Load Sharing in Distributed Systems

YUNG-TERNG WANG, MEMBER, IEEE, AND ROBERT J. T. MORRIS, MEMBER, IEEE

Abstract — **An important part of a distributed system design is the choice of a load sharing or global scheduling strategy. A comprehensive literature survey on this topic is presented. We propose a taxonomy of load sharing algorithms that draws a basic dichotomy between source-initiative and server-initiative approaches. The taxonomy enables ten representative algorithms to be selected for performance evaluation. A performance metric called the Q-factor (quality of load sharing) is defined which summarizes both overall efficiency and fairness of an algorithm and allows algorithms to be ranked by performance. We then evaluate the algorithms using both mathematical and simulation techniques. The results of the study show that: i) the choice of load sharing algorithm is a critical design decision; ii) for the same level of scheduling information exchange, server-initiative has the potential of outperforming source-initiative algorithms (whether this potential is realized depends on factors such as communication overhead); iii) the Q-factor is a useful yardstick; iv) some algorithms, which have previously received little attention, e.g., multiserver cyclic service, may provide effective solutions.**

Index Terms — **Distributed scheduling, distributed systems, load sharing, performance analysis, queueing analysis.**

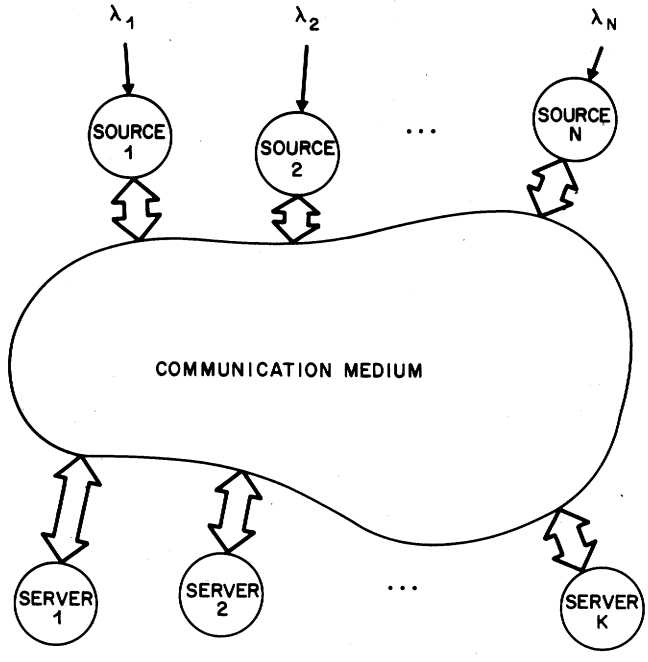


Fig. 1. A locally distributed computer system.

I. Introduction

THE confluence of low-cost processor (e.g., microprocessor) and interconnect (e.g., local area network) technologies has spurred a great interest in locally distributed computer architectures. The often cited advantages of these architectures include high performance, availability, and extensibility at low cost. But it is also being recognized that to realize this potential, crucial design decisions have to be made for which the performance implications are not intuitively obvious.

To set the stage, we consider a locally distributed computer system with configuration shown in Fig. 1. Jobs enter the system via nodes called *sources* and are processed by nodes called *servers*. This configuration represents a logical view of the system — a single physical processor might be both a source and a server node. This study is concerned with one important design issue: the choice of a *load sharing strategy* by which the multiple servers are made to cooperate and share the system workload.

The load sharing problem can be considered as part of the larger *distributed scheduling problem*.[1] Distributed scheduling can be thought of as comprising two coupled subproblems. The *local scheduling* subproblem addresses the allocation of processing resources to jobs within one node, either source or server. The local scheduling problem has been extensively studied; see, e.g., [13], [16], [20], [28]. The *global scheduling* subproblem addresses the determination of which jobs are processed by which server [1], [27]. A global scheduling strategy manifests itself as a collection of algorithms performed by the sources, servers, and possibly the communication medium. One of the key functions[2] of a global scheduling strategy is load sharing: the appropriate sharing and allocation of system processing resources. Load sharing has also been referred to as *load balancing*. This terminology arises from the intuition that to obtain good performance it would appear to be desirable to balance load between servers, i.e., to prevent the situation that one server is congested while another is lightly loaded.

Desirable properties of a load sharing strategy include:

i) *optimal overall system performance* — total processing capacity maximized while retaining acceptable delays;

ii) *fairness of service* — uniformly acceptable performance provided to jobs regardless of the source on which the job arrives;

Manuscript received November 28, 1983; revised March 15, 1984.

The authors are with AT&T Bell Laboratories, Holmdel, NJ 07733.

[1]We will focus on distributed scheduling as opposed to centralized scheduling.

[2]Other functions include data synchronization, concurrency control, etc.

iii) *failure tolerance*—robustness of performance maintained in the presence of partial failures in the system.

Other factors that need to be considered are performance of the system under contingent conditions such as overload, and the degree to which the strategy accommodates reconfiguration and extension.

An examination of the literature reveals a plethora of proposed approaches to the load sharing problem. A comprehensive literature survey based on a literature database search is presented in Appendix A. Despite this large body of literature, designers are left with only a collection of seemingly unrelated schemes to choose from, or else they must create their own. Unfortunately, a lack of uniformity of assumptions in different studies and idiosyncratic features of different schemes makes it unclear what performance gains are potentially achievable when more complex load sharing strategies are employed. The purpose of this paper is to begin the process of systematically describing and categorizing load sharing schemes and comparing their performance under a uniform set of assumptions.

II. Approach

To make a reasonably comprehensive study tractable within the space of this paper, we will make further limiting but simplifying assumptions about the system shown in Fig. 1. We assume that the jobs are individually executable, are logically independent of one another, and can be processed by any server. Such a situation might correspond to, for example, a query-intensive database system with replicated data at each server, or a system of "diskless processor" servers to which both program and data are downloaded from the sources. It may also correspond to the projection of a system onto a subsystem containing those servers that can process a class of jobs. We will only be considering load sharing strategies that distribute jobs to servers irrevocably, i.e., a job will not be assigned to one server and later reassigned to a different server. For simplicity, we assume that all servers have the same processing rate.

The load sharing strategy must cause the jobs to be shared among the servers so that concurrent processing is enabled and bottlenecks are avoided whenever possible. To demonstrate to the reader that there are many possible approaches to this problem, we might ask whether it might be better for the sources to "take the initiative" and forward jobs to a selected server or whether the sources should await the removal of the job by an available server. We could also ask what the minimal amount of information sharing is that must take place between nodes to obtain effective resource sharing.

While our model is somewhat idealized, it is general enough to allow numerous load sharing strategies to be discussed and compared. To expose some fundamental differences, in Section III we will present a taxonomy of load sharing strategies. By selecting representative algorithms of each class in the taxonomy, we will be able to conduct a performance comparison on an equal basis.

In comparing the performance of different schemes, there are a number of different and conflicting measures that could be proposed. But rather than dismiss a performance ranking of different schemes as being unattainable, we will propose in Section IV a single figure of merit for a load sharing strategy which captures some first order effects a designer should consider in selecting an algorithm.

It will be seen that this approach provides a unified framework within which we will be able to expose some fundamental distinctions between various strategies and to explore the inherent tradeoff between performance and communication overhead and complexity.

III. A Taxonomy of Load Sharing

In examining the many possible approaches to load sharing (for some examples, see the literature survey in Appendix A), we find that there are fundamentally distinguishing features of different strategies.

The first distinction we draw demarcates the type of node that takes the initiative in the global searching. If the source node makes a determination as to where to route a job we call this a *source-initiative* strategy. On the other hand, if the server "goes looking for work," i.e., determines which jobs at different sources it will process, we refer to this as *server-initiative*. In a source-initiative algorithm, queues tend to form at the server, whereas in a server-initiative algorithm queues tend to form at the sources. Another difference is that in source-initiative algorithms, scheduling decisions are usually made at job arrival epochs (or a subset thereof), whereas in server-initiative algorithms, scheduling decisions are usually made at job departure epochs (or a subset thereof).

The other axis of classification we propose refers to the level of *information dependency* that is embodied in a strategy. By this is meant the degree to which a source node knows the status of servers or a server knows the status of sources. Naturally, as the level of information available increases we expect to be able to obtain strategies with improved performance. But as more information is exchanged, communication cost may increase and more sophisticated software or hardware mechanisms may become necessary. For two strategies with the same level of information dependency, we will notice a subjective duality at play when considering a source-initiative versus server-initiative strategy. One of the outcomes of our performance analysis in later sections is that provided communication costs are not a dominating effect, server-initiative tends to outperform source-initiative strategies for the same level of information dependency.

In presenting our taxonomy we describe a source-initiative algorithm according to a function

$$\text{server} = f(\cdots)$$

by which we mean that the server selected by the source to process a job is a function of the arguments of f exhibited. Similarly, server-initiative algorithms are described by a function

$$\text{source} = f(\cdots)$$

the source at which a server seeks its next job. The taxonomy is presented in Table I. There are seven levels of information dependency which range from "blind" fixed scheduling where no information is required, to strategies where extensive information is required and algorithms such as global first come first served (FCFS)—one ideal of multiserver behavior—are attained.

Several explanations of Table I are in order. First, the number of levels of information is somewhat arbitrary and could have been aggregated into fewer or expanded into a finer classification. Note that the information levels are arranged so that the information of a higher level subsumes that of the lower levels. The parameter ω plays the role of randomization which can be viewed as providing additional information dependency.

Also shown in Table I are canonical examples of algorithms in each category. For example, a "cyclic splitting" algorithm assigns jobs to servers in a cyclic manner, remembering only where it sent a previous job (the sequence state). Its dual, a "cyclic service" algorithm, serves sources in a cyclic manner, remembering where it picked up the previous job. An algorithm such as FCFS requires much more information. Some of these example algorithms of Table I will be defined more carefully in Section V.

