Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms *

John A. Stankovic
Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, Mass. 01003, USA; tel. (413) 545-0720

Simulation results of three adaptive, decentralized controlled job scheduling algorithms which assume absolutely no a priori knowledge about jobs are presented. The results provide insight into the workings and relative effectiveness of the three algorithms, as well as insight into the performance of a special type of decentralized control. The simulation approach includes tuning the parameters of each algorithm, and then comparing the three algorithms based on response time, load balancing and the percentage of job movement. Each of the algorithms is compared under light, moderate, and heavy loads in the system, as well as a function of the traffic in the communication subnet and the scheduling interval. Modifications to the models are then introduced to further investigate the impact of various other parameters such as the cost of the scheduling algorithm itself, the effect of highly degraded state information and eliminating the movement of large (in size) jobs. Three simple analytical models are also presented and compared to the simulation results. A general observation is that, if tuned correctly, the decentralized algorithms exhibit stable behavior and considerably improve performance (response time and load balancing) at modest cost (percentage of job movement and algorithm execution cost).

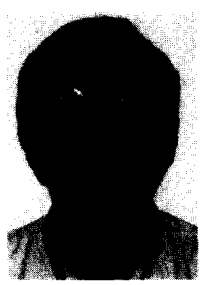
Keywords: Adaptive algorithms, decentralized control algorithms, decentralized control, distributed processing, job scheduling, networking and simulation study.

* This work was supported, in part, by the National Science Foundation under grant MCS-8104203, and by the US Army CECOM, CENCOMS under grant number DAAB07-82-K-J015.

North-Holland
Computer Networks 8 (1984) 199–217

1. Introduction

A distributed processing system is defined as a collection of processor-memory pairs (hosts) that are physically and logically interconnected, with *decentralized system-wide* control of all resources, for the cooperative execution of application programs [10,14]. Such systems may be dedicated to a single application or may implement a general purpose computing facility. By decentralized system-wide control is meant that there exists distributed resources in the system, that there is decentralized control of these resources (i.e., there is no single, central host "in charge", nor is there a central state table), and that there is system-wide cooperation between independent hosts which results in a *single* unified system. By system-wide cooperation is meant that the algorithms of the system operate for the "good of the whole" and not for a particular host. For systems meeting this restrictive definition of distributed processing, it is hypothesized that their reliability, extensibility, and performance will be better than what is generally



John A. Stankovic is an Assistant Professor in the department of Electrical and Computer Engineering at the University of Massachusetts, Amherst.

Professor Stankovic has been active in distributed systems research since 1976. He was co-editor of the January 1978 special issue of *Computer* on Distributed Processing. He now serves as the Vice-chairman of the IEEE Technical Committee on Distributed Operating Systems. In this capacity he has been responsible for serving as the editor of two special issues of the Technical Committee's Newsletter. His current research includes various approaches to process scheduling on loosely coupled networks an recovery protocols for distributed databases. He has been involved with building a distributed systems testbed called CARAT.

Professor Stankovic received the 1983 Outstanding Junior Faculty Award for the School of Engineering at the University of Massachusetts. He received his ScB in Electrical Engineering in 1970, and his ScM and Ph. D. in Computer Science in 1976, and 1979, respectively, all from Brown University, and is a member of ACM, IEEE and Sigma Xi.

available today. In this paper the term distributed processing refers to this very specific type of highly integrated distributed system.

For distributed processing systems to achieve their potential, questions relating to the effectiveness and stability of decentralized control algorithms must be answered. Solutions to the decentralized control problem in distributed databases are known. In this type of decentralized control problem, decentralized controllers must cooperate to achieve a system-wide objective of good performance subject to the data integrity constraint. The cooperation is achieved in various ways, e.g., either by the combined principles of atomic actions and unique timestamps or by the combined principles of atomic actions and two-phase locking. It can then be proven that multiple decentralized controllers can operate concurrently and still meet the data integrity constraint.

Most other research on decentralized control is better described as research on decomposition techniques rather than decentralized control. In such work, large problems are partitioned into smaller problems, each smaller problem being solved, for example, by mathematical programming techniques, and the separate solutions being combined via a few interaction variables. See [16] for an excellent summary of this type of decentralized control (decomposition).

Our interests lie in a type of decentralized control we refer to as stochastic replicated decentralized control. By replicated we mean that the decentralized entities (controllers) implementing the function are involved in the entire problem, not just a subset of it. This type of decentralized control is appropriate for certain functions like job scheduling where data integrity is not crucial. It is within the context of job scheduling that we investigate decentralized control.

Mathematical treatment [2,13,20,24,30,31] of decentralized control has not provided answers for stochastic replicated functions of distributing processing systems. This is due to the mathematically intractable nature of the problem when all simultaneously complicating factors are taken into account [22,24]. These factors include multiple, concurrent, decentralized controllers, each with equal authority, each working with noisy and out of date information; further there are unpredictable future events that can occur including significant delays, and decisions must be made quickly. Lacking analytical results, simulation can provide insight into the operation and effectiveness of this special type of decentralized control algorithm (a stochastic replicated function) as part of distributed processing systems.

In this paper simulation results of three adaptive, decentralized controlled job scheduling algorithms are presented. We try to answer the questions of whether multiple, decentralized, replicated, concurrently executing controllers can act together in such a manner so as to produce greater benefits than costs, whether concurrent actions taken by the multiple controllers produce stable behavior, and what parameters and variables are important to performance?

The simulation approach includes tuning each algorithm, and then comparing the three based on response time, load balancing, and the percentage of job movement. By tuning is meant that values for certain parameters of the algorithm are chosen to improve the performance of the algorithm. Each of the algorithms is compared under light, moderate, and heavy loads in the system, as well as a function of the traffic in the communication subnet and the scheduling interval. Modifications to the models are then introduced to further investigate the impact of other parameters such as the cost of running the scheduling algorithm itself, the effect of highly degraded state information and the effect of eliminating the movement of large jobs.

Three simple analytical models are also presented and compared to the simulation results. A general observation is that, if tuned correctly, the algorithms exhibit stable behavior and considerably improve performance (response time and load balancing) at modest cost (percentage of job movement). Equally important is that the algorithms are very simple and incur negligible run time costs. Conversely, if the algorithms are not tuned correctly they do not exhibit stable behavior and do incur high cost.

Section 2 describes exactly what is meant by a stochastic, replicated, decentralized controlled job scheduling algorithm, itemizes some special concerns of a practical algorithm for a distributed processing system, and relates this scheduling research to other current work. In the remainder of this paper we drop the terms stochastic and replicated for convenience but these characteristics are always implied. Section 3 presents the basic simu-

lation model used in this study. Section 4 describes the three algorithms to be compared as well as the motivations behind the choice of these algorithms. Section 5 presents and discusses the results of the simulations, including comparisons to analytical results. Section 6 summarizes the results of the paper.

2. Decentralized Controlled Job Scheduling

A decentralized controlled job scheduling algorithm is not the typical scheduling algorithm one finds in the literature. Its operation and environment are somewhat unique. This section describes those aspects of a decentralized controlled job scheduling algorithm and its environment that make it unique, along with its relationship to other scheduling research.

