



Direct execution simulation of load balancing algorithms with real workload distribution

Jiannong Cao^{a,*}, Graeme Bennett^b, Kang Zhang^c

^a Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

^b Centre for Information Technology Research, University of Queensland, Brisbane, Queensland 4072, Australia

^c Department of Computing, Macquarie University, Sydney, NSW 2109 Australia

Received 27 January 1999; received in revised form 13 May 1999; accepted 10 August 1999

Abstract

This paper describes the model and implementation of a distributed direct execution simulation study of load balancing algorithms for a workstation-based locally distributed system. A general simulation system for load balancing algorithms is constructed on a local area network of DEC workstations, which directly executes the codes of the load balancing algorithms but simulates the underlying network and system environment. Using the simulation system, simulations with real workload distribution are conducted. Traces of user workstation activity collected in a university department environment are used in the simulation runs. We describe the methods used for distributed direct execution simulation of load balancing algorithms. The design of our simulations of several load balancing algorithms is presented and the simulation results are discussed. © 2000 Elsevier Science Inc. All rights reserved.

Keywords: Distributed systems; Load balancing; Distributed direct execution simulation; Trace-driven simulation

1. Introduction

A distributed system consists of a number of processors connected by a communication network. There are many kinds of distributed systems. These systems may differ in the ways in which the processors are connected, communicate and coordinate with each other. They can also be characterized by the applications best supported by them. A *workstation-based locally distributed system* consists of a collection of workstations interconnected by high speed local area networks such as *Ethernet*.

Users of a distributed system can effectively share the information and the resources in the system. More importantly, a distributed system has the promise of improving the performance of a system, as the users have access to the computational power of more than a single processor. When viewed as a whole, the system represents a significant computing resource. However, the entire system potential is not being fully explored. In a workstation-based system, for example, most machines are autonomous personal workstations, each dedicated

primarily to serving its local user; often, many of the machines are idle while others are overloaded (Litzkow et al., 1988). A load balancing mechanism can be used to spread the workload evenly across the distributed system. When coupled with a load balancing algorithm, task scheduling can be made to increase the performance of the system. The idle machines will be used to perform tasks for other users who need more computational capability than their own workstation provides.

In an attempt to design a task scheduling mechanism for a workstation-based locally distributed computing environment, we face the problem of choosing a proper scheduling scheme. We regard load balancing as essential to the task scheduling mechanism. The choice of a load balancing algorithm is a crucial design decision (Wang and Morris, 1985). Previous work on performance evaluation of load balancing algorithms for distributed systems emphasizes on two approaches: theoretical analysis and simulation. Theoretical analysis involves using mathematical techniques such as queuing models and Markov chain models to model and study the performance of task scheduling algorithms (Eager et al., 1986b; Chow and Kohler, 1979; Mirc-handaney et al., 1989; Tripathi and Menasce, 1992). Alternatively, simulation has been proposed as a viable

* Corresponding author. Tel.: +852-2766-7275; fax: +852-2774-0842.
E-mail address: csjcao@comp.polyu.edu.hk (J. Cao).

method for analyzing the complex nature of the *dynamic* aspects of these algorithms (Krueger and Livny, 1988; Svensson, 1990; Johnson and Harget, 1989).

This paper is concerned with the development of simulation systems for load balancing algorithms which are considered suitable for a workstation-based environment. We have developed a software tool for simulating load balancing algorithms on a local area network of DEC workstations. In this paper, we shall describe the application of the tool to the development of simulations of four simple, dynamic load balancing algorithms, including a new algorithm that enhances one of the previous existing algorithms. Performances of these algorithms are compared using the measures of mean job response time and system throughput. The same set of algorithms have been studied in (Eager et al., 1986b) by using Markov models and solutions. They used only mean response time as the performance measure. Our simulation results agree with their observations to a great extent.

Simulation studies have been done for different load balancing algorithms under different system environments, using different techniques. Our work is distinguished from the previous work in several ways. First, the results gained from many of the previous simulations may not be considered to be directly applicable to the workstation-based distributed system, as they either only consider general distributed systems and do not address the specifics that relate to workstation-based distributed systems, or they have used queueing network models and simulation with probabilistic assumptions about job arrivals and resource demands (Livny and Melman, 1982; Chow and Kohler, 1979; Eager et al., 1986b; Shen, 1988). Second, the simulation techniques used in our study are different from those used in the existing work. Most of the previous studies use a discrete-event driven sequential simulator with synthesized simulation parameters. Our system employs the distributed direct execution simulation using collected traces of user workstation in a real-life environment. Direct execution simulation offers the potential for accurate prediction of distributed program performance on real systems, while parallelizing the simulation improves the simulation speed (Covington et al., 1991; Dickens et al., 1996) and simplifies synchronization which is inherent in distributed task scheduling (see Section 4).

Simulation with real workload distribution has been used by a number of researchers to evaluate performance of load balancing algorithms. The simulations use measured traces of actual job processing, instead of generating a synthetic workload which may not be realistic. In Svensson (1990), a trace-driven simulation was used to evaluate how a decentralized algorithm called Shortest and a centralized algorithm called Central are affected by different filters. The study, however, did not

compare the relative performance of different algorithms. Zhou (1988) performed an extensive trace-driven simulation study of (both central and distributed) dynamic load balancing algorithms. The algorithms Random, Threshold, and JSQ are identical to the ones studied by him. In addition to comparing the performances of the algorithms, he also studied the effects on load balancing performance by different factors and the behavior of the system under load balancing. However, in his study, sessions of job traces were collected on the same host, a VAX-11/780 machine. Although in the simulation runs each host is fed with the trace of a different session, the possible correlations between the loads of the various hosts are lost. Also, he used a sequential event-driven simulator and had to explicitly model the overhead cost of computing the load information by the algorithms.