We will make extensive use of the canonical examples of Table I since they are "pure and simple" representatives of the basic strategy of each category. We maintain that many of the algorithms that have been proposed in the literature can be viewed as variations on or hybrids of members of the taxonomy. In view of some of the simplifying assumptions we have made in Section II, the taxonomy cannot be regarded as exhaustive. Nevertheless, as well as being a useful conceptual tool, in the process of setting up this table we have obtained some attractive simple algorithms that have not received attention in the past. For example, a version of a multiserver cyclic service discipline will be seen to exhibit performance almost as good as FCFS but with much reduced communication and coordination overhead. It should be noted that previous researchers (see, e.g., [5]–[8]) have considered relative performance of a few of these algorithms, although almost all attention has been confined to source-initiative schemes. To relate nomenclature, Level 1 has been referred to as *static*, Level 5 as *dynamic*, and Level 3 as *semidynamic*.

IV. A Performance Metric

We have already indicated some properties we would like a load sharing strategy to possess. Because of a multiplicity of possible performance measures, it is difficult to rank algorithms on any absolute basis. Nevertheless, in comparing different strategies in our taxonomy, we will point out some fundamental differences in their performance. We find it useful to develop a yardstick based on a metric called the *Q-factor (quality of load sharing factor)* that captures to first-order some important aspects of load sharing performance.

For the system we are considering, a good load sharing algorithm will tend not to allow any server to be idle while there are jobs awaiting processing in the system. It will also not discriminate against a job based on the particular source by which that job arrives. In this sense, one ideal of a global scheduler, as defined above, is one that behaves like a multiserver first come first served (FCFS) queue. Since jobs originated by a specified user may always arrive to a particular source, they may be subject to systematic discrimination, either favorable or unfavorable. Moreover, if the aim of a distributed system design is to provide effective resource pooling, a job's performance cannot be excused by pointing to the particular source from which it originates. Thus, we can view an overall level of performance as being attained only if this level or better is attained by jobs at every source.

This leads to our definition of the Q-factor of an algorithm A:

$$Q_A(\rho) = \frac{\text{mean response time over all jobs under FCFS}}{\sup_{\frac{1}{K\mu}\sum_{i=1}^{N}\lambda_i=\rho} \max_i \{\text{mean response time at } i\text{th source under algorithm } A\}}$$

where the response time of a job is defined to be the length of time from when the job arrives at the system until it departs from the system (via a source),

N = number of sources,
K = number of servers,
μ^{-1} = mean service time,
λ_i = arrival rate at the ith source,
ρ = aggregated utilization of system.

This measure provides a factor usually[3] between zero and unity which describes how closely the system comes to a multiserver FCFS system, as seen by every job stream. The larger the Q-factor, the better the performance. Of course, this measure does not take into account conditional (on service time requirement) or delay distributional information but instead attempts to expose the overall load sharing behavior of the system. In particular, it will detect inefficiencies such as servers being idle when there is work to be done and any bias in performance due to arrival on a particular source. We will also see that it is sometimes possible to evaluate the Q-factor for an algorithm even though that algorithm defies a complete queueing analysis. Although we will only consider systems in which all servers are identical, the definition and the evaluation of the Q-factor can easily be extended to include the case of heterogeneous servers. In Section VI we will make use of the Q-factor, as well as

[3]An algorithm can have a Q-factor larger than unity. One example is the well-known "shortest job first" algorithm.

TABLE I
A TAXONOMY FOR LOAD SHARING ALGORITHMS

Level of Information Dependency in Scheduling	Source-Initiative	Server-Initiative
1	server = f (source) e.g. source partition	source = f(server) e.g. server partition
2	server = f(source, ω)† e.g. random splitting	source = f(server, ω) e.g. random service
3	server = f(source, ω, sequence state) e.g. cyclic splitting	source = f(server, ω, sequence state) e.g. cyclic service
4	server = f(source, ω, sequence state, server busy/idle status) e.g. join idle before busy queue (cyclic offer)	source = f(server, ω, sequence state, source queue emptiness) e.g. cyclic service without polling empty sources.
5	server = f(source, ω, sequence state, server queue lengths) e.g. join the shortest queue (JSQ)	source = f(server, ω, sequence state, source queue lengths) e.g. serve the longest queue (SLQ)
6	server = f(source,ω, sequence state, server queue lengths, departure epochs of completed jobs at servers) e.g. JSQ with ties broken by last departure epochs at each server	source = f(server, ω, sequence state, arrival epochs of jobs at sources) e.g. FCFS
7	server = f(source, ω, sequence state, departure epochs of completed and remaining jobs at servers) e.g. FCFS	source = f(server, ω, sequence state, arrival epochs and execution times of jobs at sources) e.g. shortest job first

† ω is a randomly generated parameter

other measures to compare the performance of different algorithms.

V. CANONICAL ALGORITHMS AND PERFORMANCE ANALYSIS

In this section we will define and analyze the performance of some of the canonical scheduling algorithms that were cited as examples in the taxonomy of Table I. We consider the following algorithms, using the earlier defined notation N for the number of sources, and K for the number of servers. The local scheduling policies, i.e., the service disciplines at the sources and servers, are FCFS.

1) Source Partition ($N \geq K$): Level 1, Source-Initiative — The sources are partitioned into groups and each group is served by one server. Each source in a group sends its jobs to the assigned server.

2) Server Partition ($N \leq K$): Level 1, Server-Initiative — The servers are partitioned into groups and each group of servers serves one source. A FCFS queue is formed at each source and jobs are removed one at a time by one of the available assigned servers. This can be considered to be a dual of source partition.

3) Random Splitting: Level 2, Source-Initiative — Each source distributes jobs randomly and uniformly to each server, i.e., it sends a job to one of the K servers with probability $1/K$.

4) Random Service: Level 2, Server-Initiative — Each server visits sources at random and on each visit removes and serves a single job. After each visit it selects the next source to be visited randomly and uniformly. This can be considered a dual to random splitting. A simple variation on this algorithm which may reduce communication overhead would have the server remove a batch of up to B jobs on each visit.

5) Cyclic Splitting: Level 3, Source-Initiative — Each source assigns its ith arriving job to the i(mod K)th server.

6) Cyclic Service: Level 3, Server-Initiative — Each server visits the sources in a cyclic manner. When a server visits the ith source it removes and serves a batch of up to B jobs and may then return to revisit the ith source for up to a total of V visits or until the source queue becomes empty. It then moves to the next source queue in a predetermined sequence. The algorithm has two parameters B and V and a visit sequence which may be different for each server.[4] The main cases we will consider have $B = \infty, V = 1$; $B = 1, V = \infty$ (often called exhaustive cyclic service), and $B = 1, V = 1$ (a case of limited cyclic service). Another variation we will consider will be referred to as "$B = 1$, gated" — this means the server revisits a source queue until the queue empties or the server finds a job next in queue that arrived to the source after the server first arrived to that source in a given cycle. This class of algorithms can be considered as dual to cycle splitting.

7) Join the Shortest Queue (JSQ): Level 5, Source-Initiative — Each source independently sends an arriving job to the server that has the least number of jobs (including the one in service). Ties are broken randomly and uniformly over tied servers.

8) Serve the Longest Queue (SLQ): Level 5, Server-Initiative — A dual to JSQ is the algorithm in which whenever a server becomes available it serves a job from the longest source queue. Again ties are broken randomly and uniformly.

9) First Come First Served (FCFS): Level 6, Server-Initiative or Level 7, Source-Initiative — All servers are applied to form an overall multiserver FCFS system. As shown in Table I, it can be considered server-initiative or, alternatively, source-initiative if more information is made available.

10) Shortest Job First: Level 7, Server-Initiative — Servers select the currently waiting job that has the smallest service time requirement.

The performance analysis of the algorithms we have defined leads to many unsolved problems in queueing theory. Some previous work does exist. References [2]–[4] considered the static (Level 1) algorithms and optimized chosen performance indices using mathematical programming or network flow techniques. Random and cyclic splitting algorithms and their variations have been considered in [5]–[9].

[4]The advantage in using a different schedule sequence for each server is discussed in [25].

TABLE II
POISSON ARRIVALS, EXPONENTIAL SERVICE TIME DISTRIBUTION, $h_o = h_e = 0$, N SOURCES, K SERVERS

Algorithm	Queueing Model	Solution Techniques
Source Partition	$M/M/1$	exact analysis (see, e.g., [13])
Server Partition	$M/M/C$	exact analysis (see, e.g., [13])
Random splitting	$M/M/1$	exact analysis (see, e.g., [13])
Cyclic Splitting	$\sum_{i=1}^{N} E_K(\lambda_i)/M/1$	simulation, approximate analysis (Appendix B-1)
Cyclic service ($B = 1, V = \infty$)	cyclic service queue	exact analysis for Q-factor (Appendix B-3)
Cyclic service ($B = 1, V = 1$)	cyclic service queue	simulation, heavy traffic analysis for Q-factor (Appendix B-4), exact analysis if $K = 1$, $N = 2$ [14]*
Cyclic service ($B = 1$, gated)	cyclic service queue	simulation, exact analysis if $K = 1$ [15]*
Join the shortest queue		simulation, approximate analysis (Appendix B-2), exact analysis if $K = 2$ [11]*
Serve the longest queue		exact analysis for Q-factor (Appendix B-3)
FCFS	$M/M/K$	exact analysis (see, e.g., [13])
shortest job first		simulation, exact analysis if $K = 1$ [16]*