A decentralized controlled job scheduling algorithm is *one* algorithm composed of n physically distributed entities, $\{e_1, e_2, \ldots, e_n\}$. Each of the entities is considered a local controller. Each of these local controllers runs asynchronously and concurrently with the others, continually making decisions over time. Each entity e_i makes decisions based on a *system-wide* objective function, rather than on a local one. Each e_i makes decisions on an equal basis with the other entities (there is no master entity, even for a short period of time!). It is intended that the job scheduling algorithm adapts to the changing busyness of the various hosts in the system. In this paper we further constrain job scheduling to operate with absolutely no a priori knowledge about the job. Its function is to assign jobs to hosts to provide a gross level of load balancing which in turn improves response time. Once a job is activated it is considered a process and is further scheduled by a process scheduler. A process scheduler can sometimes dynamically acquire information about a running process and therefore can be more sophisticated in its scheduling. Process scheduling is not treated in this paper.

The motivations for splitting the scheduling function into two parts (job and process scheduling) are:

1. The job scheduler can be simple and therefore it is possible to study the use of decentralized control algorithms in a simpler situation (in this work we are attempting to determine how multiple, interacting, decentralized components of a single function interact under simple algorithms), and,
2. To eventually study how two (or more) decentralized control algorithms interact with each other, e.g., interaction between the job scheduler and the process scheduler may provide insight into how an entire system composed of multiple decentralized control algorithms of various types might function.

The environment in which the job scheduling entities are running is stochastic in two ways. First, the observations entities make about the state of the system are uncertain (they are estimates). Second, once a decision is made (e.g., move a job to host i), future random forces (e.g., a burst of jobs arrive at host i) that are independent of the control decision can occur.

In general, the observations themselves can be made in any number of ways. In this paper it is assumed that each entity periodically transmits its view of the busyness of the system to its neighbors, and upon receiving updates from its neighbors combines those updates in some manner so as to obtain its new view of the system. The update scheme used in our simulation is described in the next section.

Since job scheduling is an operating system function, any algorithm implementing job scheduling must run quickly. This is an extremely important aspect of the algorithms and makes many potential solutions unsuitable. For example, modeling the system by a mathematical program and solving it on-line is out of the question.

The execution costs involved in running a decentralized controlled job scheduling algorithm include the cost of running the algorithm itself, the cost of transmitting update information, and the cost of moving jobs. The primary goal of the algorithm is to minimize response time with minimum job movement. The secondary goal is to balance the load. All of these costs and goals are addressed with our simulations.

It is important to note the differences between job scheduling as studied in this paper and task allocation research. The task allocation problem for distributed systems normally assumes that *all* the tasks to be allocated are known beforehand. The cost of running each task on each processor is also assumed known. Further, all intertask communication patterns, as well as the cost of com-

municating over the network are assumed known. Given these assumptions, there are graph-theoretic [3,28,29], integer programming [6,7,9] queueing theoretic [1,5,15] and heuristic methods [4,8,18] for dealing with the task allocation problem. For a more detailed survey of some of these techniques see [12].

In other job scheduling research based on the bidding scheme [11,21] specific tasks are matched to processors based on the current ability of the processors to perform this work. These schemes are suboptimal, but are more extensible and adaptable than many of the other approaches. However, the cost of making and acquiring bids may become excessive, and the factors used in making the bids have not been extensively studied. We are also performing process scheduling work based on bidding [27]. In this work various types of a priori knowledge are assumed, for example, the files needed by a process, the cost of accessing these files locally and remotely, the effect of clustering or separating cooperating processes, etc. This type of work while more sophisticated than job scheduling also requires many more assumptions about the characteristics of processes.

Wave scheduling [32] and co-scheduling [19] are concepts that apply to clustering related jobs onto the same host and is not considered in this paper because it also deals with process scheduling and assumes prior knowledge.

Again, the job scheduling problem for a general purpose distributed processing system is different from the above referenced work. Jobs arrive at each geographically distributed host in some unpredictable manner. There is no central queue. At no point in time does the decentralized job scheduler know all the jobs to be assigned. Characteristics of the incoming jobs are also unknown. That is, the processing costs of the jobs, whether the job is actually multiple tasks that communicate with each other, the files required, and the I/O requirements are all unknown. The job scheduler's task is to provide a gross level of load balancing, hoping to improve response time at low cost. In other words, a practical job scheduling algorithm must be simple, run fast, minimize job movement, and improve system-wide response time and load balancing. Given that this high level load balancing of jobs is performed, a decentralized process scheduler can also be implemented that deals with processes (jobs in execution), interprocess communication, clustering and co-scheduling of processes.

Job scheduling work similar to the work described here, except that statistical decision theory is used as an integral part of the algorithms, can be found in [23,25,26]. In such work more sophisticated job scheduling algorithms are employed and the additional benefit of that sophistication is shown to be worthwhile when the quality of state information degrades significantly.

3. The simulation model

The simulation model, programmed in GPSS, consists of a network of five hosts connected as shown in Fig. 1. The unit of time in the simulation is milliseconds. Each host is considered identical except for processor speed. The service time of a job scheduled for execution is chosen from an exponential distribution with different averages for each host. The averages are 5000, 7000, 6000, 5000, and 7000 milliseconds for hosts 1–5 respectively. There are five independent sources for arrivals of jobs. Each source is modeled by a Poisson distribution with averages λ_1, λ_2, λ_3, λ_4, λ_5. The λ's vary depending on the particular loads (light, moderate, heavy) being modeled. Specific values of the λ's are presented with the simulation results in Section 5. When a job arrives at a host from the external world, it is assigned a size based on the distribution shown in Fig. 2. Delay in the communications subnet is modeled as a simple function, i.e., the size of the information to be sent divided by the packet size (1K bits) times the average delay per packet. Hence, the delay in the subnet is independent of the topology in our simulations. Both jobs and state information are passed into the subnet, thereby modeling two of the major costs involved. The third major cost, the cost of running the algorithm is initially modeled as a small fixed cost of 50 milliseconds each time it runs on each host. This is reasonable because of the simplicity of the algorithms involved. Later

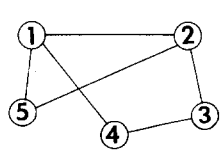


Fig. 1. Network Topology.

.1	1000	.85	22000
.2	12000	.90	30000
.4	14000	.95	34000
.6	16000	.98	38000
.7	18000	.99	44000
.8	20000	.995	50000

Fig. 2. Job Size Distribution.

this parameter is varied to test the effect on response time that a larger cost might have.

In the simulation, each host periodically calculates an estimate of the number of jobs at each host in the network, and sends this information to its nearest neighbors. This state information is updated at each host in the following way. Consider three classes of hosts, myself, my neighbors, and all others. In general, host *i* can determine precisely the number of jobs in its own queue (accurate local data), and therefore, will believe its own value rather than his neighbors perception of his workload. Since the nearest neighbors are only one hop away, their estimates of their workload, as passed to host i, will be only slightly out of date and, in general, will be a better estimate than estimates other nodes have of them. Therefore, host i uses the nearest neighbors estimate of themselves. Finally, all other views of any remaining hosts in the network are determined by taking an average of the estimates from all the incoming update messages. Between updates no attempt is made at estimating either the current state since the last update, nor any future state. Old information is simply used. We believe that the additional cost of such estimates is prohibitive in comparison to the potential benefits when the update interval is frequent (as in our case). Estimates between updates would be beneficial if (a) a reasonable estimation rule were known, and (b) the state information update interval were relatively infrequent.

The model also includes a simple FCFS process scheduler that does not allow multiprogramming. This should be adequate because we are evaluating job scheduling algorithms that operate without a priori information, and not process scheduling. A statistics gathering capability sufficient to determine the average response time per job, average queue length per host, and the number of jobs moved per host is included in the model. The model was designed so that it is also easy to plug-in different job scheduling algorithms. Three such algorithms, and the aspects of the simulation model relating to these algorithms, are described in the next section. All simulations are run for 10 minutes of simulated time, statistics are cleared, and then run for 30 more minutes. This is done to minimize the transient start up effects of an empty system. Longer runs did not substantially alter the results so to save computer time a total of 40 minutes per run was chosen as the test standard.