The rest of this paper is organized as follows. In Section 2, we first describe the underlying system environment and our design considerations of the task scheduling mechanism. In Section 3, we outline the load balancing algorithms examined in this paper. Section 4 presents the design of the simulation model for evaluating performance of selected load balancing algorithms. In Section 5, we describe the experiment design and discuss the simulation results. Section 6 concludes the paper.

2. Task scheduling in a workstation-based locally distributed system

The distributed system studied in this paper is a *homogeneous, workstation-based locally distributed system* in which the processors are identical personal workstations. Application programs may arrive and run correctly at any of the workstations in the system. Such a distributed system has certain peculiarities that should be recognized and exploited by the scheduling strategy.

In a distributed system, *task scheduling* is referred to as the act of determining which of the tasks execute on a particular processor, and how long the processor is granted (Casavant and Kuhl, 1988). A program segment that is not partitioned further is called a *process*, and is counted as one *unit of computation*. A *task*, on the other hand, is one *unit of allocation*. Processes are grouped together to form a task in order to reduce the overhead due to intensive interprocess communication. An application program in a distributed system that consists of a collection of tasks, possibly with some precedence constraints. There are two primary sources of information as input to a scheduler: the characteristics of the underlying system within which the tasks and the resources reside, and the attributes of and constraints on the tasks. Based on the information, a scheduling

algorithm defines a set of rules by which decisions can be made about how and when tasks are scheduled.

Scheduling in a distributed system can be broadly divided into two types: *job level scheduling* and *task level scheduling*. Job level scheduling treats each application as a unit of scheduling. In this case, the scheduler usually has no information on the structure of the job. Task level scheduling involves more considerations such as task decomposition, precedence relations, and clustering. The scheduler has knowledge of the structure of the job, e.g., in the form of a task graph, as well as timing information on each of the individual tasks. Job level scheduling can be thought of as a special case of task level scheduling, where all the processes of the job are grouped into a single task. Scheduling of the set of tasks which are considered as being independent can also be treated as job level scheduling. Our first step in developing a task scheduling facility for the specified system is to design a mechanism which will allow background jobs to be scheduled to idle workstations. Therefore, in this paper, we are actually dealing with job-level scheduling. By the object-oriented techniques used in our development, which enables us to separate the mechanism from the policies, task-level scheduling can be added on to the task scheduling facility at a later stage. Nevertheless, the results presented in this paper will be valuable for designers of systems that support remote execution of jobs and independent tasks (Asawa and Misra, 1991; Waldspurger et al., 1992; Litzkow et al., 1988; Shivaratri and Krueger, 1990).

Task scheduling can be either *preemptive* or *nonpreemptive*. In preemptive task scheduling, a task can be reallocated (migrated) after its execution starts. Task migration can be used to redistribute tasks for better performance improvement and error recovery, but is much more difficult than nonpreemptive scheduling, mainly because of the complexity of the preemptive task migration mechanism. We will not consider task migration in this paper.

A task scheduling policy consists of three parts: an *information policy* that determines how system state information vital to decision making is spread throughout the system, a *transfer policy* that determines whether to process a task locally or remotely, and a *location policy* that determines to which site a task selected for transfer should be sent. Task scheduling can be either *static* or *dynamic*, depending on the use of system state information. While static algorithms make no use of such information (Lo, 1984), dynamic algorithms use such information to improve individual job transfer decisions (Lin and Raghavendra, 1992; Shen, 1988; Chow and Kohler, 1979; Mehra and Wah, 1992). Static algorithms are only appropriate for systems where there is little variation in the system state. Dynamic algorithms that perform better for systems under a dynamic and unpredictable state change. Therefore, dynamic task

scheduling algorithms suit better to workstation-based locally distributed systems.

A task scheduling mechanism in a distributed system can use either *centralized* or *distributed* control (Wang and Morris, 1985). In a centralized approach, the system state information is collected at one site, where all scheduling decisions are made. The processing cost of collecting state information may be too expensive. Also, if the central server site fails, the entire system goes down. This approach, however, leads to simple control and is less complicated to implement. In a distributed approach, sites have more autonomy since each site makes its own scheduling decisions. Distributed control is more scalable to larger architectures and is usually more reliable. In this paper, we will examine dynamic task scheduling schemes using distributed control.

There are several driving forces that govern task scheduling, the two most important ones are computation speedup and system performance improvement. Load balancing aims at improving system performance by providing better utilization of all resources in the distributed system, equalizing the loads of sites in the systems. The main purpose of this paper is to describe the simulation of load balancing algorithms for our system environment.

3. Load balancing algorithms

Load balancing have been extensively studied in the literature, and many algorithms have been proposed (Shen, 1988; Wang and Morris, 1985; Chou and Abraham, 1982; Theimer and Lantz, 1989). While there are numerous load balancing algorithms, many seem to be very similar and are only variations on a theme. Furthermore, environments assumed by many of the algorithms have different features on task scheduling than that of a workstation-based locally distributed system. For example, many studies of load balancing algorithms assume that all the sites in the system are subject to the same average arrival rate of tasks, i.e., over the long-term, the external load imposed on each site is the same. This is true in an ordinary distributed system, where CPU-intensive, long running applications take up all available system resources most of the time and keep the workload uniformly high. In a workstation-based locally distributed system, however, many machines are frequently idle for long periods of time, and the workload can change rapidly and is rarely uniform. Consequently, many load balancing algorithms, such as those based on interactions with “nearest neighbors” of a host and static averages to determine their selections are considered inappropriate for our environment.

We have reviewed some representatives of load balancing algorithms found in the literature and in practical systems (Litzkow et al., 1988; Wang et al., 1992;

Cheriton, 1988; Casavant and Kuhl, 1988; Hudak and Goldberg, 1984). Based on the assumptions we have made on the underlying system and the design considerations of the task scheduling mechanism, three load balancing strategies were chosen to be examined. These strategies are dynamic in nature, use very small amounts of system state information, and easy to implement in a homogeneous distributed system (Eager et al., 1986b).