TABLE III
POISSON ARRIVALS, DETERMINISTIC SERVICE TIME, $h_o = h_e = 0$, N SOURCES, K SERVERS

Algorithm	Queueing Model	Solution Techniques
Source Partition	$M/D/1$	exact analysis (see e.g. [13])
Server Partition	$M/D/C$	exact analysis (see e.g. [17])
Random Splitting	$M/D/1$	exact analysis (see e.g. [13])
Cyclic Splitting	$\sum_{i=1}^{N} E_K(\lambda_i)/D/1$	simulation, approximate analysis (Appendix C-1 and -5), exact analysis if $N=1$ (Appendix C-5)
Cyclic service ($B=1$, $V=\infty$)	Cyclic service queue	simulation
Cyclic service ($B=1$, $V=1$)	"	simulation, heavy traffic analysis for Q-factor (Appendix C-4)
Cyclic service ($B=1$, gated)	"	simulation, exact analysis if $K=1$ [15]*
Join-the-shortest-queue		simulation
Serve-the-longest-queue		simulation
FCFS	$M/D/K$	exact analysis [17]
shortest-job-first		"

TABLE IV
POISSON ARRIVALS, DETERMINISTIC SERVICE TIME, $h_o = h_e = 0$, N SOURCES, $K \to \infty$ SERVERS

Algorithm	Queueing Model	Solution Techniques
Random Splitting	$M/D/1$	exact analysis (see e.g. [13])
Cyclic Splitting	$\sum_{i=1}^{N} D_i/D/1$	exact analysis [19]
Cyclic service ($B=1$, $V=\infty$)	Cyclic service queue	exact analysis for Q-factor (see Appendix C-6)
Cyclic service ($B=1$, $V=1$)	"	"
Cyclic service ($B=1$, gated)	"	"
Serve-the-longest-queue		"
FCFS	$M/D/\infty$	"

Dynamic algorithms have been found much more difficult to analyze, and in many cases simulation has been necessary. For example, [5], [11] have considered JSQ and [10] studies a "diffusion" algorithm (see Appendix A). Most prior analysis has concentrated on source-initiative algorithms and mainly considered average performance over all jobs excluding any fairness or uniform acceptability measures.

Before describing our analysis, we state assumptions about the transport delay of the communication medium and node processor overhead involved in interprocessor communication (IPC). The transport delay of the communication medium will be assumed to be zero. This is reasonable for many applications and current local area interconnect technology which usually has ample bandwidth and provides rapid message passing between processors. The IPC overhead is assumed to consist of a per message processing time of h_o and h_e at the source and server nodes, respectively. These burdens are supposed to take into account system call, context switch, and communication medium management. Values of h_o and h_e observed in practice are on the order of one or several milliseconds for typical microprocessor technology and are often relatively independent of message length. Thus, the overhead of transferring a bit of scheduling information might well be comparable to the overhead of moving a job. We will consider separately the cases where IPC overhead is negligible and nonnegligible.

The performance of the ten canonical algorithms described above will be analyzed under various assumptions.

1) Poisson arrivals, exponential service time distribution, $h_o = h_e = 0$, N sources, K servers.

2) Poisson arrivals, deterministic service time distribution, $h_o = h_e = 0$, N sources, K servers.

3) Poisson arrivals, deterministic service time distribution, $h_o = h_e = 0$, N sources, $K \to \infty$ servers.

4) Poisson arrivals, hyperexponential service time distribution, $h_o = h_e = 0$, N sources, K servers.

5) Batched Poisson arrivals (batch size a multiple of K), exponential service time distribution, $h_o = h_e = 0$, N sources, K servers.

6) Poisson arrivals, exponential service time distribution, $h_o = h_e > 0$, N sources, K servers.

In each case we will assume that each of the servers has a mean job processing rate of μ and the processing time at sources is negligible. For cases 1)–6), see Tables II–VII which summarize, respectively, the cases considered and the analysis methods used to obtain the results. For completeness, we also include in Tables II–VII some other relevant analysis methods (marked with *). These tables refer extensively to the literature and to Appendix B which contains details of further queueing analysis.

TABLE V
POISSON ARRIVALS, HYPEREXPONENTIAL SERVICE TIME DISTRIBUTION, $h_o = h_e = 0$, N SOURCES, K SERVERS

Algorithm	Queueing Model	Solution Techniques
Source Partition	$M/H_2/1$	exact analysis [13]
Server Partition	$M/H_2/C$	simulation, exact analysis [18]*
Random Splitting	$M/H_2/1$	exact analysis [13]
Cyclic Splitting	$\sum_{i=1}^{N} E_K(\lambda_i)/H_2/1$	simulation, exact analysis, if $N=1$ [18]*
Cyclic service ($B=1$, $V=\infty$)	Cyclic service queue	simulation
Cyclic service ($B=1$, $V=1$)	"	simulation, heavy traffic analysis for Q-factor (Appendix C-4)
Cyclic service ($B=1$, gated)	"	simulation, exact analysis if $K=1$ [15]*
Join-the-shortest-queue		simulation
Serve-the-longest-queue		simulation
FCFS	$M/H_2/K$	simulation, approximate analysis [46], exact analysis [18]*.

TABLE VI
BATCHED POISSON ARRIVALS, BATCH SIZES ARE MULTIPLES OF K (THE NUMBER OF SERVERS), EXPONENTIAL SERVICE TIME DISTRIBUTION, $h_o = h_e = 0$, N SOURCES

Algorithm	Queueing Model	Solution Techniques
Cyclic Splitting	$M^{[x]}/M/1$	exact analysis (Appendix C-7)
Cyclic service ($B=1$, gated)	cyclic service queue	simulation
FCFS	$M^{[x]}/M/K$	exact analysis [47]

TABLE VII
POISSON ARRIVALS, EXPONENTIAL SERVICE TIME DISTRIBUTION, N SOURCES, K SERVERS, $h_o = h_e > 0$

Algorithm	Queueing Model	Solution Techniques
Cyclic Splitting		simulation
Cyclic service ($B=\infty$, $V=1$)		simulation, approximate analysis [25]*

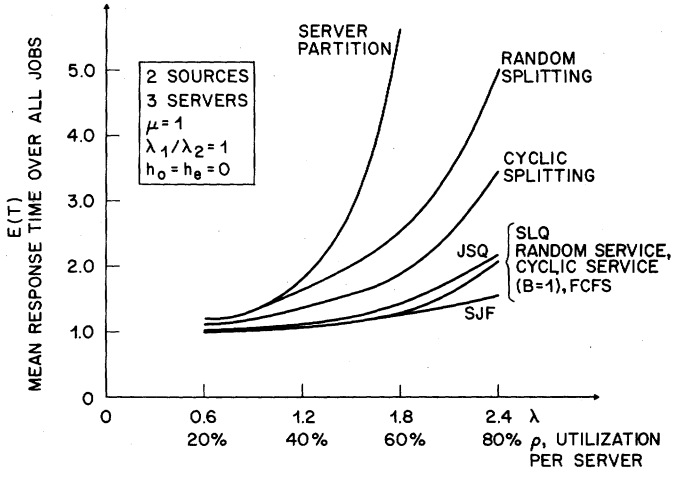


Fig. 2. Mean response time (over all jobs) as a function of utilization of service—balanced loading.

VI. PERFORMANCE ANALYSIS RESULTS AND COMPARISONS

We begin with the case where there is no IPC overhead (i.e., $h_o = h_e = 0$). This case, while idealized, is of considerable importance for several reasons. It approximates the case where IPC overheads are negligible in comparison to the work required to execute the job. More importantly, it gives insight as to what performance gains are, in principle, achievable with different algorithms. If, and only if, a significant performance gain is indicated, the cost of implementing the algorithm (development and execution) needs to be considered. In some cases, special measures (e.g., more efficient operating system procedures, hardware assists such as front ends) may be justified to make the algorithm implementable so that performance gains are achieved.

A. Performance Averaged Over All Jobs

We find that server-initiative algorithms outperform source-initiative algorithms using the same level of information. The main reason for this is that most server-initiative algorithms do not allow a server to become idle when there are jobs waiting in the system—at the very least a server-initiative algorithm will utilize server idle time to search for new work.

Fig. 2 shows the mean response time over all jobs for various algorithms as a function of utilization for the case $N = 2, K = 3, \lambda_1 = \lambda_2$. First note that the cyclic service ($B = 1$), random service, and SLQ algorithms produce the same mean response time as FCFS. This is because these algorithms simply result in an interchange of order of processing from FCFS and thus (by Little's Law [13], [20], [28]) do not change the mean delay averaged over all jobs. Of the other algorithms shown, SJF gives the best mean delay, as expected, being the only algorithm considered that has knowledge of jobs' service times. The poorer performance of JSQ compared to FCFS is, of course, attributable to the inadequacy of using number in queue as a load indicator. That results in the possibility that a job can wait at a server while

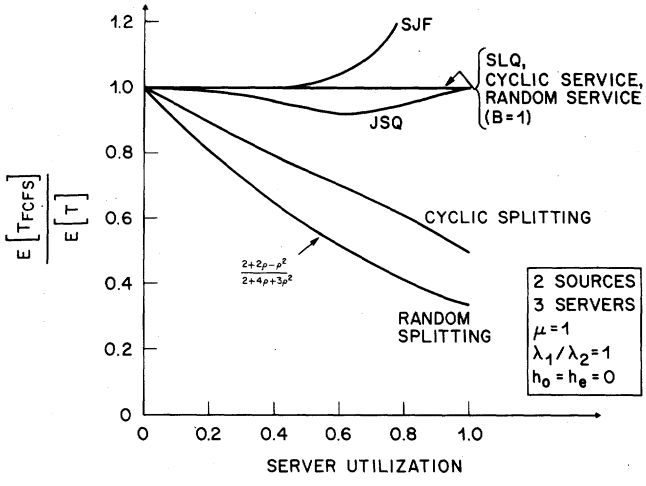


Fig. 3. Fig. 2 redrawn normalized with respect to FCFS.