4.2. The Job Scheduling Algorithms

This section describes the three job scheduling algorithms that are implemented and compared via simulation. Before the algorithms themselves are described, those aspects of the simulation that are common to the three algorithms are described.

In each of the algorithms, the job scheduling entities are activated periodically (initially every 2 seconds and then varied for several later tests) and also whenever the process scheduler of a host needs to activate a new job for execution. Furthermore, state information (in this case, the number of jobs) about the busyness of the network is passed between neighbors on a periodic basis (initially every 2 seconds and then varies with the scheduling interval in later tests). Each scheduling entity then has an out of date observation of how many jobs exist at each site in the network. In future simulations we intend to study the effect of using more sophisticated state information (not just the number of jobs) to estimate the busyness of a host, and to use asynchronous updates of state information rather than periodic.

A practical consideration for job scheduling algorithms comes into play for very lightly loaded and heavily loaded systems. In both instances it is not beneficial to move jobs; therefore, in algorithms 1 and 3, described below, jobs are not moved by a host if it *observes* a very lightly or very heavily loaded system. Very lightly loaded is defined as "each host has less than 4 jobs." A very heavily loaded system is defined as "each host has more than 20 jobs." All other situations are considered moderate loads. We did not include these cutoffs in algorithm 2 in order to test the effect of omitting such cutoffs.

Another situation that is common to these algorithms is related to the delay in the communication subnet. If the delay in moving jobs is large

compared to the periodic update rate of scheduling entities, then the scheduling entities may continue to send jobs to a host not realizing there are many jobs "on the way" (in the subnet) to that host. There are a number of ways to deal with this problem. Our approach in algorithms 1 and 2 is to choose a fairly slow periodic update rate (acceptable for job scheduling) and move at most 2 jobs to a host at one time, so that even if there is a substantial delay in the subnet it is not expected that many jobs will be "on the way." However, in algorithm 3 we keep track of which host has been sent jobs recently and use this information in an effort to mitigate the problem and to minimize job movement.

Algorithm 1. For moderate loads in the system, each entity compares its own busyness to its observation (estimate) of the busyness of the *least* busy host. Note that the host thought to be least busy is itself an estimate. The difference between the busyness of these two hosts is then compared to a bias. If the difference is less than the bias, then no job is moved, else, one job is moved to the least busy host. Jobs are not moved to oneself.

Algorithm 2. Each entity compares its own busyness to its observation (estimate) of the busyness of *every* other host. All differences less than or equal to bias1 imply no jobs are moved to those hosts. If the difference is greater than bias1 but less than bias2 then one job is moved there. If the difference is greater than or equal to bias2 then two jobs are moved there. In no case are more than $(y)(z)$ jobs moved at one time from a host, where y = fraction of jobs permitted to be moved, and z = number of jobs currently at this host. If there is more demand for jobs than an entity is permitted to move, it satisfies the demand in a pre-determined fixed order.

Algorithm 3. This algorithm performs in the same way as algorithm 1 except when an entity, e_i, sends a job to host k at time t, it records this fact. Then for time delta t, called a window, this entity will not send any more work to host k. If at any time during the window period, entity e_i calculates that host k is least busy then no job is moved during such an activation. Of course, during the window, jobs may be sent to other hosts if they are observed as least busy by greater than the bias (same bias as in algorithm 1).

One prime motivation behind these three algorithms is that they are all very simple and inexpensive to run, necessary conditions for job scheduling algorithms which have no a priori knowledge about jobs. In algorithm 1 the relative busyness between host i and the least busy host (plus a bias) is used to determine if a job should move. This is about the simplest algorithm we can devise, but one might suspect that it has stability problems. Since algorithm 1 only moves jobs to the least busy host we felt that a better algorithm might be to spread the work around, i.e., move some work to all the lightly loaded hosts from the heavily loaded ones. Hence algorithm 2 was devised. Finally, we were worried that jobs in transit to a lightly loaded host were not taken into account possibly producing instabilities. For example, if (1) host 1 was very busy, (2) host 5 was least busy, (3) host 1 were activated every 2 seconds, and (4) it took 16 seconds for jobs to reach host 5, then host 1 could conceivably send at least 8 jobs to host 5 before the first one was received. Other hosts could be doing the same thing. This could result in an unstable situation. Algorithm 3 was designed to avoid such problems in as simple a way as possible. More sophisticated techniques to deal with stability are possible, but they require the retention of more past data.

5. Simulation and Analytical results

This section presents the simulation results, compares the simulation results to analytical results, and discusses the implications and insights provided by the results. Additional simulation tests are then proposed, run and discussed, in order to corroborate some of these implications.

In the first group of simulations four main characteristics were studied:

1. *Parameters of the Algorithms* – each algorithm has one or more parameters called biases in this paper (see section 4.2).
2. *Arrival Rates* – Four different sets of arrival rates are used in the simulations (Table 1). The sets of arrival rates are labeled tuning, light, moderate, and heavy. In tuning the three algorithms, it was decided to use a different set of

Table 1
Arrival Ratevs (Jobs/Second)

HOST	TUNING	LIGHT	MODERATE	HEAVY
1	.18	.1176	.153	.2
2	.1	.1	.125	.2
3	.111	.111	.143	.2
4	.133	.0588	.143	.2
5	.125	.125	.125	.2

arrival rates than those used for the subsequent algorithm comparisons. This is because in practice, such algorithms would not be tuned precisely for the current arrival rates, but would be only an approximation. The light and moderate loads are considered the normal network situation. It was also decided to simulate a heavy system load to determine how the algorithms perform when, for relatively short times (40 minutes), a system experiences arrival rates greater than its ability to service them.

3. *Delay in the subnet* – The change in the amount of traffic in the subnet affects response times and load balancing. The affect on these algorithms is studied here.
4. *Scheduling Interval* – The affect of a change in the scheduling interval from 2 → 4 → 8 → 16 seconds is tested.

While many other characteristics could be studied including size of the network, topology, and speeds of the hosts we felt that the characteristics studied here are some of the most important.

5.1. Analytical Results – No Network

Before the network simulation model was run, it was decided to obtain an upper bound on response time for the system. Therefore, the five hosts of our model were treated as independent M/M/1 queues for each set of arrival rates with no job scheduling algorithm and no network. If a system-wide job scheduling algorithm cannot do better than leaving all incoming jobs at their site of arrival, then as far as performance is concerned the algorithm is useless. The response times (wait time plus execution time) are 37.4 seconds for the tunning arrival rates, 29.22 seconds for the light load arrival rates, and 44.67 seconds for the moderate load arrival rates. One cannot apply the M/M/1 results to the heavy load arrival rates because the arrival rates are faster than the service rates implying infinite queues. Heavy arrival rates are used only to test the operation of the algorithm assuming very heavy loads would exist for a short period. Fortunately, as will be shown, all system-wide algorithms presented in this paper perform considerably better than the no network situation.