The three strategies have identical transfer policies, but differ in their location policies. The transfer policy is a *sender-initiated, threshold* policy: a newly received task at a site is accepted for processing there if and only if the number of tasks already in service or waiting for service is less than some threshold T . Otherwise, an attempt is made to transfer the task to another site. The sender-initiated scheme offers better performance when an initial job placement is used (Eager et al., 1986a).

The three location policies are listed below, in the order of increasing amounts of state information acquired.

- *Random placement*: If the local load is above threshold, a destination site is selected at random and the task is transferred to that site. The destination site treats the task just as a task originating at the site.
- *Threshold*: If the local load is above threshold, a destination site is selected at random and probed to determine whether the transfer of a task to that site would place it above threshold. If no, the task is transferred.
- *Join shortest queue (JSQ)*: If the local load is above threshold, a collection of destination sites are selected at random and polled to determine their load levels. The task is transferred to a node with the lightest load, unless that load is above the threshold, in which case the originating node must process the task.

When nodes are about equally loaded, it makes no performance improvement to remotely execute jobs, but probably a performance decreases, due to the overhead of transferring the jobs to the remote site. We have developed a new algorithm, *BJSQ*, to enhance the JSQ algorithm by introducing a new concept called *balance factor*. In the enhanced algorithm, if the difference in load between the sender and the receiver is less than a certain balance factor, then the job will not be sent to the remote site for execution. The balanced factor is therefore used to determine the degree to which sites in the system must be unbalanced before remote execution can occur. It has been shown in simulation results to achieve slightly better performance than the JSQ algorithm.

We investigate the relative performance of the above four algorithms. The algorithm, called Normal, which executes all jobs locally and does not perform any load balancing, is also evaluated. We use the algorithm to determine the degree to which load balancing improves the system performance. It can be seen from the simu-

lation results that all the four load balancing algorithms make significant improvement in system throughput and job response time.

4. The simulation system

Our simulation system, called *DiMSIM* (distributed machine simulation), is a distributed discrete event driven simulator, which uses and simulates the underlying operating environment. The code of a distributed task scheduler is directly executed by the simulator and used as a driver, describing the activities to be simulated. The simulation system employs distributed computing in itself so that the parallelism embedded in algorithms under simulation can be exploited in a natural way. Ideally, such a simulator can be designed to accurately model the communication cost, considering real system statistics such as Ethernet speed, and algorithm overheads. Another point is, in contrast to a single-node event driven simulator, parallel simulators do not have to cope with event ordering problems, since the simulation bears resemblance to the actual concurrent program executions running at the same time on different sites.

DiMSIM is built on a local area network of DEC-stations. Fig. 1 shows the model of a site in the system. The jobs are initially stored in a waiting queue. The scheduler takes the jobs off the waiting queue and may decide either to put them onto the local ready queue, where they are executed, or to send the job to another site for remote execution. In a real system, the CPU represents the mechanism of decay of the job's burst time. In the simulation system, all that the jobs must do is start and finish. Therefore, we can model the CPU as a simple mechanism, where a process arrives, its burst time decays and then it finishes execution. The ready queue is used to model multiprogramming. All the jobs in the ready queue are decaying simultaneously (approximating a round-robin CPU scheduling policy).

The simulation program is replicated and run on each site. When the program starts, the main routine, called

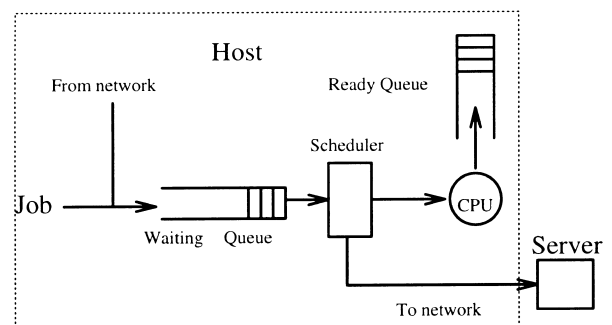


Fig. 1. A single node in the system.

Host, forks a process, called the *Server* which connects to all other sites in the system (see Fig. 2). The *Server* is responsible for sending or collecting messages and pass them to the local host or remote servers. The *Host* is the process which performs task scheduling simulation. It invokes a routine called *Simulation()*, which implements a particular load balancing algorithm.

A communication subsystem using Unix sockets is written to support the simulation system. The *Host* is set to generate an interrupt when it gets a message from the server on its socket. The interrupt handler intercepts the message and decides what to do with the message. For example, a *LOAD* message might be received from a certain node which asks for the local host's load information. This will result in the sending of load information to the requested site.

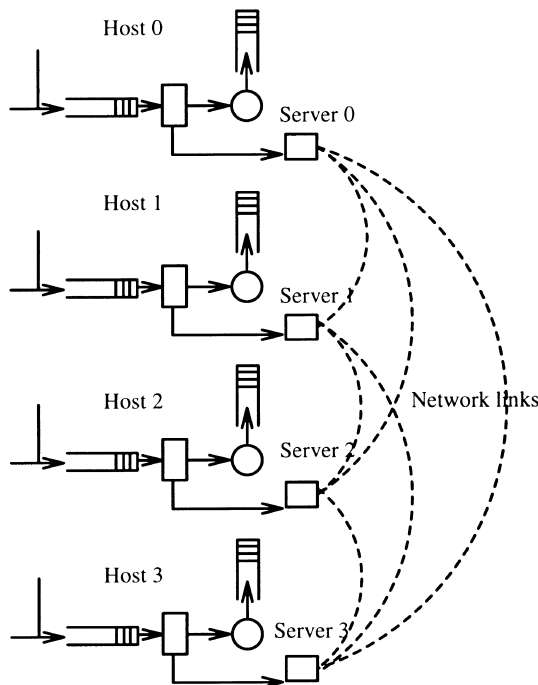


Fig. 2. The distributed simulation system.