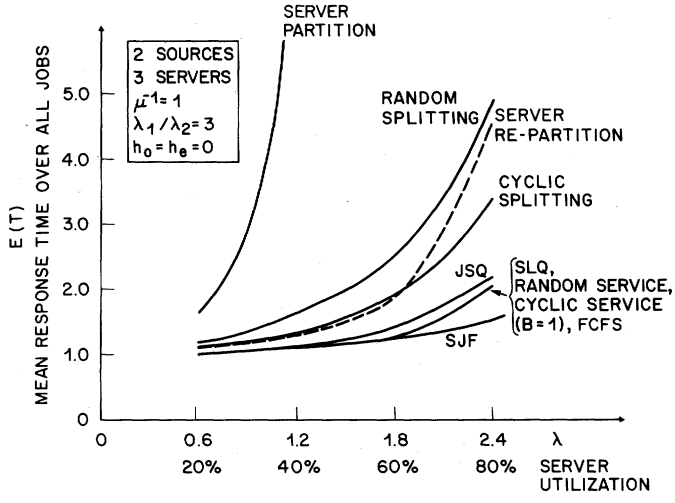


Fig. 4. Mean response time over all jobs when the load on the sources is unbalanced.

another server is idle. It is interesting to note that, as shown in Table I, if we allow the sources to know which server has the least backlog of work (although this is seldom a practical proposition), we can implement FCFS as a source-initiative algorithm. The phenomenon of jobs waiting while a server is idle and the resulting degraded performance is even more pronounced in the random and cyclic splitting algorithms, which have no knowledge of the server states. The cyclic splitting algorithm outperforms the random splitting algorithm because of the increased regularity of the job stream at the server induced by the cyclic splitting. If we normalize the mean response time of jobs under various algorithms with respect to FCFS, we obtain Fig. 3.

In Fig. 4 we unbalance the load between the two sources and reexamine performance. This imbalance does not affect overall performance of the server-initiative and JSQ algorithms and has only a small effect on the random and cyclic splitting algorithms. It has a major effect on the server partition algorithm which is unable to adapt to the imbalance and one server quickly becomes saturated. This effect can be reduced by repartitioning the servers as shown, but the repartitioning does not happen automatically.

Another interesting aspect of the failure of source-initiative algorithms to reflect actual processing times is the effect of service time distribution on overall performance. Fig. 5 illustrates the effect on mean overall response time relative to FCFS as service time distribution varies from deterministic through exponential to hyperexponential. The overall performance of SLQ and cyclic service ($B = 1$) are not affected (when normalized to FCFS) by service time distribution, but the performance of the random and cyclic splitting algorithms degrade rapidly as more variability is introduced into the service time distribution.

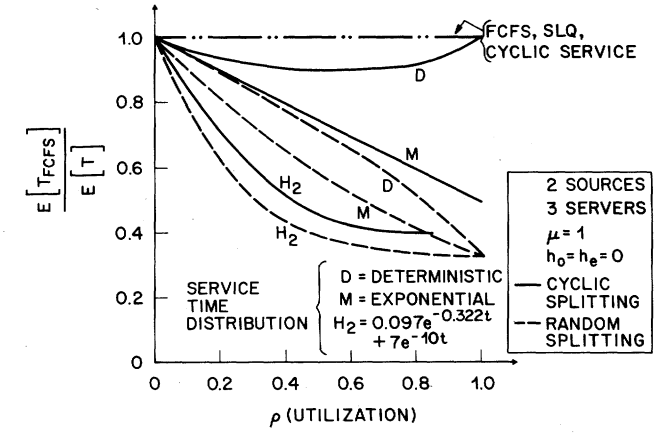


Fig. 5. The effect of service time distribution on the overall mean response time.

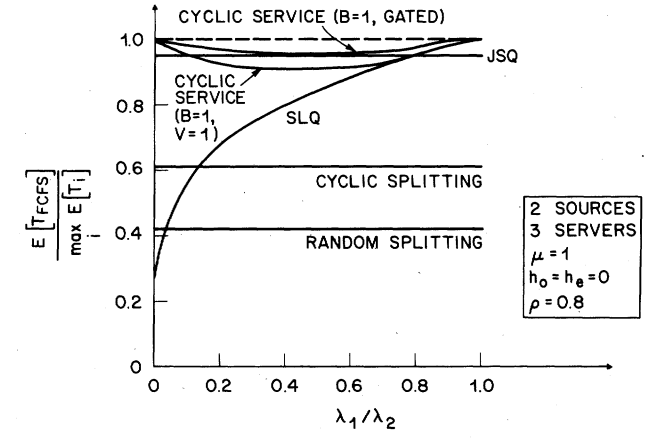


Fig. 6. The effect of load distribution on mean response time at the worst performing source—$\rho = 0.8$.

B. Load Sharing Performance

We now evaluate the ability of the algorithms to share load over servers regardless of the distribution of work over sources. We find that server-initiative algorithms usually have a higher Q-factor than the source-initiative algorithms using the same level of information.

Figs. 6 and 7 show the delay at the worst performing source as the load distribution varies between two sources for a fixed total load. While server-initiative algorithms are seen to be superior, they are more sensitive to load distribution than source-initiative algorithms. An interesting result is that the cyclic splitting curve is almost flat—this is consistent

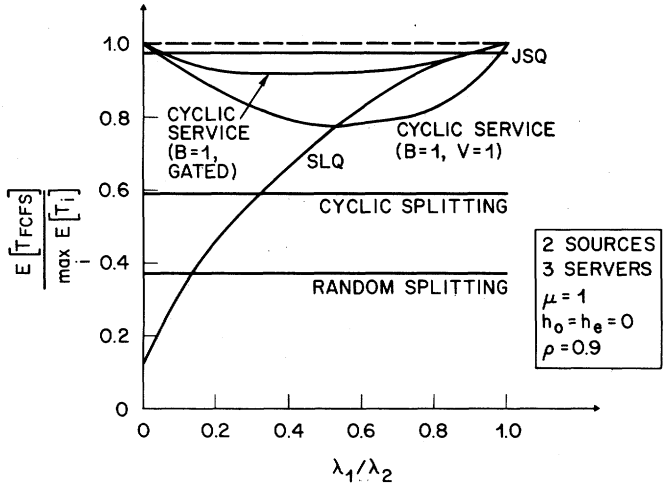


Fig. 7. The effect of load distribution on mean response time at the worst performing source—$\rho = 0.9$.

with the observations of [21]. Note that SLQ degrades rapidly when load is unbalanced. This is because the lightly loaded queue receives low priority in gaining service. In the limit, a rare arrival at the lightly loaded queue must wait until the heavily loaded queue has less than K jobs present (i.e., one of the K servers becomes free) before receiving service.

In Figs. 8–10 we use the Q-factor figure of merit introduced earlier to allow a performance summary to be plotted as a function of ρ.[5] Note that in computing the Q-factor we must find the combination of loadings (λ_i's) which cause the worst performance. For most algorithms the Q-factors degrade rapidly as $\rho \rightarrow 1$. The only source-initiative algorithm that holds up adequately well is JSQ. The cyclic service algorithms show reasonable Q-factors except at very high utilization, with the "$B = 1$, gated" version showing the best performance. The reader should note that algorithms that have identical performance when viewed over all jobs have now been delineated by the Q-factor (compare Fig. 3 to Fig. 8). Figs. 9 and 10 treat the deterministic and hyperexponential service time distributions, respectively, and show that load sharing performance of all algorithms degrades as service time variability increases. But the performance of the server-initiative algorithms is less sensitive to the service time variability.

The performance of source-initiative algorithms such as random and cyclic splitting also degrade as the number of servers becomes large (Fig. 11). On the other hand, server-initiative algorithms such as cyclic service approach the ideal of load balancing.

To investigate the effect of "bursty" arrivals on the performance of these algorithms, we consider batch arrivals at sources. The results are shown in Fig. 12. We observe that both cyclic splitting and cyclic service ($B = 1$, gated) are relatively unaffected by batches, with cyclic service showing slightly superior performance.

[5]Recall that the Q-factor is designed to compare performance of algorithms over a range of system loadings. It does not necessarily assert, for example, one algorithm is uniformly better than another under all conditions.

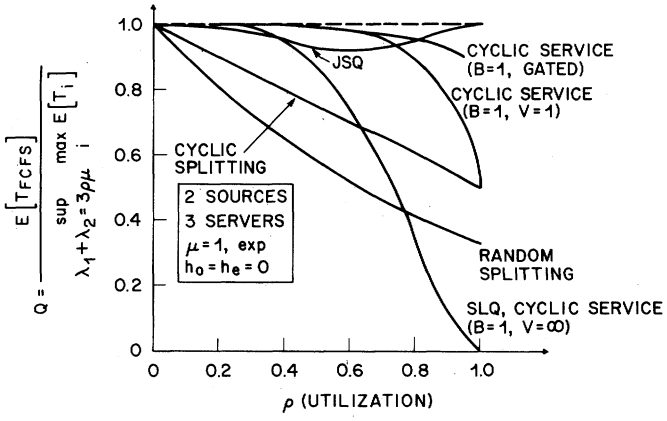


Fig. 8. Q-factors of algorithms as a function of ρ when service time distribution is exponential.