5.2. Simulation Results – Network With Imperfect Knowledge

Algorithms 1, 2 and 3 were tuned and then compared for light, moderate, and heavy system loads. Also tested was the effect of various delays in the subnet and scheduling intervals. The parameters of the tuning runs are identical to all other runs except for arrival rates and the specific parameters (described below) being tuned. In all the tables below, response times are given in seconds with 90% confidence intervals, and load balancing statistics are given in average number of jobs per host over three runs. Movement cost is given in percentage of job movement with 90% confidence intervals. By percentage of movement is meant the total number of jobs moved divided by the total number that entered the system. Note that jobs may move more that once and each move is counted separately because each move adds overhead to the system. The tuning arrival rates were chosen as an approximate mixture of light and moderate arrival rates but biased more towards the moderate arrival rates.

5.2.1. Tuning

Tuning algorithm 1 means choosing the proper bias between the number of jobs at the current host and the number at the least busy host. A bias equal to 2 was chosen for all subsequent runs because it provides a combination of good response time (an average of 14.7 sec) and load balancing with reasonably low job movement (an average of 27.6%). See the column under bias = 2 in Table 2. For a bias = 0 the variation in the percentage of job movement was somewhat unstable as shown by the 90% confidence interval (75 ± 28.5%). Percentage of movement greater than 100% is possible because jobs can move more than once. This implies that with the wrong bias algorithm 1 may not be very stable. Further evidence of this is given in section 5.6.

Tuning algorithm 2 means choosing bias1, bias2,

Table 2
Algorithm 1 Tuning

	BIASES			
	BIAS=0	BIAS=1	BIAS=2	BIAS=3
Response Time	14.13±.15	14.54±.42	14.7±.37	16.54±.7
% of Movement	75±28.5	47.3±10.7	27.6±1.45	24.3±2.5
Load Balancing (Jobs) (av over 3 runs)				
Host 1	1.06	1.37	1.67	1.80
Host 2	1.28	1.06	1.05	1.50
Host 3	0.86	1.05	0.96	1.34
Host 4	0.85	0.77	0.82	1.31
Host 5	0.82	1.30	1.12	1.29

and the fraction y of jobs permitted to be moved (refer back to section 4.2). Too many simulations would be required to tune all the possible combinations of these three biases. Educated guesses were made in the following way. It was argued that since there are only four hosts, excluding oneself, we would not move more than 3 at any one time (y bias). This allows movement to a large percentage of other hosts but in a controlled way. A number of preliminary simulation runs showed that only infrequently was the difference between the two host's busyness greater than 6 (about 4% of the time). We chose bias2 = 6 so that 2 jobs moved to a host at one time would occur infrequently, but would occur. At this point bias1 was

Table 3
Algorithm 2 Tuning

	BIAS1		
	BIAS1=2	BIAS1=3	BIAS1=4
Response Time	24.1±.45	16.3±.61	19.9±.55
% of Movement	110±33	25±4.3	23±3.6
Load Balancing (Jobs) (av over 3 runs)			
Host 1	1.50	1.21	1.63
Host 2	1.49	1.27	1.80
Host 3	1.29	1.15	1.93
Host 4	1.09	0.87	1.53
Host 5	1.05	0.98	1.61

Table 4
Algorithm 3 Tuning

	WINDOW SIZE			
	1 sec	2 sec	6 sec	8 sec
Response Time	22.21±.53	14.69±.47	19.83±.25	21.57±.61
% of Movement	31±1.9	16.6±1.7	13.1±1.15	22.33±1.4
Load Balancing (Jobs) (av over 3 runs)				
Host 1	6.01	1.95	4.88	3.59
Host 2	1.71	0.88	0.96	1.84
Host 3	1.22	1.19	0.89	1.30
Host 4	1.06	0.83	0.91	1.26
Host 5	0.85	0.85	1.02	2.24

tested and the results appear in Table 3. The result is that bias1 = 3 seems to work best and was used in subsequent testing.

Tuning algorithm 3 required picking the right bias and window. Since algorithm 3 is so similar to algorithm 1, and since bias = 2 worked so well for algorithm 1, it was decided to use the same bias for algorithm 3. A window size of 2 seconds produced the best results (Table 4). For the window size of 2 seconds there is good response time (an average of 14.69 sec), good load balancing, and a reasonably small percentage of jobs moved (an average of 16.6%). Notice that choosing a window size of 6 seconds degrades response time (an average of 19.83 sec), does not balance the load, and only a small percentage of jobs moved (an average of 13.1%). This indicates that the window size was too large to enable effective performance improvement. A window size of 8 seconds performed proportionally worse (Table 4).

5.2.2. Vary Arrival Rates

Tables 5, 6 and 7 show the results of the simulation runs by comparing algorithms 1, 2 and 3 for light, moderate, and heavy loads when the avervage delay in the subnet is approximately 8 seconds per job. In general, algorithms 1 and 3 perform similarly in terms of response time and load balancing, but algorithm 3 moves less jobs in achieving the result. For example, in the moderate load case (Table 6) algorithm 3's percentage of jobs moved is only an average of 40.75% as compared to algorithm 1's average of 57.5%, but its response time is slightly worse (an average of 24.02 sec as compared to an average of 22.81 sec).

On the other hand, algorithm 2 does not perform quite as well as algorithm 1 and 3 on light loads, significantly better on moderate loads, and worse on heavy loads. For light loads, having each host compare itself to all other hosts causes too much job movement and therefore produces slightly worse performance than algorithms 1 and 3. For moderate loads algorithm 2's spreading work to more than 1 lightly loaded host does pay dividends in the sense of improved response times. The poor performance of algorithm 2 under heavy loads is due to the lack of a cutoff. Notice (in Table 7) that algorithm 2 continues to perform load balancing regardless of the system load causing a tremendous amount of useless job movement (an average of 118%).

Table 5
Light Load Comparison

	ALGORITHM 1	2	3
Response Time	14.2±.41	15.2±2.2	14.9±.78
% of Movement	11±.5	19.9±7.5	14.8±.4
Load balancing (Jobs) (av over 3 runs)			
Host 1	0.85	0.77	0.77
Host 2	1.03	0.92	1.17
Host 3	0.83	0.77	0.84
Host 4	0.15	0.29	0.27
Host 5	0.94	1.23	1.35

Table 6
Moderate Load Comparison

	ALGORITHM 1	2	3
Response Time	22.81±.1	17.5±2.5	24.02±1.5
% of Movement	57.5±3.1	36±3.1	40.75±4.7
Load Balancing (Jobs) (av over 3 runs)			
Host 1	1.66	1.41	1.58
Host 2	1.66	1.53	2.07
Host 3	1.73	1.35	2.82
Host 4	1.43	1.01	1.83
Host 5	1.52	1.33	2.16

Table 7
Heavy Load Comparison

	ALGORITHM 1	2	3
Response Time	330.1±40.6	345.2±17	350±70
% Of Movement	28.5±2.2	118±5	7±5.1
Load Balancing (Jobs) (av over 3 runs)			
Host 1	54.0	50.7	44.4
Host 2	85.5	51.2	66.5
Host 3	69.0	50.9	54.5
Host 4	57.5	50.6	39.8
Host 5	104.0	50.9	55.1

5.2.3. Vary Subnet Delay

Next, consider the effect of different average delays (4, 8, 16 and 24 seconds) for jobs moving through the subnet. Tables 8, 9 and 10 present the response time and movement statistics for the three algorithms under light and moderate loads. Load balancing statistics are not shown because they do not provide any additional information over that given in Tables 5, 6 and 7.

