohost has failed, HSIG notifies the following processes about the fact:
 - all LMs, so that the status of the failed host can be updated in the shared memory table at each local host.
 - the CM, so that the user can be informed of the failure.
 - the Fault Tolerance process, so that unfinished jobs of the cohost can be restarted locally.
- **Fault Tolerance (FT):** This process is responsible for the fault-tolerance feature of the testbed. When a job arrives at the cohost, the FT receives all the attributes of the job along with its job-number in a "job-begin" message from the cohost's LTS. An entry is made for the job in a local table maintained by FT. The job-number acts as an index to the entry.

When a job terminates at the cohost, or gets transferred out of the cohost due to load sharing, FT gets a "job-end" message. The job-number sent in the message is used to remove the entry for that job in FT's table containing unfinished jobs of the cohost.

If the cohost fails, FT comes to know it through a message from the local HSIG. Subsequently, it sends all the jobs in its table to the local LTS so that they can be re-initiated.

When a local host fails, the job entries in its FT's table are lost. These jobs get executed at the cohost, however.
- **Load Agent (LA):** This is the information component of the load sharing policy. It may broadcast the local load periodically or respond to inquiries about the local load. Its function is completely dependent on the policy that it corresponds to.

- Load Balancer (LB): This is the control component of the load sharing policy. It makes load sharing decisions and initiates a job transfer as and when necessary.

After a job has been successfully transferred by LB, it sends a "job-end" message for that job to the cohost's FT so that the latter can remove the job from its table of unfinished jobs. LB does not send the "job-end" message until it receives an acknowledgement for the job transfer from the destination host. The LTS at the destination host sends this acknowledgement only after duplicating the job at its cohost. LB has to wait for the acknowledgement before informing the cohost's FT (otherwise the job may be "lost").

3.5.7 Message flows

There are two reasons why jobs need to be moved from one local host to another during a load sharing experiment in the testbed. One is *Job Duplication* done by the LTS when it receives a job. This is done for fault-tolerance. The other is *Job Transfer* done by the LB to achieve load sharing.

The message flows for these two types of job movement are illustrated with the help of timing diagrams in Figures 6 and 7.

Job Duplication messages are explained below:

- Message 1: LTS tells the JXFER at the *cohost* about the size of the job being duplicated.
- Message 2: After JXFER has verified that the job has been duplicated, it sends an acknowledgement to the LTS.
- Message 3: On getting JXFER's acknowledgement, LTS sends the description of the job to FT.
- Message 4: FT acknowledges the receipt of the job description.

After a job has been duplicated, its execution is guaranteed in the testbed in the event of single-node failure.

Job Transfer messages are explained below:

- Message 1: LB tells the JXFER at the *destination* host about the size of the job being transferred.
- Message 2: After JXFER has verified that the job has been transferred, it sends an acknowledgement to LB.
- Message 3: On getting JXFER's acknowledgement, LB sends the description of the job to LTS at the *destination* host.
- Message 4: LTS carries out Job Duplication for the job at its cohost. After successful Job Duplication, it sends an acknowledgement to LB.
- Message 5: LB sends the job number of the job that has been transferred to the FT at the *cohost*.

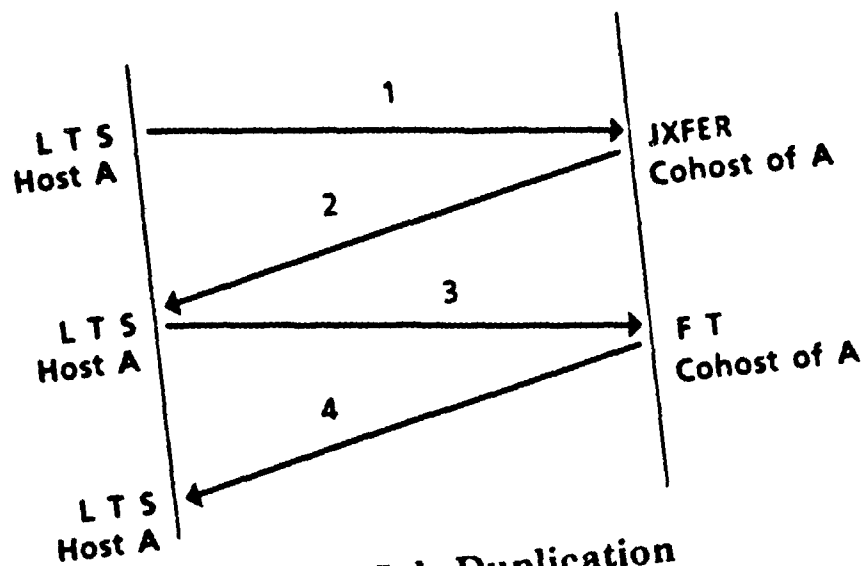


Figure 6: Job Duplication

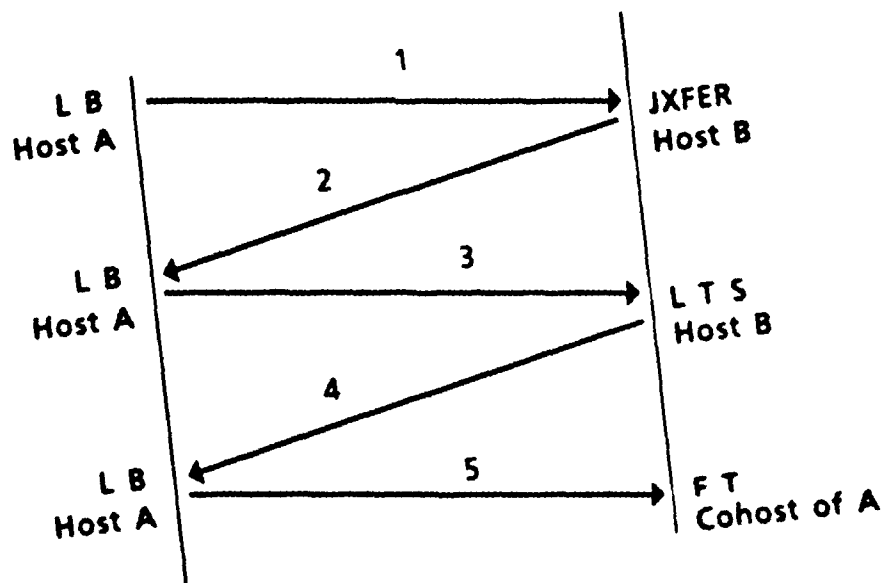


Figure 7: Job Transfer

3.5.8 User commands

The user interface (LBSHELL) has been described earlier in Section 3.5.5. In this section, the details of processing each command are being presented. The commands can be classified as follows:

- Commands for carrying out experiments:

- **SYNCALL:** This command is used to synchronize the time at all local hosts with the time at the control host. The CM sends the time at the control host to all the LMs. The LMs use it to set the local time at their respective local hosts.
- **EXPT:** This is the command that is used most often. It initiates a load sharing experiment. The input from the user is used to construct a control message. The CM executes the command in two steps:
 1. It sends the control message to all the LMs. The LMs initiate the Long Term Scheduler(LTS), the Load Agent (LA) and the Load Balancer (LB) at their respective hosts. In some cases, there may be no LA process to be initiated. After this, all LMs send a response to the CM saying that they are ready to begin the experiment.
 2. CM broadcasts a message to all LMs asking them to initiate their respective Load Generators (LGENs). The LMs initiate their LGENs and inform the CM.

When both these steps are done, a message is displayed to the user to tell him that the experiment has been initiated.

- **STAT:** This command is used to initiate the GENSTAT process for statistics generation. The CM requests log-files from all the LMs. The LMs initiate the transfer of the log-files to the control host and send an acknowledgement to the CM. On getting acknowledgements from all the LMs, the CM initiates GENSTAT with a parameter specifying the local hosts that are up. GENSTAT generates statistics using the log-files of these hosts. After statistics have been generated, GENSTAT sends a display to the user and terminates.
- **ABORT-EXPT:** This command is used to end a load sharing experiment prematurely. The LM at each local host executes the command by killing all active processes and flushing IPC queues at its local host.

- Commands for changing statuses of local hosts:

- **KILLHOST:** This command can be used to test the fault-tolerance feature of the testbed. The user is asked to specify the index of the "victim" host and the command is sent to the LM of that host. The LM executes the command by aborting all the active processes.
- **REVIVEHOST:** This command can be used to "revive" a host killed earlier by a KILLHOST command. The command is used to update the status of that host in shared memory at all the hosts.

- Commands for fault tolerance:

- **FTON:** This command switches on fault-tolerance in the testbed. The user is requested to specify node-pairs for cohosts. The LMs initiate the FT and HSIG processes at each local host.

- **FTOFF**: This command is used to switch off the fault-tolerance feature in the testbed. The LMs execute the command by terminating the FT and HSIG processes.
- Commands for general utilities:
 - **ADDALG**: This command is used to add an algorithm to the testbed. LBSHELL prompts the user for the parameters of the command. The parameters specify the number of arguments passed to LA (the information element) and LB(the control element) of the load sharing policy that is being added. These numbers are stored in a file called **algfile** resident at the control host. The contents of **algfile** are used to build the control message at the time of initiation of a load sharing experiment. Before running a load sharing algorithm for the first time, it must be added to the testbed using this command.
 - **PRINT**: If the control host has a printer hooked on to it, this command can be used to print a file resident at the control host.
- Commands for termination:
 - **SHUTOFF**: This command terminates all the processes in the network and removes all the communication interfaces. The communication interfaces have to be re-installed at each local host manually before the testbed can be used again.
 - **QUIT**: This command can be used to leave the LBSHELL temporarily.

3.6 A typical experiment in load sharing

The 'kernel' of the testbed consists of the communication interfaces, the daemons TALK, HEAR and WILD, the CM and all the LMs. Once these processes have been successfully initiated, all further interaction with the testbed can be done through the LBSHELL.

The sequence of events in the testbed during a typical load sharing experiment is given below. Some of this may be a repetition of what has been said earlier, but all the steps are listed for the sake of completeness:

1. Time Synchronization.

- The user enters the SYNCALL command to synchronize the time at all the hosts.
- CM sends a broadcast control message to all LMs with the time at the control host as a parameter.
- All LMs set the time at their respective local hosts and send an acknowledgement to the CM.
- After receiving acknowledgement from all the LMs, the CM displays a message to the user saying that the synchronization is done.

2. Switching on Fault Tolerance.

- After all hosts have been synchronized, the user issues an FTON command specifying the host-pairs which are to be cohosts to each other.
- CM sends a broadcast control message to all the LMs asking them to initiate the FT and HSIG processes.

- All LMs initiate FT and HSIG at their respective local hosts and send a message to CM saying that the required processes have been initiated.
- When all LMs have responded to the FTON request, the CM sends a display to the user saying fault-tolerance has been initiated.

3. Experiment Initiation

- When the user sees the message saying fault-tolerance has been switched on, he enters the EXPT command with the required parameters.
- The CM sends a broadcast control message to all LMs, containing the parameters entered by the user.
- The LMs initiate the LTS, the LA and the LB at each host, corresponding to the algorithm that is being evaluated. They respond to the CM indicating their readiness for the experiment.
- After all the LMs are ready, the CM sends a broadcast message asking the LMs to initiate their LGENs.
- The LGENs are started up and the actual experiment begins. The LMs inform the CM of the initiation of the experiment.
- The CM sets an alarm for the estimated duration of the experiment and informs the user that the experiment has been initiated.