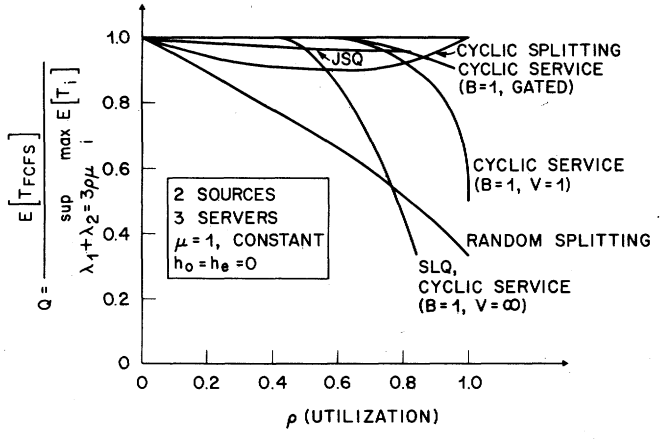


Fig. 9. Q-factors of algorithms as a function of ρ when service time distribution is deterministic.

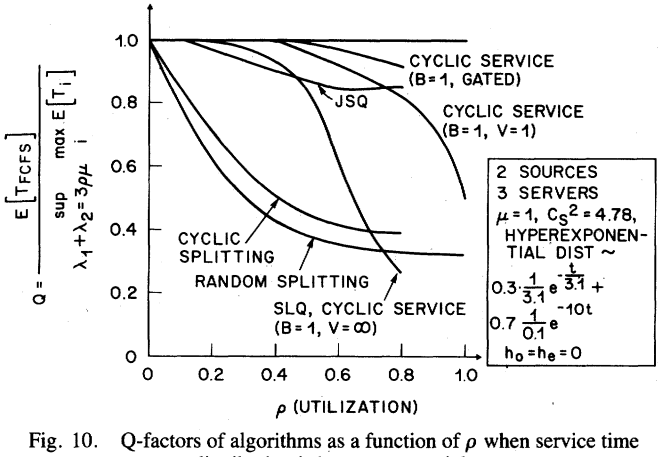


Fig. 10. Q-factors of algorithms as a function of ρ when service time distribution is hyperexponential.

C. *Effect of IPC Overhead on Algorithms*

It was argued at the beginning of this section that performance analysis with assumptions of negligible IPC overhead can still offer considerable insight into design issues. In the remainder of this paper we will limit ourselves to demonstrating a few effects when IPC overhead is not negligible. It is rather difficult to conduct an unbiased and complete performance evaluation when IPC overhead is taken into ac-

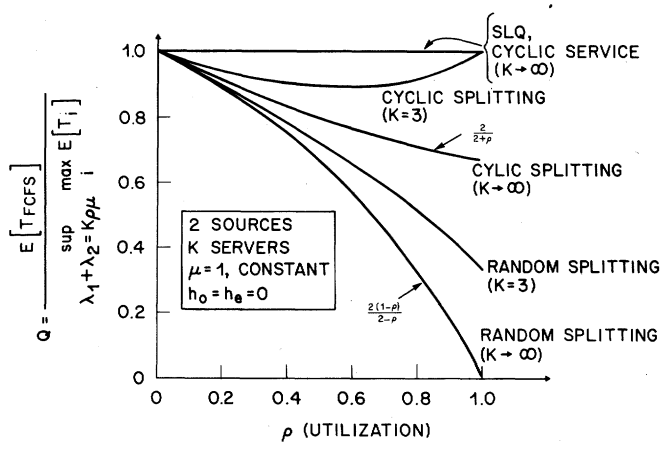


Fig. 11. Q-factors of algorithms as a function of ρ when the number of servers increases.

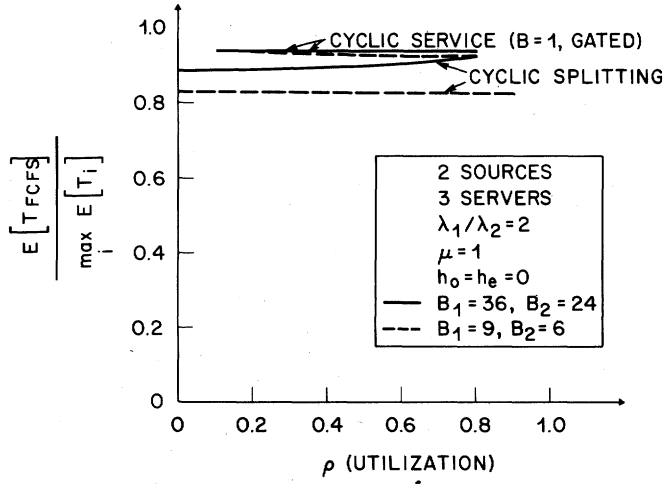


Fig. 12. The effect of batch size of arrivals on the mean response time of jobs at the worst performing source.

count. The amount of overhead introduced by an algorithm is implementation dependent and therefore highly subjective. For example, while JSQ outperforms other source-initiative algorithms it would seem to involve a large IPC overhead. For this reason we select for illustration the cyclic splitting and cyclic service algorithms which have been shown, from the results above, to be promising representatives of the source- and server-initiative approaches. The assumptions on the IPC overhead for these two are as follows.

For the cyclic splitting algorithm, the ith job arriving to a source is sent to the $i \pmod K$th server. This requires a write per job at the source and a read at the server. On completing service, a server must write each job back to the originating source and that source has to read the result. Thus, the IPC overheads per job are $2h_o$ at the source and $2h_e$ at the server. These overheads are assumed to be deterministic.

For the cyclic service ($B = \infty, V = 1$) algorithm, the server polls a source by writing a message to that source. The source receives (by reading) the message and writes back a "batch" of all accumulated jobs ($B = \infty$) to the polling server which receives them with a single read. If the batch is nonempty, the server executes the batch of jobs and writes them with a single write back to the originating source which has to read the results; otherwise, it polls the next source, etc. Thus, to process a batch, the server has done two writes and a read while the source has done two reads and one write. The IPC overheads are thus $3h_o$ at the source and $3h_e$ at the server per (nonempty) batch of jobs. For this example we are assuming that the IPC overhead of "polling" equals that of moving a job, as discussed in Section V.

When IPC overhead is zero, cyclic service has better performance. But as overhead increases, Fig. 13 shows that cyclic splitting can be advantageous because it has no polling latency and may have smaller per job overhead. Although it is not shown in Fig. 13, as overhead increases even further, cyclic service eventually becomes preferable as a consequence of its batching effect. Fig. 14 shows a similar effect. For a fixed value of overhead, as the arrival rate increases, cyclic splitting shows some initial advantage which is eventually overridden by the decreasing per job overhead of cyclic service.

It could be argued that batching effects can also be incorporated to advantage in a cyclic splitting algorithm using, for example, a time-out mechanism at the sources. But then latency effects also appear in the cyclic splitting algorithm and performance comparisons become ambiguous. This example illustrates some of the difficulties of carrying out an objective comparison of algorithms in the presence of IPC overhead.

VII. Conclusions

With the enormous scope and complexity of the distributed scheduling problem, we have only been able to touch on a subset of important issues. We have concentrated on what is obviously a crucial component — the load sharing strategy — but we remind the reader that there are also other important components of this problem including data synchronization and concurrency control, maintenance, fault detection/recovery, etc. One issue that we did cite in Section I was that of failure tolerance. It should be noted that of the algorithms we have considered, the server-initiative varieties degrade more gracefully under passive failures since a failed server will cease to request work.

In our analysis of load sharing performance we have only been able to consider a simple class of distributed systems and a small subset of all possible parameter ranges. Much further work needs to be done. But we have demonstrated some basic properties of different approaches. Some of our conclusions are as follows.

i) Widely varying performance, both from an overall and individual sources' viewpoint, is obtained with different strategies — the load sharing algorithm is a critical design decision.

ii) With the same level of information available, server-initiative algorithms have the potential of outperforming source-initiative algorithms. This superiority is demonstrated for the case of negligible IPC overhead but whether this holds true in another situation requires examination of

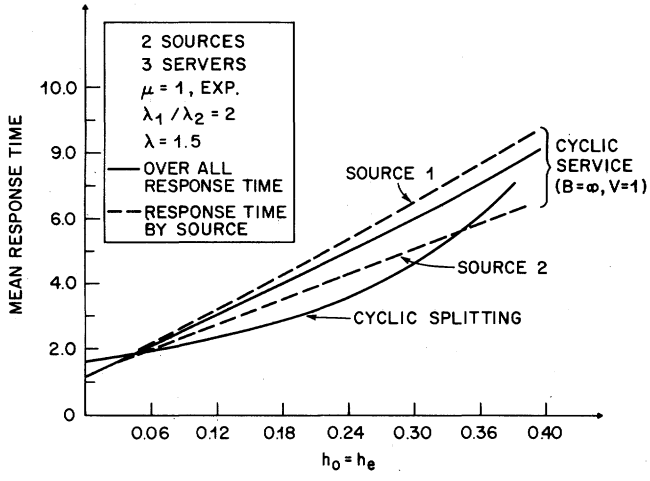


Fig. 13. The mean response of jobs as a function of the interprocessor communication overhead.

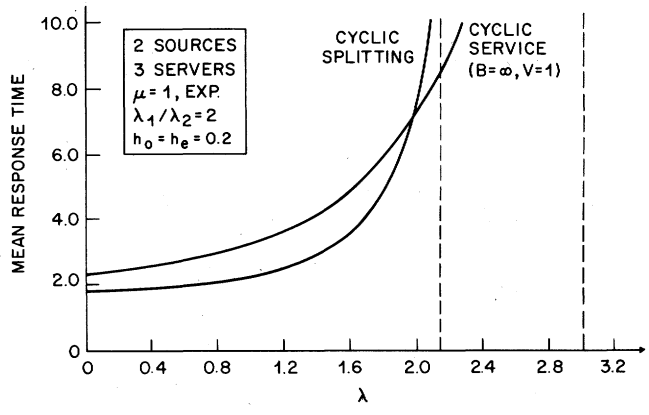


Fig. 14. The mean response time of jobs as a function of load where the interprocessor communication overhead is nonnegligible.

the communication cost involved.

iii) The use of the Q-factor provides a useful yardstick for comparing load sharing performance of different algorithms.

iv) Algorithms that have previously received little attention in the literature, e.g., multiserver cyclic service, can in some circumstances provide effective load sharing at fairly low communication cost.