The results presented in Tables 8, 9 and 10 show increased response time with increased delay in the subnet as is to be expected. However, there are two interacting factors that are involved. As the average delay for moving jobs through the subnet grows, there is greater cost in moving a job. Initially one might conclude that this adds to the average delay for a job in the system. Although this occurs sometimes, it is not necessarily true. A job moving through the subnet is doing so, presumably, to arrive at a shorter queue (in time)

Table 8
Algorithm 1 Varying Subnet Delay

	SUBNET DELAYS (sec)			
	4	8	16	24
LIGHT LOAD				
Response Time	12.7±.3	14.2±.41	15.04±.7	16.4±.63
% of Movement	10±2	11±.5	15.3±.9	8.2±1.7
MODERATE LOAD				
Response Time	17.1±1.3	22.8±.1	27.7±1	42.04±4.6
% of Movement	35±2.6	57.5±3.1	62.6±4.1	91±11.0

Table 9
Algorithm 2 Varying Subnet Delay

	SUBNET DELAYS (sec)			
	4	8	16	24
LIGHT LOAD				
Response Time	14.1±.8	15.2±2.2	17.0±1.9	18.8±.7
% of Movement	14.4±1.1	19.9±7.5	21.7±4.3	22.2±2.1
MODERATE LOAD				
Response Time	18.6±1.7	17.5±2.5	23.7±1.6	25.7±2.1
% Of Movement	31.5±4.8	36±3.1	45±4	43±6.7

Table 10
Algorithm 3 Varying Subnet Delay

	SUBNET DELAYS (sec)			
	4	8	16	24
LIGHT LOAD				
Response Time	14.0±.3	14.9±.78	14.8±.3	17.5±.61
% of Movement	9.6±.4	14.8±.4	15±1.1	15.01±1.6
MODERATE LOAD				
Response Time	22.4±.7	24.02±1.5	33.4±3.2	42.8±3.9
% of Movement	27±1.7	40.75±4.7	57.9±2.0	70±3.0

than the one it just left. Hence, only if the delay in the final queue (where it actually receives service) plus the time needed to move is greater than the delay it would have experienced without moving or by moving faster, is there a net degradation in system response time. It seems likely that there is a tradeoff point that is some function of the delay in the subnet and the waiting times in the hosts. This is the first factor affecting the results presented in Tables 8, 9 and 10.

The second factor involves a host's view of the network. When host i sends a job to host k, it's queue length is decreased by one. Host k queue length is not increased until some later time. Jobs in the subnet are no longer in any queue, therefore, temporarily out of the system in the host's eyes. In some situations (when there are alot of jobs in the subnet), there are fewer candidates for job movement and, therefore on the average less jobs move. Having less candidates for movement may give rise to increased response times.

The results seem to indicate that as delay in the subnet grows, job movement increases and response time decreases (gets worse). The magnitude of the changes being smaller than expected are believed to be caused by a combination of the above described factors.

5.2.4. *Vary Scheduling Interval*

Since algorithm 2 performed the best we decided to first study the effect of varying the scheduling interval on algorithm 2. The results are shown in Table 11. Notice that there is little difference in response time for scheduling intervals of 2, 4, and 8 seconds. Hence, there is no need to run the algorithm faster than every 8 seconds. When the scheduling interval is 16 seconds one starts to see a degradation in response time because the reaction time is not fast enough and there is not enough job movement to produce a good response time. Several similar runs were also made with algorithms 1 and 3 and they produced similar results. Rather than making alot more runs to generate confidence intervals for algorithms 1 and 3 we decided to combine varying the scheduling interval with a large subnet delay for algorithm 3.

Table 11
Algorithm 2 Scheduling Interval

	(delay in subnet - 8 sec)			
	Scheduling Interval (sec)			
	2	4	8	16
MODERATE LOAD				
Response Time	17.5±2.5	17.6±2.2	18.9±3.1	20.8±2.7
% of Movement	36±3.1	31.6±7.2	23.4±2.5	19±1.8

Table 12
Algorithm 3 Scheduling Interval

	(delay in subnet - 16 sec)		
	Scheduling Interval (sec)		
	2	4	8
MODERATE LOAD			
Response Time	33.4±3.2	27.5±3.4	22.0±3.9
% of Movement	57.9±2	36.9±5.1	16.9±2.8

Table 12 shows the result of varying the scheduling interval when the delay in the subnet is large (16 sec). These results are interesting in that the smallest scheduling interval results in poorest performance. For example, when the scheduling interval is 2 seconds, the response time is on the average 33.4 seconds and there is an average of 57.9% job movement. In this case too many jobs are moved into the subnet where the delay is now very substantial. Therefore, when the cost of moving a job gets too great, it pays to move fewer jobs as is the case when the scheduling interval is 8 seconds. This is seen in Table 12 where only an average of 16.8% of the jobs move resulting in better response time because the delay in the subnet is dominating the response time of the system. To mitigate this problem a possible variation to algorithm 3 would be to adapt the window size (currently 2 seconds) based on the current estimated delay in the subnet.

5.3. *Analytical Results – Fractional Assignment*

Assume for a moment that each host knows the fraction of jobs arriving at its site from the external world that it should keep and the fraction that it should send to each of the other hosts to minimize response time. In other words assume that the statistical properties of the network were known. Such an algorithm would have lower cost than the three algorithms just evaluated because

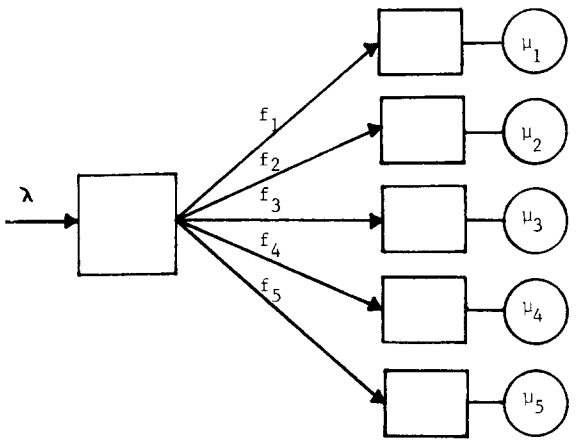


Fig. 3. Fractional Assignment Queueing Model.

no state information is passed around the network. But how would this "fractional assignment" algorithm compare to the above algorithms as far as response time is concerned. By making some simplifying assumptions, a queuing model can be formed and solved for this fractional assignment algorithm.

Assume a central queue with overall arrival rate $\lambda = \Sigma\lambda_i$, and 5 hosts with service rates μ_1, μ_2, μ_3, μ_4, μ_5. When jobs enter the central queue they are immediately transferred to one of the hosts based on the optimal fractional assignment, f_i, to each host (Figure 3) to minimize response time.

The system response time, T is given by

$$T = \sum_{i=1}^{5} f_i/(\mu_i - \lambda f_i),$$

and the fraction f_i can be calculated from

Table 13
Fractional Assignment Response Times

	System Load	
OPTIMUM FRACTION	LIGHT	MODERATE
.24 .16 .20 .24 .16	14.59	30.55

$$f_i = \frac{\mu_i}{\lambda} - \frac{\left(\sqrt{\mu_i}\right)\left(\sum_{i=1}^{5} \mu_i - \lambda\right)}{\lambda \sum_{i=1}^{5} \sqrt{\mu_i}}.$$

For a derivation of the formula for f_i see the Appendix. The results of these calculations for the values used in the simulations are given above in Table 13.

Note that the response times calculated here do not contain any delay for actual movement of jobs so this is a very optimistic figure. Further, it was assumed that the arrival rates were known so that the f_i's are the best possible fractions. Nevertheless, at a moderate load the three algorithms described above which make use of state information, perform considerably better than the fractional assignment. At light loads, they seem to be about equal.