4. Experiment Termination.

- When the alarm set by the CM expires, it sends a broadcast request to all the local hosts, asking them to respond with their respective loads. It keeps polling all the local hosts for their loads until the load at all the hosts becomes 0. Then the CM sends a broadcast request to each LM for the termination of LTS, LA and LB.
- The LMs terminate their respective LTSs, LAs and LBs, and send acknowledgements to the CM.
- The CM informs the user that the experiment is done.

5. Statistics Generation.

- The user can use the STAT command to generate the results of the experiment. The CM executes the STAT command by sending a request for log-files that were produced during the experiment to all the LMs.
- All LMs send their log-files to the control host along with an acknowledgement to the CM.
- On getting acknowledgements from all LMs, the CM initiates GENSTAT and informs the user that statistics will be generated.
- GENSTAT generates the statistics for the experiment and informs the user when statistics generation is over.
- The user can print the statistics or view them on his screen.

3.7 Matching goals to implementation

The goals for the testbed were identified in Section 2.1. Here we examine to what extent those goals are met.

The first two goals of the design were that the testbed should be flexible (to enable the evaluation of a wide range of load sharing policies) and modular (to allow "insertion" of an algorithm without much effort). The design of the testbed has been kept as open-ended as possible. If a load sharing policy can be decomposed clearly into its information and control components, the LA and LB processes for the algorithm can be written and the algorithm can be evaluated.

It was desired that the testbed should provide parametric execution of the algorithms. LBSHELL prompts the user for the parameters of an algorithm at the time of initiation. The code for an algorithms need not be modified to test the effect of changing its parameters.

Another requirement for the testbed was fault tolerance. Basic fault tolerance is provided in the testbed. If a local host fails in the testbed, there is minimal loss of jobs.

One of the goals was to minimize the impact of fault tolerance on the code for the load sharing algorithms. The only modification to a load sharing policy to make it fault tolerant is that it should have a timeout every time a message is expected from another host. If no message is received at the end of the timeout period, it can send a status query to the LM at that host. The LM always responds to the query. If the response indicates that the host has failed, the load sharing algorithm can stop waiting for the message.

The other goals of the testbed were that it should have a statistics generation facility and a single user-interface. GENSTAT takes care of statistics generation and can be used to get the costs and benefits of a load sharing algorithm. LBSHELL provides a single user-interface as desired. Once the 'kernel' of the testbed has been initiated, the user can interact with the whole system sitting at the control host.

3.8 Future enhancements

The control structure in the testbed with the CM-LM hierarchy is fairly open ended. It should be possible to enhance the testbed to deal with other problems in distributed systems besides load sharing. Some changes to the testbed that will make it more useful as an evaluation tool are listed below:

- Currently, the Load Generator generates only CPU bound jobs, and the load at each host is equal to the sum of the number of currently executing jobs and the number of jobs queued inside the Long Term Scheduler. LGEN can be modified to generate a richer job mix e.g. some jobs can be made IO bound. This modification must be accompanied by a change in the load index to account for all the resources that can be used by the jobs.
- The ability to dynamically change the parameters of a load sharing algorithm can be added. With this ability, the tuning of the parameters can be done with greater ease and accuracy.
- A dynamic monitoring facility can be introduced. The user can use it to obtain a more detailed view of the part of the testbed that he is interested in. In combination with the ability to change parameters dynamically, the monitoring facility can provide useful information about the response of a load sharing algorithm to specific stimuli.
- There is scope for improvement in the user interface. The window mechanism provided by the 3B2 system software can be used to provide a better screen to the user for interacting

with the testbed.

- The fault-tolerance feature can be implemented using a check-pointing mechanism to provide recovery from the point of failure within a job.

4 Load sharing algorithms

4.1 Selection of algorithms

We evaluate distributed load sharing algorithms that use dynamic assignment (i.e. used the current state of the system for making decisions) with the process migration approach. The reason for choosing dynamic policies is that they make more informed decisions than static policies and are hence expected to perform better. Algorithms with the process migration approach have been chosen because this approach has not been studied much in the implementation studies done so far.

Both sender-initiated and receiver-initiated policies have been chosen for evaluation. An attempt has been made to select algorithms that use different types of information and control mechanisms to test the versatility of the testbed.

Another criterion in the selection of the algorithms has been to see if the properties of a broadcast homogeneous local area network can be exploited. The 3BNET allows broadcast messages and algorithms that require this feature can be studied effectively in the testbed.

4.2 Implementation issues

The testbed developed for evaluation of load sharing policies is modular enough to allow the implementation of load sharing algorithms without too much effort. The following steps can be used to add a load sharing algorithm to the testbed:

1. The information and control elements of the algorithm need to be coded in the programming language C to generate the LA and LB processes that run at all local hosts in the testbed. Since copies of the same code execute at each local host, the coding should be done carefully to avoid deadlock problems during execution. One example of a deadlock problem is where the LB processes at different hosts wait in a circle, each waiting for the next to respond to a job request.
2. The names of the object modules produced by compiling the code should be "laN" and "lbN", where N is the number of the algorithm. (Each algorithm is identified by a unique algorithm-number)
3. LBSHELL should be modified so that it can prompt the user for the parameters of the algorithm when the algorithm is initiated.
4. The algorithm should be added to the testbed using the ADDALG command of the LBSHELL.

4.3 Description of selected algorithms

4.3.1 Algorithm 1

This is a sender-initiated algorithm that uses a threshold transfer policy in which the control element is activated at a host whenever its load exceeds a certain threshold T [EAG86B]. The value of T is a parameter to the algorithm. The control element selects a random subset of all the hosts in the network and probes them one by one to determine their respective loads. The probe is answered by the information element at the probed host, by simply responding with the value of the load at the host.

The number of hosts selected for probing (L_p) is the second parameter of the algorithm. For each probed host, the control element determines if transferring a job to that host would make its load greater than the threshold T . If not, a job is transferred to that host and the probing is discontinued. If so, then the next host in the set is probed. The probing continues until a candidate host is found or the set of L_p hosts gets exhausted.