Of course, these observations cannot be applied out of context. For example, two algorithms, one source-initiative and one server-initiative, may use the same "level of information" but involve considerably different amounts of communication if N is much larger than K or vice versa. And in applications where IPC overhead is significant, implementation options such as batching jobs at nodes need to be pursued.

Throughout this paper we have concentrated on scheduling strategies based on a communication medium that provides only rudimentary message passing. But if we were instead to construct a communication medium with capabilities such as broadcasting, virtual addressing, conditional addressing, and selective or conditional reception, we open up a much wider spectrum of scheduling options.

Appendix A
Literature Survey

Load sharing algorithms have frequently been classified as either *static* or *dynamic*. The static algorithms distribute the load according to some rules that are set *a priori*.

Static algorithms that allocate servers permanently or semipermanently to arriving job streams can be found in [2]–[4] where load sharing is formulated as a mathematical programming or network flow problem. Solutions are obtained by performing optimization against some chosen performance index that assumes a known constant workload. This approach may require an update of server allocations if arrival rates change. A *random splitting* algorithm that distributes jobs among all servers according to a given probability distribution can be found in [5]. *Cyclic splitting* algorithms in which jobs are distributed among all servers according to some cyclic schedule can be found in [7], [8].

The major drawback of static algorithms is that they do not respond to short-term fluctuations of the workload. Dynamic schedules attempt to correct this drawback but are more difficult to implement and to analyze. There are at least six types of these schedules found in the literature (for other detailed surveys, see [27], [49]).

1) Hierarchical: This type of algorithm can be found in the resource management of the Stony Brook Multicomputer [30], the MicroNet [31], [32], and the X-tree system [33] where schedulers are organized into a logical control hierarchy. Each scheduler is responsible for either some lower level schedulers or servers (sometimes called "workers"). Load distribution is done by requesting and reserving resources up and down the hierarchy until sufficient resources are obtained. The scheduling decisions are usually carried out unilaterally with one scheduler at the root of the hierarchy. The load sharing problem has not been specifically addressed.

2) Sequential: Examples of systems include the ARAMIS [34] and the Delta [26] where the nodes of the system form a virtual ring. Each node has its own scheduler that is responsible for accessing shared resources in the system for the users attached to that node. A control token is passed around the virtual ring to enforce a serial operation of the schedulers in allocating or deallocating shared resources. The emphasis of the scheme is to provide some concurrency control rather than load sharing. It may be viewed as a dynamic centralized scheduling algorithm since at any given time there is only one view of the system state — the view of the scheduler that holds the token. Note that some reliable mechanism has to be provided so that the token will not be lost indefinitely nor be duplicated.

3) Diffusion: Examples of this class of algorithms can be found in MuNet [35], CHoPP [36], and the most recent version of Roscoe [10]. The load sharing effort is accomplished in these systems by having the neighboring nodes communicate and cooperate with each other so that the load migrates from heavily to lightly loaded nodes. One advantage of this

algorithm is that the communication cost can remain relatively low even when the size of the system increases. However, the rate of diffusion and the stability of the system are not fully understood. Some simulation results that demonstrate the superiority of the scheme over static scheduling were documented in [10].

4) Contract Bidding: This popular algorithm can be found in DCS [37] and CNET [38]. There can be two procedures: *normal* and *reversed*. In the normal contract bidding procedure, sources that distribute the load will broadcast a "request for bids" to all servers. On receiving the request for bids, a server may submit a bid for the work according to some criteria, e.g., its work queue length, availability of its resources, etc. Later the requesting source will choose one server (typically the lowest bidder) to process the work based on the bids it has received. In the reversed contract bidding procedure, idle servers will send messages to sources to indicate their willingness to take on work rather than waiting for "request for bids." The multiple cyclic server system analyzed in [25] can be considered as a case of the reversed contract bidding scheme. The performance of this class of algorithm depends on the appropriateness of the choice of the bid and communication overhead.

5) Dipstick: Examples of this type of algorithm may be found in the ADAPT [39] and an earlier version of Roscoe [40]. Similar mechanisms may also be found in ICNS [41] and the POGOS [27]. Under this algorithm, a node always tries to execute a job locally before it sends that job to other nodes. The criterion used by a node to decide whether to accept and execute a job is typically based on some indicator of its load. Often, some minimum and maximum values of the indicator are prespecified. If a node's load is below the minimum, it may broadcast to others that it is willing to take additional load. If a node's load exceeds the maximum, it will attempt to send incoming jobs to other nodes. Provisions have to be made for those jobs that have been given a "run-around." It appears to be difficult to choose an appropriate criterion for accepting or rejecting work by a node so that the system is stable [27]. Note also that the communication overhead can be severe, especially when the system is heavily loaded.

6) State feedback: Examples include DFMP [42], the Purdue University ECN [43], and the classical "Join the Shortest Queue" method and its variants [5], [6], [44]. A scheme based on Bayesian decision theory has also been proposed and analyzed [45]. In this approach an estimate of a system state such as processor queue lengths is maintained and is periodically updated. The algorithm simply routes the work to the currently "least congested" server. One problem with this approach is the potentially high cost of obtaining the required state information (delay and communication overhead, etc.). It is known that excessive delays in updating the information can cause instability of the system. Another problem is that a chosen load indicator, such as queue length, is often inadequate especially when the servers possess a multiplicity of resources. Note that the join the shortest queue method may also be considered as a form of normal contract bidding where the bid is in terms of the queue length.

APPENDIX B
QUEUEING ANALYSIS

1. Approximating the Superposition of Renewal Processes [21]

The superposition of N renewal processes ($\Sigma_{i=1}^{N} GI_i$) may be approximated by a renewal process as follows.

Step 1: Let the approximating mean interevent time be $1/\lambda = 1/\Sigma_{i=1}^{N} \lambda_i$ where λ_i^{-1} is the mean interevent time of the ith component process.

Step 2: Let the approximating coefficient of variation be either Approximation I:

$$c_I^2 = \omega_1 c_a^2 + (1 - \omega_1) c_s^2,$$

$$\omega_1 = \frac{1}{1 + 6(1 - \rho)^{2.2} N}$$

or Approximation II:

$$c_{II}^2 = \omega_2 c_a^2 + (1 - \omega_2) c_m^2,$$

$$\omega_2 = \frac{1}{1 + 2.1(1 - \rho)^{1.8} N}$$

where

$$c_a^2 = \frac{1}{\lambda} \sum_{i=1}^{N} \lambda_i c_i^2 \qquad \text{(the asymptotic approximation method)},$$

$$c_m^2 = 1 \qquad \text{(the Markov approximation method)},$$

and c_s is the coefficient of variation of the stationary interval distribution of the superposed process which is given by

$$G(t) = 1 - \frac{1}{\lambda} \sum_{i=1}^{N} \lambda_i [1 - F_i(t)] \left\{ \prod_{\substack{j=1 \\ j \neq i}}^{N} \lambda_j \int_t^{\infty} (1 - F_j(x))\, dx \right\}$$

where $F_j(\cdot)$ is the probability distribution function of the interevent time of the jth component process.

Step 3: Depending on whether $c_I (c_{II})$ is greater or less than one, fit a "convenient" distribution to the approximating renewal process. In [21], Albin has chosen to fit an M^d-*distribution* (shifted Exponential) if $c_I (c_{II}) \leq 1$ and an H_2^b-*distribution* (balanced mean hyperexponential) if $c_I (c_{II}) > 1$.

Remarks: a) Both approximations have been tested [21] for queues with exponential service time distributions by comparing the approximating mean waiting time against the simulated results. Good accuracies were reported for both approximations. Further discussions and improvements of these approximations can be found in [50].

b) Suppose the component processes are $E_K(\lambda_i)$ (i.e., of same number of phases with possibly different rate λ_i), $\Sigma_{i=1}^{N} \lambda_i = \lambda$. Then Approximation II (that is based on the asymptotic approximation and the Markov approximation) will produce a coefficient of variation independent of the ratios of λ_i/λ for the approximating renewal stream. This is due to the fact that the coefficient of variation of $E_k(\lambda_i)$ is $1/\sqrt{K}$ and is independent of λ_i. Therefore, the good accuracy of Approximation II implies that the overall mean delay in a $\Sigma_{i=1}^{N} E_K(\lambda_i)/M/1$ would be insensitive to the ratios λ_i/λ

for a given $\Sigma_{i=1}^{N} \lambda_i = \lambda$. We do observe such an insensitivity in our simulation results when we study the cyclic splitting algorithm.

2. *Approximate Analysis of JSQ Discipline*

Let the arrival process be Poisson of rate λ. Let the number of queues (servers) be K. Let the service time distribution of jobs be exponential of mean $\mu^{-1} = 1$. It is known that (see, e.g., [13]) the mean waiting time in an $M/M/K$ FCFS system is given by

$$\overline{W}_{\text{FCFS}} = \frac{\lambda^K}{d(\lambda)}$$

where $d(\lambda) = (K - \lambda)\{K!(1 - \lambda/K)\Sigma_{k=0}^{K-1} \lambda^k/k! + \lambda^K\}$.