A graphical tool is also connected into the system to monitor the simulation. Currently, the monitoring tool shows task arrivals at each site and load distribution profile for each site (see Fig. 3). Two programs were written to monitor the operation of the simulations. The programs communicate with the simulation system and use the X-Windows system for the graphics display.

5. Simulation design and results

In our context of simulation, a *job* is a program which runs for a period of time and then finishes. All the information necessary for a job is the arrival time and burst time. A job can be generated at a particular site or arrive from another site within the network. We assume that jobs are individually executable, logically independent of one another and can be processed at any site.

5.1. Generation of workloads

Workload can be described in terms of the cost factors in the cost function defined for the task scheduling mechanism. Although a complex cost function can be defined to incorporate CPU time, I/O operations and communications over network, we have adopted a simplified model, in which workload is measured in terms of the length of the ready queue. Queue length represents the number of jobs being executed on a site at a particular time. This model has been used also by many other papers (Zhou, 1988).

There are two methods to generate the workload: *probabilistic methods* and *real workload profiles*. The difference between the two methods is that jobs are characterized by accounting statistics or probability distributions. In most of the probabilistic workload methods, processes arrive at a site according to a Poisson distribution with a certain probability of arrival. It has been argued that the workload assumptions made in the probabilistic methods may not be realistic (Svensson, 1990). For this reason, in our simulation, we have

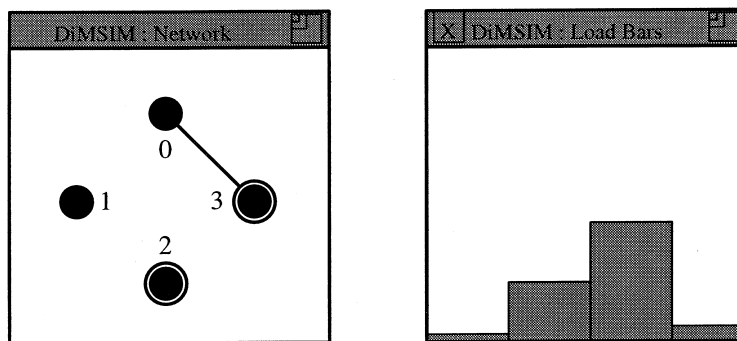


Fig. 3. The monitoring tool: node 0 is sending a job to node 3, while a job is being executed locally at node 2; the load bars correspond to the load on each node in order from left to right.

Table 1
Job list

Job name	File size
finger	126 976
more	42 089
ifmail	98 304
w	86 016
vi	172 032
groff	49 152
nn	405 504
elm	335 872
grep	24 576
ftp	167 936
ls	114 688
cp	28 672
mv	24 576

adopted the *trace-driven* approach, using job traces derived from previous accounting files to generate the workload. Our results thus can be used to verify theoretic workload models.

The accounting information is stored in log files of the system. Typical information contained within the files was system time, user time and real time used by a job, the user id, arrival time, finishing time, etc. A subset of the jobs, classified as “user jobs” or jobs executed from Unix shell, executed for the last six months were extracted from the system log files. These jobs represented jobs that would be sent to the task scheduler by the users. They defined a workload that consisted primarily of interactive and “fast-turnaround” applications, such as text editing, mail handling, and program development (compilation). Some of the jobs included in the subset file are listed in Table 1, together with the corresponding file sizes. Occasionally, however, a few users may run large, CPU-intensive computations. The total number of such applications running in the system varied considerably, depending on how many users are present and active at a given time.

A program, called *Accread* was written to load the raw accounting structure from the accounting log files, sorts the arrival times, and converts the data to a format used in the simulations. The data file generated contains CPU burst times, job names, and arrival times.

5.2. Modeling overhead cost

Sending a job for remote execution at another site induces a certain overhead. Two factors contribute to the overhead: computing and exchange of load information, and job transfer for remote execution. Overhead of computing load information is accounted for in the simulation by the load balancing algorithms execution, while the communication delay was accounted for by estimating the *transfer time* for passing the messages and the particular job across the Ethernet from the source site to the destination site.

Our system environment consists of workstations connected by a Ethernet which supports broadcast. A simple method for modeling the transfer time is to use a unit amount of time. However, many experiments show that the elementary cost can greatly depend on the length of the message. In our simulation, transfer time is defined to be proportional to the message size. It is calculated using the formula: $t = f \times r$, where t is the transfer time, f the size of the message or the job file, and r is the rate of transfer in seconds. We have used a constant for the value of r .

Our system has a network file server so that files do not have to be moved with the jobs to be transferred. The costs of accessing the program and data files, however, are still not the same for all the hosts since the host holding the files is also used for processing. We do not have measurement data on the contention at the file server and remote file accesses. Therefore, our cost assumptions are a rough approximation.

5.3. Performance measures

Performance measures used for the evaluation and comparison include *mean job response time* and *system throughput*. The response times of a job is determined by the amount of time the job spends queueing before it begins execution, i.e., the difference between the submission time and the starting time of that job. Throughput is defined as the number of jobs completed per unit of time.

The throughput figure is cumulative. Throughput is calculated in terms of waiting times of jobs, which depend on the workload of the system. Jobs executed on a site with a lighter load will run faster than if they were run on a heavily loaded site. Therefore, it is necessary to add a *waiting time* to the CPU burst time of each job to model variable execution time of a process depending on the current load of the host. The formula for calculating waiting time of a job i is as follows:

$$w_i = b_i + q_i,$$

where w_i is the waiting time, b_i the burst time and q_i is the queue length of the node, where the job is executed. The total waiting time of a job is also known as the turnaround time of the job, which is the total time spent on the ready queue, after it has been scheduled there for execution, not including response time.