4.3.2 Algorithm 2

This is a receiver-initiated algorithm that works in the opposite way to Algorithm 1 [EAG86B]. When a job finishes processing at a host, if there are less than T tasks remaining at the host, the control element is activated. It selects L_p random hosts and probes them for their load. The information element at each host works exactly as in Algorithm 1 — it responds to a probe by sending the value of the load at its host.

The control element uses the value of the load returned by a probe to determine if getting a job from the probed host would reduce the load at that host below the threshold T . If not, a job is requested from the host. If so, the next host in the random set is probed. Probing ends when a candidate host is found or the set of L_p hosts gets exhausted.

A special case of this algorithm, with $T=1$, is proposed in [LIVN82].

4.3.3 Algorithm 3

In this algorithm, the control part of the algorithm gets activated at regular intervals [STA84B]. It computes the difference between its own "busyness" (represented by its load) and the busyness of the *least* busy host. If the difference is greater than a bias BT , it initiates a job transfer to that host, otherwise no job is moved.

Besides BT , this algorithm has another parameter Tw which represents a time-window. Whenever a job is transferred to a host, say $HOSTA$, the time of the transfer is recorded by the control element. If at any time during the window period, the control element determines that $HOSTA$ is again the least busy host, no job transfer is done. The window mechanism is used for stability.

The information element of the algorithm is responsible for estimating the busyness of all other hosts and for informing other hosts about the busyness of its host. This can be done at every status change at its host or at regular intervals of time.

4.3.4 Algorithm 4

This algorithm, called the "broadcast idle" algorithm, uses a receiver-initiated policy [LIVN82].

When a host becomes idle, it broadcasts a message requesting a job from the other hosts. On receiving this message, the other hosts (any one of which can potentially fulfill the request) activate the control element of the algorithm. The control element terminates immediately if the

load at its host is not greater than 1. If the load is more than 1, it waits for a length of time equal to D (a parameter of the algorithm) divided by the local load. The more heavily loaded a host is, the lesser will be the duration of its wait.

When the wait is over, the control element checks to see if some other host has made a "broadcast reservation" in response to the "broadcast idle" request. If it finds that a reservation has been made already, it terminates; otherwise it broadcasts a reservation message itself and waits for a response from the host that was idle. The response can be positive or negative depending on whether that host is still idle or not. If the response is positive, it transfers a job to the host. If the response is negative, it does not transfer a job.

There is no explicit information element in this policy.

4.3.5 Algorithm 5

This algorithm uses both sender and receiver initiated strategies for load sharing. It is based on the "Symmetric" algorithm proposed in [MIRC87], and is a combination of Algorithms 1 and 2. The information element at each host responds to load queries as in Algorithms 1 and 2.

The control element gets activated when the local load becomes more than the sender-threshold(ST) or less than the receiver-threshold(RT). A random subset of L_p nodes is probed in either case and a job transfer decision is made depending on which threshold was crossed at the local host. The criteria for the decision are the same as described for Algorithms 1 and 2. The algorithm terminates when a candidate host is found or the probe limit L_p is exceeded.

5 Experimental evaluation

In this section, some of the experiments done using the testbed are described.

5.1 Environment

The experimental environment has the following characteristics:

- Four local hosts are used in all experiments and the load distribution among the hosts is given by a 4-character representation. (For example, a distribution of *VHML* means that the loads are Very Heavy, Heavy, Moderate and Light at hosts 1, 2, 3 and 4 respectively).
- The experiments are done in a controlled environment i.e. only testbed processes are allowed to run at the control and local hosts.
- Each experiment consists of several 35-minute *runs* of one or more load sharing algorithms. The first 5 minutes of data in the log-files is ignored during statistics generation.

These time-durations are determined using trial runs. First we determine the time taken by the system to reach steady-state: for a trial run lasting 35 minutes, statistics are generated after dropping the first 2, 5, 10 and 15 minutes of data. The results show that the system reaches steady-state after the first 5 minutes. Next, we decide upon the minimum duration of an experiment: in runs lasting 15, 25, 35, 45 and 55 minutes, it is observed that the results for the 15, 25 and 35 minute runs do not match. However, when the duration is increased beyond 35 minutes, the results stay the same.

- Fault tolerance is switched on during all the experiments.

- The mean service time for the jobs at a local host is the sum of:
 1. the mean time for computation (10 seconds).
 2. the mean time for job duplication (done for fault tolerance), which varies with job size.
- The mean inter-arrival time at each local host is computed using the mean service time and the traffic intensity ρ at the host. ρ is 0.1, 0.6, 0.8 or 0.9 for Light(L), Medium(M), Heavy(H) and Very heavy(V) loads, respectively. The value of ρ is derived using the load specification received from the control host at the beginning of an experiment.
- Both the inter-arrival time and the service-time are generated from an exponential distribution. The impact of this assumption is examined in Experiment 4 where we vary the service-time distribution to observe its effect on load sharing.
- The degree of multiprogramming for the Long Term Scheduler (LTS) is 1 for all experiments because all the jobs are CPU bound; increasing it beyond 1 should not affect the expected response time. Thus, the scheduling policy is First-Come First-Served (FCFS).

5.2 Format of result tables

The results for the experiments are presented in tables. All the tables use the same format. Each row corresponds to a 30-minute run of a load-sharing experiment. The load distribution among the local hosts for all the rows of a table is given in parentheses along with the title of the table.