We now approximate the mean waiting time in an $M/M/K$ JSQ system by

$$\overline{W}(\lambda) = \frac{n(\lambda)}{d(\lambda)}$$

where $n(\lambda) = \Sigma_{i=0}^{K+1} a_i \lambda^i$ is to be determined by the jth order derivatives

$$\overline{W}_{\text{JSQ}}^{(j)}(0), \qquad j = 0, 1, 2, \cdots, K$$

and

$$\overline{W}_{\text{JSQ}}(\lambda), \qquad \lambda \to K.$$

a) Light Traffic Analysis of JSQ:[6] Consider the JSQ system with $\lambda \to 0$. The probability of waiting is of the order of the probability that an arriving customer finds K in the system in service. This latter probability is given by the probability that an arrival occurred before any of K jobs in service could depart and the previous job arrived before any of $K - 1$ jobs in service could depart, etc. Thus, the probability of waiting is

$$\frac{\lambda}{\lambda + \mu K} \cdot \frac{\lambda}{\lambda + \mu(K - 1)} \cdot \cdots \cdot \frac{\lambda}{\lambda + \mu} + o(\lambda^K)$$

and

$$\overline{W}_{\text{JSQ}}(\lambda) = \frac{1}{\mu} \prod_{j=1}^{K} \frac{\lambda}{\lambda + j\mu} + o(\lambda^K).$$

Thus,

$$\overline{W}_{\text{JSQ}}^{(n)}(0) = \begin{cases} 0, & n < K \\ \dfrac{1}{\mu^{K+1}}, & n = K. \end{cases}$$

b) Heavy-Traffic Analysis: It has been shown (see, e.g., [23]) by diffusion analysis that for the $M/M/K$ queueing system

$$\frac{\overline{W}_{\text{JSQ}}(\lambda)}{\overline{W}_{\text{FCFS}}(\lambda)} \to 1, \qquad \text{as } \lambda \to K.$$

[6]This is based on the ideas of [22].

From the light traffic result, we have that

$$a_i = 0, \qquad i = 0, 1, \cdots, K - 1$$

$$a_K = K.$$

From the heavy traffic result, we have

$$a_{K+1} = \frac{1}{K} - 1.$$

Hence, the approximate mean waiting time in a $M/M/K$ JSQ system is

$$\overline{W}_{\text{JSQ}}(\lambda) \approx \frac{\lambda^K \left[K - \left(1 - \frac{1}{K}\right)\lambda\right]}{d(\lambda)}.$$

Remark: This approximation has been compared to simulation results for $K = 2, 3, 5$. The maximum error (as a function of $\rho = \lambda/K\mu$) for the case where $K = 2$ was about 8 percent (at $\rho \approx 0.7$). The maximum error seems to increase as the number of servers increases. For the case where $K = 5$, we have seen an error of 25 percent. It appears that additional derivative information at $\rho \to 1$ would help to improve the approximation but we do not know of any such results. We have tried requiring that $\overline{W}_{\text{JSQ}}^{(1)}(\lambda)/\overline{W}_{\text{FCFS}}^{(1)}(\lambda)$ also tends to 1 as $\lambda \to K$ and obtained a $K + 2$ degree polynomial $n(\lambda)$ in our approximation. For the case $K = 5$, the approximation errors were reduced significantly (to about 7 percent at $\rho = 0.8$ and 3 percent at $\rho = 0.7$).

3. *Derivation of Q-Factors for SLQ and Cyclic Service ($B = 1, V = \infty$)*

For the SLQ ($K \geq 2$) and cyclic service ($B = 1, V = \infty$) algorithms, it can be argued that the Q-factor is attained when the ith source has an arrival rate $\lambda_i \to 0$ while all the rest of the jobs go to one of the remaining sources—the system behaves like a nonpreemptive priority queue. Under this condition, a tagged arriving job at the ith source waits if and only if all servers are busy. The mean waiting time given that the tagged job waits is the mean forward recurrence time of the K-server busy period where the latter is defined as the time between when all K servers become busy and the first time one of the servers becomes free. Under the assumption that the arrivals form a Poisson process and that the service time distribution is exponential, the K-server busy period is the same as the busy period of a $M/M/1$ system with a service rate K times that of the servers of the original system. Thus, the mean waiting time of jobs at the ith source is given by

$$Q_{\text{SLQ}} = Q_{\text{cyclic service } (B=1, V=\infty)}$$

$$= \frac{\dfrac{C(K, \lambda)}{K\mu(1 - \rho)} + \mu^{-1}}{\dfrac{C(K, \lambda)}{K\mu(1 - \rho)^2} + \mu^{-1}}$$

$$= (1 - \rho) \cdot \frac{C(K, \lambda) + K(1 - \rho)}{C(K, \lambda) + K(1 - \rho)^2}$$

where $C(K, \lambda)$ is the well-known Erlang-C formula [13], the probability that all K servers are busy in an $M/M/K$ system, and $\rho = \lambda/K\mu$.

4. Derivation of Q-Factor as $\rho \to 1$ for Cyclic Service ($B = 1, V = 1$)

First let

$$\lambda_1(\rho) \doteq \frac{K\mu\rho}{N} + (N-1)\epsilon,$$

$$\lambda_i(\rho) = \frac{K\mu\rho}{N} - \epsilon, \qquad i \neq 1,$$

and $\lambda(\rho) = \sum_{i=1}^{N} \lambda_i(\rho)$. Assume $Q(\rho) \to Q$ as $\rho \to 1$. Now queues $2, \cdots, N$ see servers which, before each service, take vacations that are dominated by the sum of $N-1$ service times of mean μ^{-1}. Thus, queues $2, \cdots, N$ each have effective service rates $\geq \mu/N$ for each of the K servers but $\lambda_i(\rho) \to K\mu/N - \epsilon$ as $\rho \to 1$. Thus, each of these queues is stable as $\rho \to 1$ (having queue length dominated by a stable $M/G/K$ queue), and queue 1 alone is unstable. Denote the mean sojourn time of queue i as $\overline{T}_i$ and the overall mean sojourn time as $\overline{T}$, i.e., $\lambda\overline{T} = \sum_{i=1}^{N} \lambda_i \overline{T}_i$. Then as $\rho \to 1$,

$$Q(\rho) \leq \frac{\overline{T}}{\overline{T}_1} = \frac{\lambda_1(\rho)}{\lambda(\rho) - \sum_{i=2}^{N} \lambda_i(\rho)\overline{T}_i/\overline{T}} \to \frac{1}{N} + \frac{(N-1)\epsilon}{K\mu}.$$

Thus, since $\epsilon > 0$ is arbitrary, we have that $Q \leq 1/N$.

Now Q could not be less than $1/N$. For assume the contrary and let $Q(\rho) \to Q = g/N, g < 1$ where $\lambda_1(\rho), \lambda_2(\rho), \cdots, \lambda_N(\rho)$ are now chosen so that

$$\frac{\lambda_j(\rho)}{\lambda(\rho) - \sum_{i \neq j} \lambda_i(\rho)\overline{T}_i(\rho)/\overline{T}(\rho)} < Q(\rho) + 1 - \rho$$

and

$$j = \left\{ j \colon \max_i \overline{T}_i = \overline{T}_j \right\}.$$

Then

$$\frac{\lambda_j(\rho)}{\lambda(\rho)} < \frac{\lambda_j(\rho)}{\lambda(\rho) - \sum_{i \neq j} \lambda_i(\rho)\overline{T}_i/\overline{T}_i} < Q(\rho) + 1 - \rho$$

so

$$\lim_{\rho \to 1} \frac{\lambda_j(\rho)}{\lambda(\rho)} \leq \frac{g}{N}$$

or

$$\lim_{\rho \to 1} \lambda_j(\rho) \leq \frac{gK\mu}{N}.$$

But the latter result would imply that the queue with the largest delay is stable, and this is a contradiction. Thus, $Q = \lim_{\rho \to 1} Q(\rho) = 1/N$.

5. Equivalence of the Queues $GI/D/c$ and $GI_c/D/1$

Consider a $GI/D/c$ queue with the discipline that the ith arriving job is assigned to a separate queue for the $i(\text{mod } c)$th server. Because of the deterministic service times, this algorithm will always assign a job to a queue with the least backlog of work. It follows that jobs see a system equivalent to $GI/D/c$ FCFS. Thus, we have that the $GI/D/c$ FCFS queue has a sojourn time equivalent to that of a $GI_c/D/1$ FCFS queue where GI_c denotes the renewal stream with interarrival time given by the sum of c interarrival times from the renewal stream GI. Similar observations have been made in [24], [48]. In particular, it has been noted in [24] that the waiting time distribution of an $M/D/c$ system is the same as that of an $E_c/D/1$ system.

6. Analysis of Q-Factors for Some ∞-Server Cases

First note that for $M/D/K$ systems, the queue length process (i.e., the number of jobs in the system) is independent of the queueing discipline provided that every server selects a job for processing independently of the job's service requirement and serves it to completion. For these disciplines and a given ρ, as $K \to \infty$ the number of busy servers is distributed as

$$\frac{\rho^i}{i!} e^{-\rho}.$$

In particular, this is true for FCFS as well as cyclic service ($B = 1$) and SLQ. Thus, with probability 1, the number of idle servers is greater than 0 and consequently jobs do not wait. It then follows that

$$Q_{\text{SLQ}}(\rho) = Q_{\text{cyclic service } (B=1)}(\rho) = 1.$$

7. Mean Waiting Time in a $GI^{[X]}/G/1$ Queue

Let the batch interarrival time have distribution $A(\cdot)$. Let the batch size $X = n$, with probability p_n, $n = 1, 2 \cdots$. Let the service time distribution of each job be $S(\cdot)$. Then the mean waiting time of a tagged job can be written as

$$\overline{W} = \overline{W}_{\text{BATCH}} + \overline{W}_{\text{FB}}$$

where $\overline{W}_{\text{BATCH}}$ is the mean waiting time of the first job in a batch, and $\overline{W}_{\text{FB}}$ is the total mean service times of the jobs in front of the tagged job in the same batch.