System throughput is increased when jobs take less time to complete. Load balancing algorithms increase throughput by placing the jobs on nodes with lighter loads. However, there is not necessarily a direct correlation between system throughput increase and task response time decrease. Response time can sometimes have little or no noticeable effect on the throughput results. For example, this phenomenon may depend on the arrival rate of jobs in the system. If the turnaround time

of jobs is less than the average job inter-arrival time, the speedup factor of task execution gained by load balancing does not show up on the average throughput calculations. Since users are concerned with how long their jobs take to execute, we must also look at the response time results for more accurate performance comparisons.

A scheme was devised in order to allow for response times to be calculated for jobs that were migrated to a node, different from the original node, where the job was submitted to, with a different clock. It works as follows. When a job is submitted its start time is tagged with the job; when the job is migrated or executed, the response time is calculated. If the job is put onto the ready queue, the total response time is calculated and the information is pushed onto the ready queue along with the other job information. If, however, the job is to be sent to another node, the following steps take place. First, the time that the job has been waiting before being scheduled is calculated. This is the value for response time to be sent to the remote node. When the remote node receives the job, it subtracts the value from its own time clock, and pushes this information onto its waiting queue as the starting response time. When this job is finally executed on the ready queue the same process is repeated as if it had originated from the current node.

5.4. Simulation results

This section discusses the results obtained from various simulation runs of the selected load balancing algorithms. A complete simulation study would consist of simulation runs with various adjustable parameter values, such as system sizes, load levels, thresholds, etc. Comprehensive performance data could then be collected and plotted in various ways. Due to the limit of

space, in the following description of the results, four DECstations are used and the results are grouped into four different simulation runs: each run corresponds to a

Table 4
Job arrival and execution for Run 3

Algorithm	Job arrival	Scheduled for remote execution	Job executed
Normal	166	0	101
Random	166	151	116
JSQ	166	115	134
Threshold	166	13	135
BJSQ	166	63	135

Table 5
Job arrival and execution for Run 4

Algorithm	Job arrival	Scheduled for remote execution	Job executed
Normal	220	0	124
Random	220	200	120
JSQ	220	62	173
Threshold	220	6	173
BJSQ	220	62	173

Table 2
Job arrival and execution for Run 1

Algorithm	Job arrival	Scheduled for remote execution	Job executed
Normal	55	0	31
Random	55	44	40
JSQ	55	37	45
Threshold	55	43	45
BJSQ	55	42	45

Table 3
Job arrival and execution for Run 2

Algorithm	Job arrival	Scheduled for remote execution	Job executed
Normal	111	0	70
Random	111	97	82
JSQ	111	81	92
Threshold	111	67	72
BJSQ	111	61	92

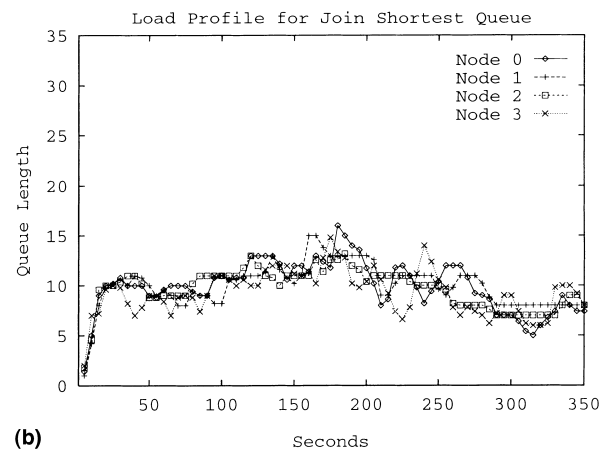
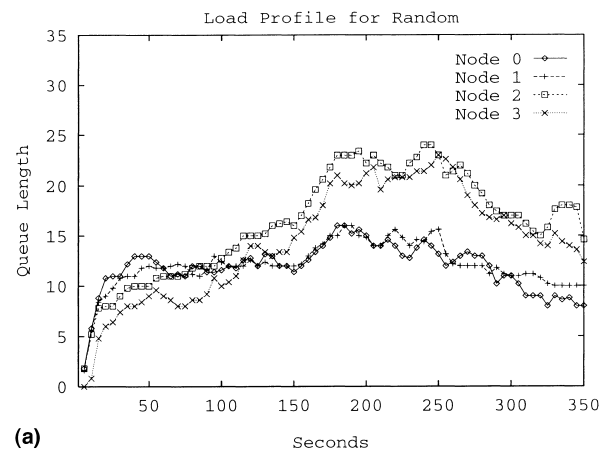


Fig. 4. (a) Load profile for Random in Run 3; (b) load profile for JSQ in Run 3.

particular system *workload characterization*, which is obtained from the system accounting files. Each workload pattern contained jobs which had been executed and finished execution during two week periods. Run 1 is performed using a workload pattern which represents a system where, most of the time, one site was overloaded, and three sites were idle. Run 2 used a workload pattern which represents a system, where half the machines are overloaded and half are idle. Run 3 is based on a workload pattern which represents a system, where three sites are overloaded and one site is idle, for most of the time. Finally, in Run 4, all the four sites were overloaded.

For each run, a table containing additional useful information about the simulation is presented (see Tables 2–5). The table contains the total number of jobs that *arrived* at all the nodes, the number of job arrivals that were *scheduled* on remote nodes for execution, and the total numbers of jobs that were actually *executed* by all the nodes in the system during the period of the simulation. The number of job arrivals can be compared with the number of job executions to see the proportion of jobs that finished execution during the run. The number of remote job executions can be used to see the

proportion of jobs that were scheduled remotely, so as to gauge the overhead of the different algorithms.