5.4. Analytical Results – Lower Bound

The previous results indicate that very simple decentralized system-wide job scheduling algo-

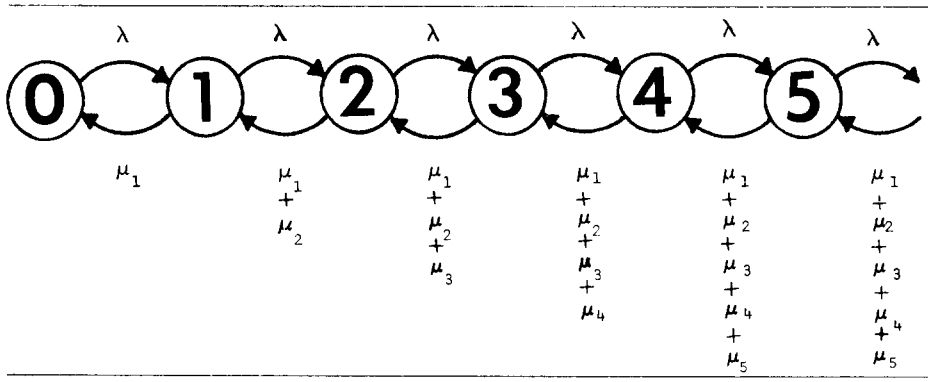


Fig. 4. State Transition Diagram.

rithms can improve response time considerably. However, in theory, how much better can we expect to do, i.e., what is the lower bound. Analytical solutions to the complex network situation we simulated are not known. However, by making simplifying assumptions it is possible to analytically obtain some idea of the lower bound.

Assume that the arrival process is Poisson and that service times are described by an exponential distribution. Assume that the service rate μ_i is ordered so that $\mu_i \geq \mu_j$ iff $i \leq j$. For our network of 5 hosts, this situation is described by the state transition diagram given in Figure 4.

In general, let the number of processors be N. Define

$$\mu(i) = \sum_{j=1}^{i} \mu_j,$$

$$M(i) = \prod_{j=1}^{i} \mu(i), \quad i = 1,2,\ldots,$$

$$M(0) = 1.$$

Then, using the conservation of flow principle and $\Sigma_j P(j) = 1$, we obtain

$$P_0 = 1 \Big/ \left(\sum_{j=0}^{N-1} \frac{\lambda^j}{M(j)} + \frac{\lambda^N}{M(N)} \Big/ \left(1 - \frac{\lambda}{\mu(N)}\right)\right).$$

The expected queue length is given by

$$\bar{L} = P_0 \left(\sum_{j=1}^{N-1} j \frac{\lambda^j}{M(j)} + \frac{\lambda^N}{M(N)} \left[\frac{N-1}{1-\lambda/\mu(N)} + \left(1 - \frac{\lambda}{\mu(N)}\right)^{-2} \right] \right).$$

Using Little's formula, the expected delay is

$$T = \bar{L}/\lambda.$$

The response times, T, calculated as lower bounds, are given in Table 14.

This lower bound calculation assumes no communication subnet delay, and that a job can be preempted and moved to a faster processor as soon as that processor becomes available, again at no cost. Although this analytical model cannot be achieved in practice, it does serve to put a lower bound on our results.

Table 14
Lower Bounds Response Times

System Load	
Light	Moderate
6.16 sec	9.3 sec

5.5. Comparison and discussion

In this section we first specifically compare the three algorithms tested and then discuss the insights provided by these simulations. Certain observations suggest further testing, which was done, and those results are presented in the next section. For easy comparison Tables 15 and 16 summarize the important statistics presented in the previous tables.

For light loads (Table 15), algorithms 1 and 3 are about equal, improving the no network response time by about 50% at a cost of moving an average of between 11 and 14.8 percent of their jobs, respectively. Both of these algorithms also perform slightly better than the fractional assignment algorithm when you consider that the response time calculated for it did not include delay due to job movement.

At moderate loads (Table 16), algorithms 1 and 3 improve the no network response times by approximately 49% and the fractional assignment response times by about 25%. Furthermore, for moderate loads algorithm 3 is superior to algorithm 1 because it achieves approximately the same performance (response time and load balancing) with less cost (job movement). Although such costs are already figured into response times we consider algorithm 3 better than algorithm 1 because we wish to avoid unnecessary job movement.

Note that for both light and moderate loads algorithms 1 and 3 perform better than the fractional assignment algorithm because they are able to adapt to the information about the state of the network in an effective way rather than relying on statistical properties. In conclusion, algorithms 1 and 3 seem to be stable if tuned correctly, and significantly improve performance, i.e., multiple decentralized controllers of a stochastic replicated function are cooperating effectively at very low cost.

Overall, algorithm 2 performs even better than algorithms 1 and 3 but has a tendency to move too many jobs if not tuned well (see the tuning runs).

Table 15
Summary Statistics – Light Load

	(Averages only - 90% CI shown in previous tables)					
	No Network (Upper Bnd)	Network Simulation A1	A2	A3	Fractional Assign.	Lower Bound
Response Time	29.22	14.2	15.2	14.9	14.6	6.16
Movement	0	11	19.9	14.8	-	-
Load Balancing (Jobs)						
Host 1	-	0.85	0.77	0.77	-	-
Host 2	-	1.03	0.92	1.17	-	-
Host 3	-	0.83	0.77	0.84	-	-
Host 4	-	0.15	0.29	0.27	-	-
Host 5	-	0.94	1.23	1.35	-	-

The tuning is also more complicated because of the three biases, but as for algorithms 1 and 3 the chosen biases seem to be fairly robust over the various arrival rates and system conditions. Further evidence of this is given in section 5.6. This implies that the algorithm can retain its simplicity and operate over a wide range of arrival rates. A modification would be to dynamically adapt the parameters (biases) of the algorithm paying the associated cost. Such a modification could be the basis for a future study. For moderate loads the response time of algorithm 2 was 61% better than the no network case, 43% better than the fractional assignment case, and approximately 24% better than algorithms 1 and 3. We, therefore, recommend algorithm 2 but cutoffs, such as those found in algorithms 1 and 3 should be added to algorithm 2.

While startling results were not obtained, nor expected, the simulations did provide a number of insights. We now summarize these ideas.

It seems that even very simple decentralized

Table 16
Summary Statistics – Moderate Load

	(Averages only - 90% CI shown in previous tables)					
	No Network (Upper Bnd)	Network Simulation A1	A2	A3	Fractional Assign.	Lower Bound
Response Time	44.67	22.81	17.5	24.02	30.55	9.3
Movement	0	57.5	36	40.7	-	-
Load Balancing (Jobs)						
Host 1	-	1.66	1.41	1.58	-	-
Host 2	-	1.66	1.53	2.07	-	-
Host 3	-	1.73	1.35	2.82	-	-
Host 4	-	1.43	1.01	1.83	-	-
Host 5	-	1.52	1.33	2.16	-	-

control algorithms for job scheduling have a number of internal parameters that can significantly affect stability and response time. In our simulations it was possible to choose values for these parameters that worked well over the reasonably different arrival rates, subnet delays and scheduling intervals tested. In general, such parameters should be adaptive to insure stability and performance over a more diverse set of conditions.

The goal of our job scheduling algorithms is to provide a gross level of load balancing to insure adequate work for each host. If hosts are all very busy we found it to be *imperative* that scheduling is turned off under such conditions. Another potential cutoff involves avoiding the movement of large jobs because of their large delays in the subnet. This is one piece of a priori information that is easy to obtain. This modification is investigated further in the next section.