$\overline{W}_{\text{BATCH}}$ can easily be obtained by considering the batch as one single job (whose service time distribution has a Laplace–Stieltjes transform $\sum_{n=1}^{\infty} p_n \hat{S}^n$) and solving the corresponding $GI/G/1$ queue. Since the mean number of jobs in front of a randomly chosen job in a batch is given by $(E[X^2]/E[X] - 1)/2$, we have

$$\overline{W}_{\text{FB}} = \frac{1}{2}\left(\frac{E[X^2]}{E[X]} - 1\right)\mu^{-1}$$

where μ^{-1} is the mean service time of a job.

REFERENCES

[1] G. R. Andrews, D. P. Dobkin, and P. J. Downey, "Distributed allocation with pools of servers," in *Proc. ACM SIGACT-SIGOPS Symp. Princ. Distrib. Comput.*, 1982, pp. 73–83.

[2] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *IEEE Computer*, vol. 11, pp. 97–106, July 1978.

[3] W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing," *IEEE Computer*, vol. 13, pp. 57–69, Nov. 1980.

[4] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 401–412, July 1982.

[5] L. M. Ni and K. Abani, "Nonpreemptive load balancing in a class of local area networks," in *Proc. Comput. Networking Symp.*, Dec. 1981.

[6] Y. C. Chow and W. H. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. Comput.*, vol. C-28, pp. 354–361, May 1979.

[7] T. Yum, "The design and analysis of a semidynamic deterministic routing rule," *IEEE Trans. Commun.*, vol. COM-29, pp. 498–504, Apr. 1981.

[8] A. K. Agrawala and S. K. Tripathi, "On the optimality of semidynamic routing schemes," *Inform. Processing Lett.*, vol. 13, no. 1, pp. 20–22, Oct. 1981.

[9] L. M. Ni and K. Hwang, "Optimal load balancing for a multiple processor system," in *Proc. Int. Conf. Parallel Processing*, 1981, pp. 352–357.

[10] R. M. Bryant and R. A. Finkel, "A stable distributed scheduling algorithm," in *Proc. 2nd Int. Conf. Distrib. Comput. Syst.*, 1981.

[11] Y. C. Chow and W. H. Kohler, "Dynamic load balancing in homogeneous two processor distributed systems," in *Computer Performance*, K. M. Chandy and M. Reiser, Eds. Amsterdam, The Netherlands: North-Holland, 1977.

[12] S. W. Furhman, "Performance analysis of a class of cyclic schedules," unpublished.

[13] R. B. Cooper, *Introduction to Queueing Theory, 2nd ed.* London: E. Arnold, 1981.

[14] M. Eisenberg, "Two queues with alternating service," *SIAM J. Math*, vol. 36, pp. 287–303, 1979.

[15] O. Hashida, "Gating multiqueues served in a cyclic order," *Systems-Computers-Controls*, vol. 1, no. 1, pp. 1–8, 1970.

[16] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*. New York: Addison-Wesley, 1967.

[17] C. D. Crommelin, "Delay probability formula when the holding times are constant," *P. O. Elec. Eng. J.*, vol. 25, pp. 41–50, 1932.

[18] J. H. A. de Smit, "The queue $GI/M/S$ with customers of different types or the queue $GI/H_m/S$," *Adv. Appl. Prob.*, vol. 15, pp. 392–419, 1983.

[19] A. E. Eckberg, Jr., "Response time analysis for pipelining jobs in a tree network of processors," *Applied Probability—Computer Science, The Interface*. Birkhauser, 1982.

[20] D. Gross and C. M. Harris, *Fundamentals of Queueing Theory*. New York: Wiley, 1974.

[21] S. Albin, "Approximating superposition arrival processes of queues," unpublished.

[22] M. I. Reiman and B. Simon, "Queues in light traffic," to be published.

[23] G. J. Foschini, "On heavy traffic diffusion analysis and dynamic routing in packet switched networks," *Computer Performance*, K. M. Chandy and M. Reiser, Eds. Amsterdam, The Netherlands: North-Holland, 1977.

[24] N. U. Prabhu, *Queues and Inventories*. New York: Wiley, 1965.

[25] R. J. T. Morris and Y. T. Wang, "Some results for multi-queue systems with multiple cyclic servers," in *Proc. 2nd Symp. Perform. Comput. Commun. Syst.*, Zurich, Switzerland, Mar. 21–23, 1984.

[26] G. Le Lann, "A distributed system for real-time transaction processing," *IEEE Computer*, vol. 14, pp. 43–48, Feb. 1981.

[27] L. M. Casey, "Decentralized scheduling," *Australian Comput. J.*, vol. 13, pp. 58–63, May 1981.

[28] L. Kleinrock, *Queueing Systems, Vol. 2: Computer Applications*. New York: Wiley, 1976.

[29] H. Aiso, Y. Matsushita, *et al.*, "A minicomputer complex-KOCOS," in *Proc. IEEE/ACM 4th Data Commun. Symp.*, Oct. 1975.

[30] R. G. Kieburtz, "A distributed operating system for the Stony Brook Multicomputer," in *Proc. 2nd Int. Conf. Distrib. Comput. Syst.*, Apr. 1981.

[31] L. D. Wittie and A. M. van Tilborg, "MICROS, A distributed operating system for MICRONET, A reconfigurable network computer," *IEEE Trans. Comput.*, vol. C-29, pp. 1133–1144, Dec. 1980.

[32] A. M. van Tilborg and L. D. Wittie, "Distributed task force scheduling in multi-microcomputer networks," in *Proc. AFIPS*, 1981.

[33] B. Miller and D. Presotto, "XOS: An operating system for the X-tree architecture," *Oper. Syst. Rev.*, vol. 15, pp. 21–32, 1981.

[34] J. P. Cabanel, M. N. Marouane, R. Besbes, R. D. Sazbon, and A. K. Diarra, "A decentralized OS model for ARAMIS distributed computer system," in *Proc. 1st Int. Conf. Distrib. Comput. Syst.*, Oct. 1979.

[35] R. H. Halstead and S. A. Ward, "MuNet: A scalable decentralized architecture for parallel computation," in *Proc. IEEE 7th Annu. Symp. Comput. Arch.*, 1980.

[36] H. Sullivan, T. R. Bashkow, and D. Klappholz, "A large scale homogeneous, fully distributed parallel machine II," in *Proc. 4th Annu. IEEE Symp. Comput. Arch.*, 1977.

[37] D. C. Farber and K. C. Larson, "The distributed computer system," in *Proc. Symp. Comput. Commun. Networks and Teletraffic*, 1972.

[38] R. G. Smith, "The contract net protocol: High level communication and control in a distributed problem solver," *IEEE Trans. Comput.*, vol. C-29, pp. 1104–1113, 1980.

[39] R. Peebles and T. Dopirak, "ADAPT: A query system," in *Proc. COMPCON*, Spring 1980.

[40] M. H. Solomon and R. A. Finkel, "The ROSCOE distributed operating system," in *Proc. 7th Symp. Oper. Syst. Princ.*, 1980.

[41] R. H. Howell, "The integrated computer network system," in *Proc. 1st Int. Conf. Comput. Commun.*, 1972.

[42] F. C. Colon, R. M. Glorioso, W. H. Kohler, and D. C. Li, "Coupling small computers for performance enhancement," in *Proc. AFIPS*, 1976.

[43] K. Hwang, W. J. Croft, G. H. Goble, B. W. Wah, F. A. Briggs, W. R. Simmons, and C. L. Coates, "A UNIX-based local computer network with load balancing," *IEEE Computer*, vol. 15, pp. 55–66, Apr. 1982.

[44] S. Majumdar and M. L. Green, "A distributed real time resource manager," in *Proc. IEEE Symp. Distrib. Data Acquisition, Comput. Control*, 1980.

[45] J. A. Stankovic, "The analysis of a decentralized control algorithm for job scheduling utilizing Bayesian decision theory," in *Proc. IEEE Int. Conf. Parallel Processing*, 1981.

[46] R. Hokstad, "The steady-state solution of $M/K_2/m$ queue," *Adv. Appl. Prob.*, vol. 12, pp. 799–823, 1980.

[47] M. F. Neuts, *Matrix-Geometric Solutions in Stochastic Models*. The Johns Hopkins Univ. Press, 1981.

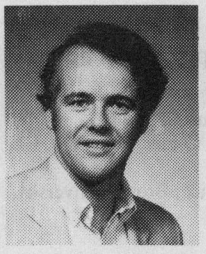
[48] V. B. Iversen, "Decomposition of an $M/D/r.k$ queue with FIFO into k $E_k/D/r$ queues with FIFO," *Oper. Res. Lett.*, vol. 2, no. 1, pp. 20–21, Apr. 1983.

[49] H. L. Applewhite *et al.*, "Decentralized resource management in distributed computer systems," Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., Feb. 1982.

[50] W. Whitt, "The queueing network analyzer," *Bell Syst. Tech. J.*, vol. 62, no. 9, pp. 2779–2815, Nov. 1983.

Yung-Terng Wang (S'76–M'79) was born in Taiwan, China, on December 27, 1949. He received the B.S. degree in electrical engineering from National Taiwan University in 1972 and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1977 and 1978, respectively.

From 1972 to 1974, he served in the Chinese Army. In 1979, before he joined the AT&T Bell Laboratories, he was a Lecturer in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. He is currently a Supervisor of the Department of Performance Analysis, AT&T Bell Laboratories, Holmdel, NJ. His current interests are in the areas of modeling, analysis, and engineering of computer and communications systems.



Robert J. T. Morris (S'75–M'78) received the combined B.Sc.B.E. degree in computer science, mathematics, and electrical engineering from the University of New South Wales, Sydney, Australia, in 1974, and the M.S. degree in system science and the Ph.D. degree in computer science from the University of California, Los Angeles, in 1976 and 1978, respectively.

Since 1978, he has been with AT&T Bell Laboratories, Holmdel, NJ, working on a variety of computer and communication system projects. He is currently Supervisor of the Performance Analysis Studies Group.