For each simulation run, the *workload profiles* are also collected. The profile shows for a particular algorithm the loading on each of the nodes in the simulation against time. It shows the effect of the algorithm on the workload distribution. Fig. 4 shows some examples of the profiles.

The results of the different algorithms on system performance are shown in Figs. 5 and 6. The average throughput of the system under the four algorithms and in the four cases are shown in Fig. 5.

In terms of throughput, in all the runs, JSQ, Threshold, and BJSQ all obtained the equal highest throughput value. In different cases, some of them achieved the same throughput result as the others with less jobs scheduled for remote execution and, therefore, with less overhead involved. In many situations, the Random algorithm, with no exchange of state information among the nodes, dramatically improves system throughput relative to Normal. However, in Run 4, Random did a poor job of balancing the load and it actually degraded the system performance. This can be seen in Table 4 and Fig. 5(d). Random attempted to

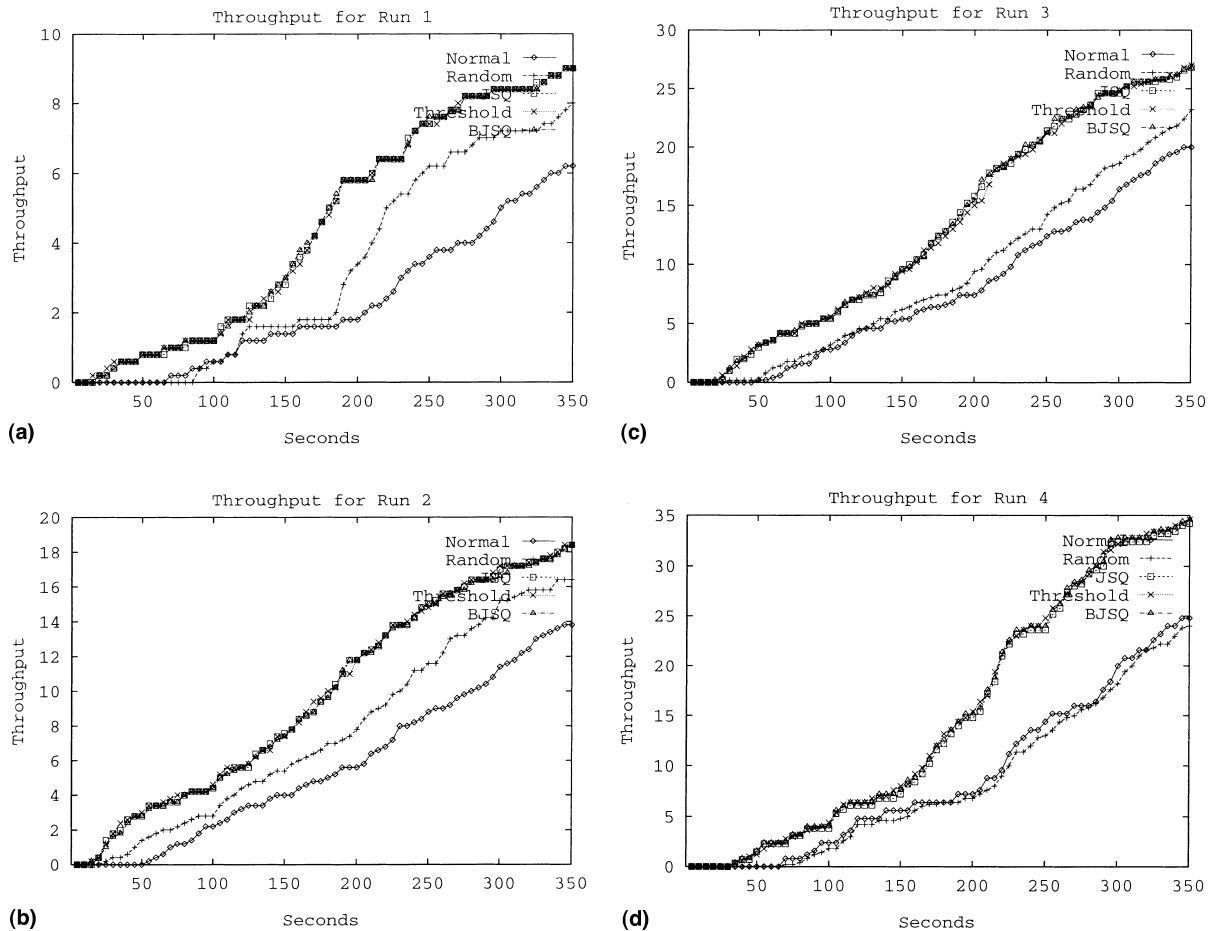


Fig. 5. (a) Throughput for Run 1; (b) throughput for Run 2; (c) throughput for Run 3; (d) throughput for Run 4.