The entries in the table are explained below:

- **Alg #:** Contains the algorithm number. Algorithm number 0 corresponds to no load sharing — the *base case* against which other algorithms are compared. Algorithm numbers 1 to 5 correspond to the load sharing algorithms described in Chapter 4.
- **Params:** Contains the parameters used for a run of an algorithm. The parameters are specific to the algorithm. (e.g. the value of job threshold and probe-limit for algorithms 1 and 2, , the frequency of scheduling in algorithm 3, the value of D in algorithm 4 etc.)
- **Benefit:** Contains the benefit obtained in a run of an algorithm:
 - **Resp.Time:** The mean response time for all the jobs that were executed at all the local hosts.
 - **%Impvmt:** The percentage improvement in the mean response time as compared to the base case (mean response time for algorithm 0).
 - **Variance:** The variance of response time of all the jobs that were executed at all the local hosts.
 - **%Impvmt:** The percentage improvement in variance as compared to the base case (variance for Algorithm 0).
- **Cost:** Contains the cost incurred in a run of an algorithm:
 - **%JMvt:** Represents the percentage of job movement i.e. the total number of job transfers as a percentage of the total number of jobs executed.

- **%BadDcs:** Represents the percentage of bad decisions i.e. the total number of bad decisions as a percentage of the total number of job transfer decisions. A bad decision means that the load at the destination host is higher than the load at the source host in a job transfer.
- **Number of Msgs:** The number of messages exchanged for load sharing.
- **Exec.Cost:** The total CPU time (in seconds) used by the Load Agent (LA) and Load Balancer (LB) processes during execution at all the local hosts.

5.3 Experiment 1: Performance of all the algorithms

Experiment description

All the algorithms are executed for five different load distributions at the local hosts — one where the average loads at the host are different(VHML), and the rest with the same average load at all the hosts(LLLL,MMMM,HHHH and VVVV).

The parameters of the algorithms are tuned for each run in two stages — first to achieve stability, and then to get the best cost-benefit tradeoff.

It should be noted that for this experiment, the information element of algorithm 3 does not do periodic load exchange — it does a load broadcast for every status change at a local host.

Analysis of results

Tables 1 through 5 depict the results for this experiment. The tuned values of the parameters for each run are also given.

When the average loads at the hosts are unequal (Table 1), the improvements in the mean response time for algorithms 1 through 5 range from 43% to 58%. The improvements in the variances are between 78% and 87%.

When all the hosts have the same average load, the benefits are not as good compared to when their average loads are different. However, the benefits at higher loads (HHHH and VVVV) are better than those at lower loads (LLLL and MMMM). The increase in benefit at higher loads is accompanied by an increase in cost.

Of all the algorithms, algorithms 1 and 5 provide the best benefits at all the loads. Algorithm 5 has a much higher cost of messages and computation as compared to algorithm 1 making algorithm 1 the best load sharing algorithm over a wide range of loads.

Algorithms 2 and 4 give comparable benefits when the average load at the hosts are unequal or all the hosts are lightly loaded. As the load at all the hosts keeps increasing, algorithm 2 causes more job movement than algorithm 4. As a result, it provides greater benefit. However, algorithm 2 also becomes more expensive than algorithm 4, especially when all the hosts are heavily loaded. For example, when the load distribution is VVVV (Table 5), algorithm 2 exchanges 5 times as many messages as algorithm 4.

The percentage of bad job transfer decisions is always 0 for algorithm 3. This implies that it makes good transfer decisions. This is to be expected because of its load information exchange policy: load is exchanged after every status change at a local host. However, the high rate of load exchange results in a higher message cost than other algorithms. When the average loads at the hosts are unequal, its performance benefit is marginally better than that of Algorithm 2 or 4, but its cost is much higher. When the average load at all the hosts is the same, algorithm 3 performs very poorly with increasing load, giving the worst benefit for the highest cost.

Alg #	Params	Benefit				Cost			
		Mean Response Time		Variance of Resp. Time		Job Movement		Msgs and Computation	
		Resp Time (secs)	%Imprmnt	Variance (secs) ²	%Imprmnt	%MMvt	%RedDcs	Number of Msgs	Exec Cost (secs)
0	---	43.91	---	1439.06	---	---	---	---	---
1	T=2; Lp=1	18.88	57.00	180.71	87.44	19.54	2.63	649	13.04
2	T=2; Lp=2	24.81	43.50	315.44	78.08	18.04	4.29	1638	24.40
3	LBf=10; BT=3; Tw=10;	23.31	46.91	251.53	82.52	12.82	0.00	2216	44.57
4	D=4	24.23	44.82	289.71	79.87	18.72	6.85	1121	24.44
5	ST=2; RT=1; Lp=2;	18.39	58.12	179.10	87.55	24.88	8.25	1940	27.77

Table 1 : Performance of all the algorithms (Load = VHML)

Alg #	Params	Benefit				Cost			
		Mean Response Time		Variance of Resp. Time		Job Movement		Msgs and Computation	
		Resp Time (secs)	%Imprmnt	Variance (secs) ²	%Imprmnt	%MMvt	%RedDcs	Number of Msgs	Exec Cost (secs)
0	---	14.11	---	197.88	---	---	---	---	---
1	T=1; Lp=2	12.70	9.99	152.02	23.18	3.70	0.00	21	4.70
2	T=1; Lp=2	13.63	3.40	172.72	12.71	1.85	0.00	338	7.13
3	LBf=10; BT=1; Tw=10;	12.85	8.93	156.50	20.91	5.56	0.00	304	8.81
4	D=2	13.65	3.26	173.38	12.38	1.85	0.00	141	5.11
5	ST=1; RT=1; Lp=2;	12.69	10.06	149.18	24.61	3.70	0.00	363	7.80

Table 2 : Performance of all the algorithms (Load = LLLL)

Alg #	Params	Benefit				Cost			
		Mean Response Time		Variance of Resp Time		Job Movement		Migs and Computation	
		Resp Time (secs)	%Impmnt	Variance (secs) ²	%Impmnt	%Mht	%BadDcs	Number of Migs	Exec Cost (secs)
0	---	22.82	---	357.57	---	---	---	---	---
1	T=2;lp=2	16.73	26.69	172.20	51.84	23.67	10.20	914	17.99
2	T=2;lp=2	18.20	20.25	181.12	49.35	15.66	18.46	2163	30.13
3	LBf=10; BT=3; Tw=10;	20.28	11.13	213.00	40.43	4.81	0.00	2120	44.38
4	D=4	17.86	21.74	163.26	54.34	18.36	2.63	1260	26.23
5	ST=2; RT=1; lp=2;	17.92	21.47	193.13	45.99	17.30	1.39	1934	27.19