In general, increased delays in the subnet increase system response time. However, we found that there is a flat portion to the response time curve where such an effect is felt only if the delay in the final queue (where a job actually receives service), plus the time needed to move is greater than the delay the job would experience without moving, or by moving faster. In other words, why hurry to the new host when you are just going to wait there too.

Increased delays also degrade the quality of the state information hosts have about each other. It would be interesting to better determine the effect of "out of date" information. Additional simulations were run to provide some insight into the effect of poorer and poorer information with the results being presented in the next section.

By comparing our simulation results to the optimal, fractional assignment algorithm, obtained analytically, we have specifically shown that significant performance improvement can be obtained by passing state information between hosts, even if that state information is simple.

When implementing a decentralized job scheduling algorithm an important question is how often to invoke the scheduler. Invoking the scheduler too often increases cost but a more important factor is the increase in non-beneficial job movement. While this result is interesting by itself, it has further implications. For example, as the cost of moving jobs increases, it becomes more important to move less jobs; one method is by slowing down the job scheduler. In some systems the overhead costs of the job scheduling algorithm could also play a major role, but in the above simulations this overhead was small (50 ms). This suggests further tests to determine the effect of increased overhead costs. Again, these are described in the next section.

Finally, since algorithm 2 performed best, it seems that trying to spread work among lightly loaded hosts is a good idea that does not result in too much job movement assuming proper tuning.

5.6. Additional Simulations

We now present the results for three additional simulation studies referred to in the previous section. The statistics presented are averaged over 2 runs. However, if a statistic is taken from the previous section (for purposes of comparison), then it is shown with its confidence interval. Only two runs (with different random number generator seeds) per test were made because of the expense of simulation, and, more importantly, because the confidence intervals on previous tests were small.

5.6.1. Increased Overhead

As stated before, the overhead of invoking the scheduling algorithm was modelled as 50 ms per invocation per host. Increased costs of this overhead INCREASED of this overhead might occur due to different costs of context switches on different machines and operating systems. Table 17 shows the effect on Algorithms 1 and 3 of increased overhead. For our runs, the extra time required for the scheduler, by itself, would add 0.37 sec or 0.74 sec to response time for the 150 ms and 250 ms runs, respectively. But the overall effect is much more significant in a negative way (e.g., for algorithm 1 the actual degradation in response time is from 17.1 sec for the 50 ms case to 26.3 sec for the 150 ms case). Hence, it seems quite important to keep the cost of the algorithm low. Further, algorithm 1, because of the high percentage of movement, should be re-tuned for the higher scheduling costs. This adds credence to our observation that tuning needs to be adaptive and that algorithm 1 is not quite as good as the others because it is more prone to instabilities.

5.6.2. Degraded State Information

In these tests, when state information was sent

Table 17
Scheduling Overheads

		Overheads		
ALGORITHM		50ms	150ms	250ms
1	Response T.	17.1±1.3	26.1	28.1
	% of Movement	35±2.6	87.4	95.2
3	Response T.	22.4±.7	27.0	27.3
	% of Movement	27±1.7	33.4	38.4

Scheduling Interval - 2 sec
Delay per packet - 250 ms

into the subnet, it was delayed an extra 2, 3, 5, 7, and 10 seconds. The scheduling interval was 2 seconds. Hence, in the worst case, the scheduler was using out of date information from an immediate neighbor that was more than 10 seconds old. Algorithm 2 and 3 each show equivalence classes of response times (Table 18). In algorithm 2 the degradation in response time occurs when the data gets between 5 and 7 seconds old. Delays of less than 5 seconds result in about the same response time of roughtly 19–21 sec, while delays between 7 and 10 seconds also have the same effect on response time but produce response times of approximately 26.5 sec. Algorithm 3 follows the same pattern but the break between equivalence classes occurs between 7 and 10 seconds. Such significant degradation in response time implies the need to take such poor information into account. Such a technique has been successful and is reported in [26]. Finally, notice that algorithm 2 again performs better than algorithm 3 under yet another set of conditions.

5.6.3. Job Size Cutoff

In this test a cutoff was added to algorithm 3 such that any job greater than or equal to 30000, 40000, and 45000 bytes, respectively, was not transmitted through the subnet. The result is shown in Table 19. Note that when the cutoff is 30000, too many jobs (see job size distribution – Figure 2) are restricted from moving and this severely degrades response time. A cutoff of 45000 significantly improves response time (from 22.4 sec with no cutoff to 15.7 sec with a 45000 byte cutoff). It is obvious that algorithm 3 should use a cutoff of 45000 bytes given the conditions tested here. We then applied the same cutoff of 45000 bytes to algorithm 2. Here we found no difference between

Table 18
Delayed State Information

		DELAYS				
ALGORITHM		2 sec	3 sec	5 sec	7 sec	10 sec
2	Response T.	19.6	21.3	19.1	26.7	26.4
	% of Movement	41.2	53.7	40.5	85.6	77.1
3	Response T.	27.0	-	26.9	27.2	44.1
	% of Movement	33.4	-	47.4	59.4	75.5

Cost of Scheduler - 150 ms
Scheduling Interval - 2 sec.

Table 19
Job Size Cutoff (Algorithm 3)

	Cutoff Size in Bytes				
	None	30000	35000	40000	45000
Response T.	22.4±.7	35.1	21.4	22.6	15.7
% of Movement	27±1.7	23.8	25.3	42.4	19.3

Delay per packet - 250 ms
Scheduling Interval - 2 sec

the no cutoff simulation and the cutoff of 45000 bytes (Table 20). This indicates that algorithm 2 is more robust than algorithm 3 and does not need the cutoff under the conditions tested.

6. Summary

This paper presents the simulation results of three stochastic, replicated, decentralized controlled job scheduling algorithms operating in a network with imperfect knowledge. The major assumptions of our work are that the job scheduler must operate with no a priori information and must execute quickly. While our simulation model accounts for a stochastic environment and includes all major costs, it is simplified in two ways: the model of the subnet is simple and there is no multiprogramming.

Overall, the results show that the multiple decentralized controllers of job scheduling algorithms can effectively cooperate with each other to achieve a system-wide objective of good response time and load balancing with limited job movement. When tuned properly, the algorithms operated in a stable manner. The tuning was shown to be necessary and should be adaptive, but once tuned the algorithms are fairly robust for different arrival rates, scheduling intervals and subnet dealys.

Table 20
Job Size Cutoff (Algorithm 2)

	Cutoff Job Sizes	
	None	45000
Response T.	18.6±1.7	18.6
% of Movement	31.5±4.8	38.2

Delay per packet - 250 ms
Scheduling Interval - 2 sec

By specifically comparing the performance of the three algorithms to each other and to analytical results, it was concluded that while algorithms 1 and 3 worked reasonably well, algorithm 2 was the best. Algorithm 1 is extremely simple so it is surprising that it works as well as it did, but it did seem to be the least stable of the three.

The simulations also provided a number of insights, as described in sections 5.5 and 5.6, into the performance and stability of decentralized control job scheduling algorithms as part of distributed processing systems. These include the fact that stability can be handled by tuning the parameters of the algorithm, that it is imperative to turn off scheduling under heavy loads, that avoiding the movement of large jobs can sometimes improve performance, that out of date information does significantly degrade performance and if long delays are expected a technique such as [26] is probably necessary, that increased delays in the subnet only degrade response time after a threshold is surpassed, that making use of even the simplest state information significantly improves performance, that invoking the scheduler too often increases the non-essential job movement and that response time is very sensitive to the cost of running the job scheduler.