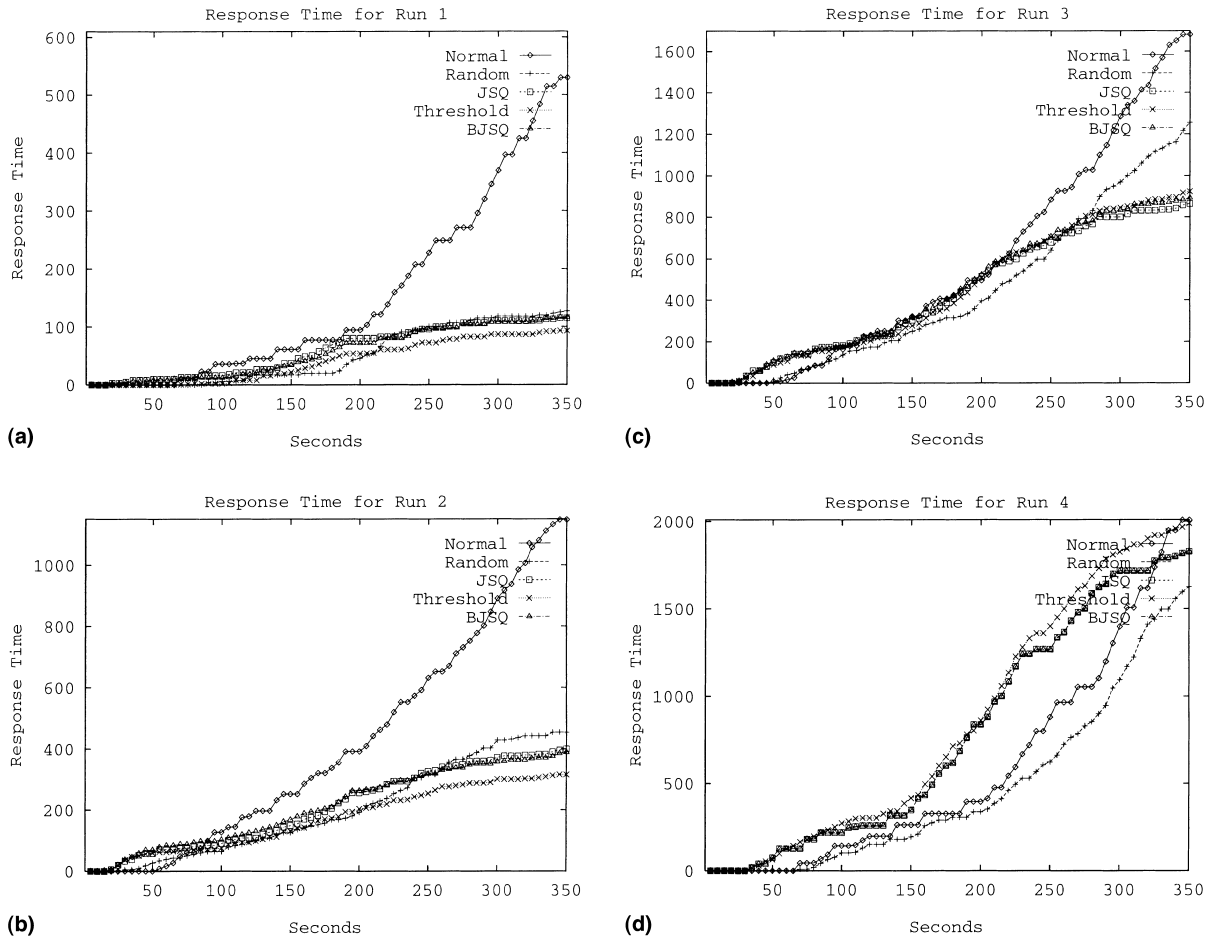


Fig. 6. (a) Response time for Run 1; (b) response time for Run 2; (c) response time for Run 3; (d) response time for Run 4.

remotely execute 200 out of the 220 jobs that arrived. The nodes were devoting most of their time to transferring tasks and little time to processing them. As mentioned previously, there is no benefit in executing a job on an overloaded node with a similar loading. This result confers with the observation made by Eager et al. (1986b) about the Random policy. They observed that the Random policy can degrade performance when many nodes are overloaded.

The average response times of the four algorithms and of the four cases are shown in Fig. 6. In situations where most of the nodes in the system are light-loaded, the Threshold algorithm achieved the best response time results. When most of the nodes become overloaded, JSQ and BJSQ performs better than the Threshold. The response time graph in Fig. 6(d) needs some explanation. The Threshold algorithm actually has a much worse response time than the normal case right up until the end of the simulation period at about 340 s. JSQ and BJSQ share a similar path slightly less than Threshold, but still worse than the normal curve. However, JSQ and BJSQ suddenly dip at the end of the graph to achieve a better response time for the period than normal and threshold. Random appeared to have the best

average response time. These apparent inconsistencies with the throughput results can be explained. Because the load was so high on average, most of the jobs which arrived did not actually finish by the time the simulation was ended, and therefore did not appear in the throughput and response time results. This can be seen in Table 4. If the simulation had been run for a longer period of time, then we would see that the response time would decrease relative to the Normal algorithm.

6. Conclusions

We have described a distributed simulation system designed for simulating load balancing algorithms. The simulation directly executes the codes of the load balancing algorithms on the real networking environment and uses real workload distributions. We have also reported our simulation results for four simple, dynamic load balancing algorithms which were considered suitable for a workstation-based locally distributed system.

It can be observed that, in most of the cases, all the algorithms can make substantial improvement in system performance. Even a simple load balancing algorithm

such as Random works well except in the situation, where all sites are overloaded. However this was an extreme case, and does not usually occur in the workstation-based distributed system. Runs 1 to 3 exemplify the normal conditions in the system, as observed in system log files containing previous processing statistics.

Threshold, JSQ and BJSQ all achieved similar results. However, Threshold achieved its results with a smaller number of remote executions and using less system state information. In the simulation runs, the BJSQ algorithm proved to achieve similar throughput results to JSQ and Threshold. BJSQ uses more state information than Threshold and is more complex than JSQ, but its performance is not significantly better than that of Threshold and JSQ. This validated the suggestion made by Eager et al. (1986b) that state information beyond that used by Threshold, or a more complex usage of state information, is of little benefit.

It was found that the cost of transferring system state information was negligible for the small number of sites considered. More simulations are needed to examine this cost for larger number of sites, and with different geographical distances between the sites.

The number of different system parameters that could have been tested in the simulations was very large, and many more simulations could have been performed. It would be interesting to investigate further the effects of varying the system sizes, threshold, and balance factor levels, on the system performance.

Acknowledgement

This research is partially supported by the University Grant Council of Hong Kong under the CERG project 9040229.