Table 3 : Performance of all the algorithms (Load = MMMMM)

Alg #	Params	Benefit				Cost			
		Mean Response Time		Variance of Resp Time		Job Movement		Migs and Computation	
		Resp Time (secs)	%Impmnt	Variance (secs) ²	%Impmnt	%Mht	%BadDcs	Number of Migs	Exec Cost (secs)
0	---	40.84	---	1290.68	---	---	---	---	---
1	T=2;lp=2	27.50	32.66	644.66	50.05	20.90	5.61	1450	24.04
2	T=2;lp=2	27.85	31.81	545.76	57.72	24.03	10.48	2253	33.13
3	LBf=10; BT=3; Tw=10;	32.16	21.25	678.50	47.43	9.11	0.00	2124	43.07
4	D=4	29.25	28.38	616.47	52.24	15.95	6.10	1223	26.44
5	ST=2; RT=1; lp=2;	27.54	32.57	586.47	54.56	22.92	5.93	2462	33.45

Table 4 : Performance of all the algorithms (Load = HHHH)

Alg #	Params	Benefit				Cost			
		Mean Response Time		Variance of Resp. Time		Job Movement		Migs and Computation	
		Resp Time (secs)	% Impmnt	Variance (secs) ²	% Impmnt	% Mvt	% BadDcs	Number of Migs	Exec Cost (secs)
0	---	68.90	---	2040.00	---	---	---	---	---
1	T=3; Lp=2	32.85	52.32	482.70	76.34	24.62	6.16	1621	27.75
2	T=3; Lp=2	33.89	50.81	436.24	78.62	29.90	14.12	2705	49.85
3	LBI=10; RT=5; T=10;	43.96	36.20	666.41	67.33	10.61	0.00	3183	63.49
4	D=8	41.58	39.65	717.26	64.84	8.77	1.92	698	18.55
5	ST=3; RT=3; Lp=2;	32.39	52.99	456.79	77.61	27.66	3.05	3961	49.03

Table 5 : Performance of all the algorithms (Load = VVVV)

Alg #	Params	Benefit				Cost			
		Mean Response Time		Variance of Resp. Time		Job Movement		Migs and Computation	
		Resp Time (secs)	% Impmnt	Variance (secs) ²	% Impmnt	% Mvt	% BadDcs	Number of Migs	Exec Cost (secs)
0	---	43.91	---	1439.06	---	---	---	---	---
2	No Timer T=2; Lp=2	24.81	43.50	315.44	78.08	18.04	4.29	1638	24.40
2	With Timer (freq=20) T=2; Lp=2	21.80	50.35	224.66	84.39	18.56	9.72	1410	24.87
4	No Timer D=4	24.23	44.82	289.71	79.87	18.72	6.85	1121	24.44
4	With Timer (freq=10) D=4	19.37	55.89	184.78	87.16	20.88	2.47	1146	28.86

Table 6 : Effect of timer-driven scheduling on receiver-initiated algorithms (Load = VHML)

5.4 Experiment 2: Timer-driven scheduling in receiver-initiated algorithms

Experiment description

In Table 1, the receiver-initiated algorithms (Algorithms 2 and 4) show 10% less improvement in mean response time as compared to algorithm 1. The control element in these algorithms (which does the network-wide job scheduling) is threshold-driven i.e. if the local load becomes less than the threshold T when the last job terminates, the control element is activated. If the local load stays below the threshold after that, the control element is not invoked again. If the scheduling is done periodically, the control element can be activated more frequently (not just after the last job terminates). This may result in better performance.

In this experiment, the control element in algorithms 2 and 4 is activated periodically using a timer. The frequency of the timer is tuned for both algorithms such that their costs do not increase too much beyond those in Table 1.

Analysis of results

Table 6 shows the improvement in performance obtained by introducing timer-driven scheduling in algorithms 2 and 4. Algorithms 2 and 4 give 7% and 10% further improvements in mean response time respectively, without extra message overhead.

The use of periodic scheduling in receiver-initiated algorithms is a modification to the published versions of these algorithms. The results of this experiment show that it can improve the performance significantly.

5.5 Experiment 3: Single-node failure

Experiment description

The fault-tolerance feature of the testbed is tested in this experiment. One of the local hosts is "killed" half-way through the experiment by the **KILLHOST** command of the **LBSHELL**. Due to fault tolerance, the jobs in the middle of their execution at the *victim* host get re-initiated at its cohost.

Analysis of results

Table 7 shows the results for this experiment. The response times are higher when a host fails because the jobs that get aborted at the *victim* host are restarted at the cohost. This causes a sudden increase in the load at the cohost and degrades the overall response time. The cost of messages and computation is less for single-node failure because no more jobs originate at the *victim* host after it fails.

The percentage of job movement goes up from 19.5% (in the no failure case) to 21.7% (with single node failure) because load sharing tries to take care of the sudden increase in the load at the cohost by transferring more jobs. The percentage improvement in response time decreases from 57% to 55%. The cost of messages and computation is not much affected.

The impact of failures on load sharing has not been discussed much in the literature. It depends to a large extent on the way fault tolerance is provided. The fault tolerance mechanism used in the testbed does not cause too much performance degradation under failures.