Simulations of more sophisticated algorithms are underway and we will be able to compare their performance to the simpler algorithms presented in this paper. However, as this paper shows, it is

extremely important to keep the cost of the algorithm itself very low. In many cases the more sophisticated algorithm will not be worth the cost.

7. Acknowledgments

I would like to thank Don Towsley for his suggestion to compare the simulation results to the best "fractional assignment" and for his help with the analytical models.

Appendix

This appendix describes the derivation of the formula for f_i given in section 5.3. Starting with

$$T = \sum_{i=1}^{5} f_i/(\mu_i - \lambda f_i),$$

we need to minimize delay, T, subject to $\Sigma_{i=1}^{5} f_i = 1$. The Lagrangian function can be defined as

$$L = \sum_{i=1}^{5} \frac{f_i}{\lambda_i - \lambda f_i} - g\left(\sum_{i=1}^{5} f_i - 1\right),$$

where g is the Lagrangian multiplier. Taking the derivative and setting it equal to zero results in

$$\frac{\partial L}{\partial f_i} = \frac{\mu_i}{(\mu_i - \lambda f_i)^2} - g = 0,$$

$$f_i = \left(\mu_i - \frac{\sqrt{\mu_i}}{\sqrt{g}}\right)/\lambda \qquad (1)$$

Since $\Sigma_{i=1}^{5} f_i = 1$ we have

$$\sum_{i=1}^{5} \mu_i/\lambda - \sum_{i=1}^{5} \sqrt{\mu_i}/\lambda\sqrt{g} = 1,$$

$$\sqrt{g} = \sum_{i=1}^{5} \sqrt{\mu_i}\Big/\left(\sum_{i=1}^{5} \mu_i - \lambda\right).$$

Substituting $\sqrt{g}$ into formula (1) gives

$$f_i = \frac{\mu_i}{\lambda} - \frac{\left(\sqrt{\mu_i}\right)\left(\sum_{i=1}^{5} \mu_i - \lambda\right)}{\lambda \sum_{i=1}^{5} \sqrt{\mu_i}}.$$

References

[1] A.K. Agrawala, S.K. Tripathi, and G. Ricart, "Adaptive Routing Using a Virtual Waiting Time Technique," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 1, January 1982.

[2] Masanao Aoki, "Control of Lage-Scale Dynamic Systems by Aggregation," *IEEE Transactions on Automatic Control*, June 1978.

[3] S.H. Bokhari, "Dual Processor Scheduling with Dynamic Reassignment," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 4, July 1979.

[4] R.M. Bryant, and R.A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proceedings 2ND International Conference in Distributed Computing Systems*, April 1981.

[5] Yuan-Chieh Chow, and Walter Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," *IEEE Transactions on Computers*, Vol. C-28, No. 5, May 1979.

[6] W.W. Chu, "Optimal File Allocation in a Multiple Computing System," *IEEE Transactions on Computers*, Vol. C-18, pp. 885–889, October 1969.

[7] W.W. Chu, L.J. Holloway, M. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer*, Vol. 13, pp 57–69, November 1980.

[8] Kemal Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems, *IEEE Computer*, Vol. 15, No. 6, June 1982.

[9] O.I. El-Dessouki, and W.H. Huan, "Distributed Enumeration on Network Computers," *IEEE Transactions on Computers*, Vol. c-29, pp 818-825, September 1980.

[10] Philip Enslow, "What is a Distributed Data Processing System," *IEEE Computer*, Vol. 11, No. 1, January 1978.

[11] D.J. Farber, et al, "The Distributed Computer System," *Proceedings 7th Annual IEEE Computer Society International Conference*, February 1973.

[12] M.J. Gonzalez, "Deterministic Processor Scheduling," *ACM Computing Surveys*, Vol. 9, No. 3, pp 173–204, September 1977.

[13] R.A. Jarvis, "Optimization Strategies in Adaptive Control: A Selective Survey," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-5, No. 1, January 1975.

[14] Douglas Jensen, "The Honeywell Experimental Distributed Processor – An Overview of Its Objectives, Philosophy and Architectural Facilities," *IEEE Computer*, Vol. 11, No. 1, January 1978.

[15] L. KLeinrock, and A. Nilsson, "On Optimal Scheduling Algorithms for Time-Shared Systems," *JACM*, Vol. 28, No. 3, pp 477–486, July 1981.

[16] R.E. Larsen, "Tutorial: Distributed Control," IEEE Catalog No. EHO 153-7, IEEE Press, New York, 1979.

[17] G. LeLann, "Distributed Systems – Towards a Formal Approach," *Proceedings IFIP Congress*, Toronto, North Holland Publ., pp 155–160, August 1980.

[18] P.Y.R. Ma, E.Y.S. Lee, and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," *IEEE Transactions on Computers*, Vol. C-31, No. 1, pp 41–47, January 1982.

[19] John K. Ousterhout, D. Scelza, and P. Sindu, "Medusa: An Experiment in Distributed Operating System Structure," *CACM*, Vol. 23, No. 2, February 1980.

[20] Nils R. Sandell, Pravin Varaiya, Michael Athans, and Michael Safonov, "Survey of Decentralized Control Methods for Large Scale Systems, *IEEE Transactions on Automatic Control,* Vol. AC-23, No. 2, April 1978.

[21] R.G. Smith, "The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver," *IEEE Transactions on Computers,"* Vol. C-29, No. 12, Dec. 1980.

[22] John A. Stankovic, "A Comprehensive Framework for Evaluating Decentralized Control," *Proceedings 1980 International Conference on Parallel Processing*, August 1980.

[23] John A. Stankovic, "Analysis of a Decentralized Control Algorithm For Job Scheduling Utilizing Bayesian Decision Theory," *Proceedings 1981 International Conference on Parallel Processing,* August 1981.

[24] John A. Stankovic, N. Chowdhury, R. Mirchandaney, I. Sidhu, "An Evaluation of the Applicability of Different Mathematical Approaches to the Analysis of Decentralized Control Problems," *Proceedings COMPSAC82* November 1982.

[25] John A. Stankovic, "A Heuristic for Cooperation Among Decentralized Controllers," *Proceedings INFOCOM 83, April 1983.*

[26] John A. Stankovic, "Bayesian Decision Theory and Its Application to Decentralized Control Algorithms," to appear *IEEE Transactions on Computers.*

[27] J.A. Stankovic and I. Sidhu, "An Adaptive Bidding Algorithm For Processes, Clusters and Distributed Groups," *Proc. 4TH International Conference on Distributed Computing,* May 1984.

[28] H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, pp 85–93, January 1977.

[29] H.S. Stone, and S.H. Bokhari, "Control of Distributed Processes," *IEEE Computer*, pp 97–106, July 1978.

[30] Robert R. Tenney, and Nils R. Sandell, Jr., "Structures for Distributed Decisionmaking," *IEEE Transactions on Systems, Man, and Cybernetics,* Vol. SMC-11, No. 8, pp 517–527, August 1981.

[31] Robert R. Tenney, and Nils R. Sandell, Jr., "Strategies for Distributed Decisionmaking," *IEEE Transactions on Systems, Man, and Cybernetics,* Vol. SMC-11, No. 8, pp. 527–538, August 1981.

[32] L. Wittie, and Andre M. van Tilborg, "MICROS, A Distributed Operating System for Micronet, A Reconfigurable Network Computer," *IEEE Transactions on Computers,* Vol. C-29, No. 12, December 1980.