References

- Casavant, T.L., Kuhl, J.G., 1988. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering* 14 (2), 141–153.
- Cheriton, D.R., 1988. The V distributed system. *Communications of the ACM* 31 (3), 314–333.
- Chou, T.C.K., Abraham, J.A., 1982. Load balancing in distributed system. *IEEE Transactions on Software Engineering SE-8*, 401–411.
- Chow, Y.-C., Kohler, W.H., 1979. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers c-28* (5), 354–361.
- Covington, R., Dwarkadas, S., Jump, J.R., Madala, S., Sinclair, J.B., 1991. Efficient simulation of parallel computing systems. *International Journal of Computer Simulation* 1 (1), 31–58.
- Dickens, P.M., Heidelberger, P., Nicol, D.M., 1996. Parallelized direct execution simulation of message-passing parallel programs. *IEEE Transactions on Parallel and Distributed Systems* 7 (10), 1090–1105.
- Eager, D., Lazowska, E., Zahorjan, J., 1986a. A comparison of receiver-initiated and sender-initiated dynamic load sharing. *Performance Evaluation* 6 (1), 53–68.
- Eager, D., Lazowska, E., Zahorjan, J., 1986b. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering SE-12* (5), 662–675.
- Hudak, P., Goldberg, B., 1984. Experiments in diffused combinator reduction. *Proceedings of the ACM Symposium on Lisp and Functional programming*.
- Johnson, I.D., Harget, A.J., 1989. On the performance of load balancing algorithms in distributed systems. In: Ritter, G.X. (Ed.), *Information Processing 89*. Elsevier, Amsterdam, pp. 175–180.
- Krueger, P., Livny, M., 1988. A comparison of preemptive and non-preemptive load distributing. In: *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pp. 123–130.
- Litzkow, M.J., Livny, M., Mutka, M.W., 1988. Condor – a hunter of idle workstations. In: *Proceedings of the Eighth International Conference on Distributed Computing Systems*, San Jose, California, 13–17 June, pp. 104–111.
- Livny, M., Melman, M., 1982. Load balancing in homogeneous broadcast distributed systems. *Proceedings of the ACM Computer Network Performance Symposium*.
- Lin, H.-C., Raghavendra, C.S., 1992. A dynamic load-balancing policy with a central job dispatcher. *IEEE Transactions on Software Engineering* 18 (2).
- Lo, V.M., 1984. Heuristic algorithms for task assignment in distributed systems. In: *Proceedings of the Fourth International Conference on Distributed Systems*, pp. 30–39.
- Mehra, A.P., Wah, B.W., 1992. Adaptive load-balancing strategies for distributed systems. In: *Proceedings of the Second International Conference on Systems Integration*, IEEE Computer Society, Morristown, NJ, pp. 666–675.
- Mirchandaney, R., Towsley, D., Stankovi, J.A., 1989. Adaptive load sharing in heterogeneous systems. In: *Proceedings of the IEEE Ninth International Conference on Distributed Computing Systems*, pp. 298–305.
- Shen, S., 1988. Cooperative distributed dynamic load balancing. *Acta Informatica* 25, 663–676.
- Shivaratri, N.G., Krueger, P., 1990. Two adaptive location policies for global scheduling algorithms. In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 502–509.
- Asawa, M., Misra, S.P., 1991. ROBINHOOD: resource sharing by time stealing between DOS PCs on a LAN. Technical Report 91/2, Department of Computer Science, James Cook University.
- Svensson, A., 1990. History, an intelligent load sharing filter. *IEEE Transactions on Computers* 2878 (7), 546–553.
- Theimer, M.M., Lantz, K.A., 1989. Finding idle machines in a workstation-based distributed system. *IEEE Transactions on Software Engineering* 15 (11), 1444–1457.
- Tripathi, S.K., Menasce, D.A., 1992. Scheduling issues in heterogeneous multiprocessor systems. In: Becker, M., et al. (Eds.), *Transputers'92*. IOS Press, pp. 1–16.
- Waldspurger, C.A., et al., 1992. Spawn: a distributed computational economy. *IEEE Transactions on Software Engineering* 18 (2), 103–117.
- Wang, Y., Morris, R., 1985. Load sharing in distributed systems. *IEEE Transactions on Computers C-34* (3), 204–217.
- Wang, J., Zhou, S., Zheng, X., Delisle, P., 1992. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. TR-CSRI-257, Computer Systems Research Institute, University of Toronto, 1992.
- Zhou, S., 1988. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering* 14 (9), 1327–1341.

Dr. Jiannong Cao received the B.S. degree (1982) from Nanjing University, China, and the M.S. degree (1986) and Ph.D. degree (1990) from Washington State University, USA, all in computer science. He has been on the faculty of computer science at James Cook University (Queensland, Australia), University of Adelaide (South Australia, Australia), and City University of Hong Kong. He joined the Hong Kong Polytechnic University in 1997, where he is currently an assistant professor in the Department of Computing. Dr. Cao's research interests include parallel and distributed systems, computer networks, Internet computing, fault tolerance, and programming methodology and environments. His main areas of expertise are the design and analysis of algorithms and protocols for parallel/distributed system functions and services, and the development of software tools for programming distributed systems. He has published more than 70 technical papers in the above areas, which appeared in international journals and conference proceedings. His recent research has focused on how to build high-performance, fault tolerant distributed systems and applications

on the Internet. Dr. Cao is a member of ACM and IEEE Computer Society. He has served as a reviewer for international journals and also as a programme committee member for several international conferences.

Mr. Graeme Bennett graduated in 1993 with B.Sc. (Honours) in Computer Science from James Cook University, Australia. He then joined the Center for Information Technology Research at the University of Queensland, where he has been working on various projects as system programmer.

Dr. Kang Zhang is currently a Senior Lecturer with Computing Department of Macquarie University. He received his B.Eng. from the University of Electronic Science and Technology of China, in 1981; and Ph.D. from the University of Brighton in 1990. His research interests include program visualization, parallel programming tools, visual programming, and parallel implementation of logic programs.