Alg #	Params	Benefit				Cost			
		Mean Response Time		Variance of Resp Time		Job Movement		Migs and Computation	
		Resp Time (secs)	% Impmnt	Variance (secs) ²	% Impmnt	% Mvt	% BadDcs	Number of Migs	Exec Cost (secs)
0	No failure	43.91	---	1439.06	---	---	---	---	---
1	No failure T = 2; Lp = 1	18.88	57.00	180.71	87.44	19.54	2.63	649	13.04
0	One node failure	47.14	---	1515.97	---	---	---	---	---
1	One node failure T = 2; Lp = 1	21.28	54.86	237.66	84.32	21.69	1.39	631	13.18

Table 7 : Effect of single-node failure (Load = VHML)

Alg #	Params	Benefit				Cost			
		Mean Response Time		Variance of Resp Time		Job Movement		Migs and Computation	
		Resp Time (secs)	% Impmnt	Variance (secs) ²	% Impmnt	% Mvt	% BadDcs	Number of Migs	Exec Cost (secs)
0	Exp Arrv Exp Svc	43.91	---	1439.06	---	---	---	---	---
1	Exp Arrv Exp Svc T = 2; Lp = 1	18.88	57.00	180.71	87.44	19.54	2.63	649	13.04
0	Exp Arrv Unif Svc	35.64	---	871.70	---	---	---	---	---
1	Exp Arrv Unif Svc T = 2; Lp = 1	17.57	50.70	88.64	89.83	18.56	4.17	654	13.29

Table 8 : Effect of variation in the service-time distribution of jobs (Load = VHML)

5.6 Experiment 4: Effect of service-time distribution

Experiment description

Load sharing exploits the variability in the service times of the jobs to improve performance. If the variability is higher, instantaneous load differences occur more frequently and more benefits can be obtained at the same cost.

In this experiment, the service times for the jobs are generated using a uniform distribution instead of an exponential distribution. The variance of an exponential distribution is higher than the variance of a uniform distribution with the same mean.

Analysis of results

The results of Experiment 4 are shown in Table 8.

The response time for algorithm 0 decreases from 43.91 seconds to 35.64 seconds when the service-time distribution changes from exponential to uniform. The decrease in the variance of response time is more significant — it drops from 1439.06 sec^2 to 871.70 sec^2 .

It is evident from the table that load sharing provides lesser benefit when the variance of the service time distribution is lower: the percentage improvement in response time is only 50.7% when the distribution is uniform as compared to 57% when it is exponential. The costs are comparable in both cases.

Most load sharing studies assume an exponential distribution for the service time. The effect of varying the service time distribution on load sharing performance has not received much attention in literature. The result of this experiment is significant because it provides insight on this aspect.

6 Summary and conclusions

This paper describes the implementation of a testbed for load sharing in a distributed system consisting of AT&T 3B2 minicomputers. The testbed provides a framework where a distributed load sharing algorithm can be implemented after being decomposed into the constituent information and control elements. It is designed to evaluate dynamic load sharing policies which use the process migration approach. It can be used to evaluate the performance of both sender and receiver initiated policies.

The testbed contains an optional fault tolerance feature to take care of single-node failure. A given load sharing algorithm can be made fault tolerant with minor modifications to its code.

The testbed provides a single user-interface to the distributed system. It has a statistics generation facility which uses data produced during an experiment to produce the benefits and costs of the algorithm being evaluated. These statistics can be used to compare the performance of different load sharing algorithms.

The usefulness of the testbed is illustrated by implementing five load sharing algorithms. Experiments are conducted to evaluate and compare these algorithms under different loading conditions and for various parameter values. These experiments confirm some previous results in load sharing:

- when the average load at all the hosts is the same, load sharing still provides significant gain in performance at moderate loads [LIVN82].
- if the frequency of scheduling is too low, the benefits from load sharing are small; if the frequency is increased, benefits improve but the costs also become higher [STA84B].

- decreasing the frequency of load information exchange results in poorer scheduling decisions: if the frequency falls below a threshold, load sharing becomes harmful [STA84B].
- tuning the parameters of an algorithm implicitly takes care of its stability [STAN85].
- sender-initiated policies perform better than receiver-initiated policies at low to moderate loads [EAG86A].
- network delays have an adverse effect on load sharing; the more the delay, the worse the performance [MIRC87].

The experiments also provide some new insights into the behavior of load sharing algorithms:

- sometimes the benefits of an algorithm are deceptively high; it is necessary to consider the costs to realistically determine the performance of the algorithm.
- the benefits of a receiver-initiated algorithm, in which scheduling is triggered by queue-threshold, can be significantly improved by introducing periodic scheduling. The increase in benefits is obtained with very little increase in cost.
- fault tolerance, if done carefully, does not cause too much degradation in the performance of a load sharing policy.
- load sharing exploits the variability in the service times of jobs to improve performance: the greater the variance in the service times, the more the gain from load sharing.

References

- [ATT84A] *Advance Printing, AT&T 3B2 Computer AT&T 3BNET Manual*, October 1984.
- [ATT84B] *AT&T 3B2 Computer UNIX^R System V Release 2.0 Inter-Process Communication Utilities Guide*, October 1984.
- [ATT84C] *AT&T 3B2 Computer UNIX^R System V Release 2.0 Programmer Reference Manual*, October 1984.
- [BARA85] Amnon Barak and Amnon Shiloh, A distributed load balancing policy for a multicomputer, *Software-Practice and Experience*, September 1985.
- [BERS85] Brian Bershad, Load balancing with Maitre d', *Tech. Report UCB/CSD 86/276, Computer Science Division, University of California, Berkeley*, December 1985.
- [CASA86] T.L. Casavant and J.G. Kuhl, A formal model of distributed decision-making and its application to distributed load balancing, *Proc. International Conf. on Distributed Computing Systems*, May 1986.
- [CHOU83] Timothy C.K. Chou and Jacob A. Abraham, Load redistribution under failure in a distributed system, *IEEE Transactions on Computers*, September 1983 .
- [CHOW79] Yuan-Chieh Chow and Walter H. Kohler, Models for dynamic load balancing in a heterogeneous multiple processor system, *IEEE Transactions on Computers*, May 1979.

- [DIKS87] Piyush Dikshit, SAHAYOG: A testbed for fault-tolerant load sharing in a distributed system, *Master's thesis, Department of Computer Science, University of Maryland at College Park*, June 1987.
- [EAG86A] Derek L. Eager, Edward D. Lazowska, John Zahorajan, A comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing, *Performance Evaluation* 6, 1986.
- [EAG86B] Derek L. Eager, Edward D. Lazowska, John Zahorajan, Adaptive load sharing in homogeneous distributed systems, *IEEE Transactions on Software Engineering*, May 1986.
- [EZZA86] A. Ezzat, Load Balancing in NEST: A Network of Workstations, *Proceedings 1986 Fall Joint Computer Conference, Dallas, Texas*, November 1986.
- [FERR86] D. Ferrari and S. Zhou, A load index for dynamic load balancing, *Proceedings 1986 Fall Joint Computer Conference, Dallas, Texas*, November 1986.
- [HAC 86] Anna Hac and Xiaowei Jin, Dynamic load balancing in a distributed system using a sender-initiated algorithm, *Unpublished Report*, 1986
- [HWAN82] K. Hwang, W. Croft, G. Goble, B. Wah, F. Briggs, W. Simmons and C. Coates, A UNIX-based local computer network with load balancing, *IEEE Computer*, April 1982.
- [KORR86] Richard Korry, A Load Sharing Algorithm for a workstation environment, *Technical Report 86-10-03, Department of Computer Science, University of Washington, Seattle*, October 1986.
- [LELA86] W. Leland and T. Ott, Load balancing heuristics and process behavior, *ACM SIG-METRICS Conference on measurement and modeling of computer systems*, May 1986.
- [LIVN82] M. Livny and M. Melman, Load Balancing in Homogeneous Broadcast Distributed Systems, *ACM Computer Network Performance Symposium*, April 1982.
- [MIRC87] R. Mirchandaney, Don Towsley and John A. Stankovic, Analysis of the effect of delays on load sharing, *Unpublished Report*, February 1987.
- [NI 82] Lionel M. Ni, A Distributed Load Balancing Algorithm for point-to-point local computer networks, *Proceedings Computer Networks CompCon(Pages 116-123)*, September 1982.
- [NI 85] Lionel M. Ni, Chong-Wei Xu and Thomas B. Gendreau, A Distributed Drafting Algorithm for Load Balancing, *IEEE Transactions on Software Engineering*, October 1985.
- [PATN86] Lalit M. Patnaik and Kailasam V. Iyer, Load-leveling in fault-tolerant distributed computing systems, *IEEE Transactions on Software Engineering*, April 1986.
- [STA84A] John A. Stankovic, A perspective on Distributed Computer Systems, *IEEE Transactions on Computers*, December 1984.
- [STA84B] John A. Stankovic, Simulation of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms, *Computer Networks* 8, 1984.
- [STAN85] John A. Stankovic, Stability and distributed scheduling algorithms, *IEEE Transactions on Software Engineering*, October 1985.

- [STON78] Harold S. Stone, Critical load factors in two-processor distributed systems, *IEEE Transactions on Software Engineering*, May 1978.
- [TRIP85] Satish K. Tripathi, David Finkel and Erol Gelenbe, Load Sharing in Distributed Systems with Failures, *ISEM Research Report 30, Universite de Paris-Sud, Orsay, France*, March 1985.
- [WANG85] Yung-Terng Wang and Robert J.T. Morris, Load Sharing in Distributed Systems, *IEEE Transactions on Computers*, March 1985.
- [ZHOU86] Songnian Zhou and Domenico Ferrari, An experimental study of load balancing performance, *Unpublished Report*, 1986.
- [ZHOU86] Songnian Zhou, A Trace Driven Simulation Study of Dynamic Load Balancing, (*Submitted for publication*), 1986.

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b RESTRICTIVE MARKINGS N/A	
2a SECURITY CLASSIFICATION AUTHORITY N/A			3 DISTRIBUTION/AVAILABILITY OF REPORT approved for public release; distribution unlimited	
2b DECLASSIFICATION/DOWNGRADING SCHEDULE N/A				
4 PERFORMING ORGANIZATION REPORT NUMBER(S) CS-TR-1891; UMIACS-TR-87-34			5 MONITORING ORGANIZATION REPORT NUMBER(S)	
6a NAME OF PERFORMING ORGANIZATION University of Maryland		6b OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research	
6c ADDRESS (City, State, and ZIP Code) Department of Computer Science University of Maryland College Park, MD 20742			7b. ADDRESS (City, State, and ZIP Code) 800 North Quincy Street Arlington, VA 22217-5000	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-87-K-0124	
8c. ADDRESS (City, State, and ZIP Code)			10 SOURCE OF FUNDING NUMBERS	
			PROGRAM ELEMENT NO.	PROJECT NO.
11 TITLE (Include Security Classification) SAHAYOG: A testbed for load sharing under failures				
12 PERSONAL AUTHOR(S) Piyush Dikshit, Satish K. Tripathi				
13a. TYPE OF REPORT Technical		13b. TIME COVERED FROM TO		14. DATE OF REPORT (Year, Month, Day) June 1986
15 PAGE COUNT 34				
16 SUPPLEMENTARY NOTATION				
17 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP		
19 ABSTRACT (Continue on reverse if necessary and identify by block number) This paper describes the implementation of a testbed for distributed load sharing in a broadcast local area network consisting of AT&T 3B2 minicomputers. The testbed is designed to evaluate dynamic load sharing policies which use the process migration approach. It contains an optimal fault-tolerance feature to handle single-node failures. Five load sharing algorithms are implemented and evaluated using the testbed under different loading conditions and for various parameter values. The experiments done using these algorithms confirm some earlier results about load sharing and also provide some new insights.				
20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL			22b TELEPHONE (Include Area Code)	22c OFFICE SYMBOL

END

DATE

FILM

JAN
1988