ADAPTIVE LOAD SHARING IN HETERGENEOUS SYSTEMS: POLICIES, MODIFICATIONS, AND SIMULATION

KARIM Y. KABALAN*, WALEED W. SMARI+ and JACQUES Y. HAKIMIAN*

* *Department of Electrical and Computer Engineering, American University of Beirut, P.O. Box: 11-0236, Beirut, Lebanon*

+ *Department of Electrical and Computer Engineering, University of Dayton, 300 College Park, Dayton, OH 45469, USA*

Abstract: Distributed computing systems offer the potential for improved performance and resource sharing. In a real-time system, uncontrolled task arrivals may temporarily overload some nodes while leaving other nodes idle. Adaptive load sharing offers techniques for queue control in order to achieve optimal or near-optimal efficiency and performance. This paper discusses several adaptive load sharing algorithms for heterogeneous distributed computing systems, proposes some modifications to existing algorithms that will account for the delay in transferring tasks from one node to another, and verifies and validates those proposed changes with some of the simulation results obtained.

Keywords: Load sharing and balancing, Adaptive policies, Dynamic Load Balancing algorithms, Load distribution, DCS.

1. INTRODUCTION

Parallel and distributed computing systems present high-performance environments that are capable of providing immense processing capacity [Lewis and El-Rewini, 1992]. In order to realize these capabilities, efficient resource allocation algorithms and load distribution schemes (load balancing or sharing) must be employed. The objective would be to maximize the overall performance of the system using an optimality criterion [Xu and Lau, 1997]. It would be inaccurate to say that the computing power of a distributed system increases proportionally with the number of processors involved. This is often not the case. Care should be taken so that some processors do not become overloaded and some others stay idle. Also, task migration (shifting of a job from one node to another for optimization purposes) should be controlled in a way that the transfer cost incurred is worth the enhancement in computational efficiency.

The issue of load distribution emerged when distributed computing systems and multiprocessing techniques began to gain popularity. Over the past few decades, various load distribution algorithms have been proposed and implemented. These techniques have been classified and characterized in a number of studies [Baumgartner and Wah, 1991; Casavant and Kuhl, 1994; Lewis and El-Rewini, 1992; Xu and Lau, 1997]. One such classification characterizes load distribution schemes as static and dynamic, while another taxonomy considers those techniques as adaptive vs. nonadaptive. As such, let us consider these three categories of load distribution algorithms.

In a static load distribution scheme, the assignment of processes (or tasks) to processors is done at compilation time before processes are executed, usually using a priori knowledge about the tasks and system on which they run. A main advantage of these techniques is that they will not introduce any run-time overhead. Another advantage behind static algorithms is that they are simple to construct and can be quite efficient in homogeneous distributed systems. However, their performance depends on projected execution times and interprocess communication requirements, and hence, they may not be suitable for dynamic and/or unpredictable environments and applications. In general, these problems are NP-complete, even under simple assumptions. Therefore, most of the methods employed use mathematical approaches to obtain sub-optimal solutions [Efe, 1982; Harget and Johnson, 1990; Lo, 1988; Sofianopoulou, 1992].

In dynamic load distribution, no decision is made until processes start executing in the system. The scheme uses processes and system state information when making the load distribution decisions. Dynamic algorithms are especially critical in applications where parameters that affect the scheme cannot be easily determined a priori or when the workload evolves as computation progresses. These load distribution algorithms incur run-time overhead. The overhead is due mainly to requirements to maintain a consistent view of the system state at run-time, and some negotiation scheme for process migration across system's nodes. Key issues to consider when designing a dynamic load distribution algorithm are: a load assessment rule, an information exchange policy, an initiation scheme, and a load distribution operation that includes a task selection policy, a node selection policy, a transfer (or migration) policy and a profitability policy [Corradi, 1999; Hu and Blake, 1999; Karatza, 2001; Touheed et.al., 2000; Xu and Hwang, 1993; Xu and Lau, 1997; Zhou, 1988].

Static and dynamic algorithms can be extremely efficient in homogeneous systems. If all machines are identical, then the choice "to which node should we assign the next task?" falls on the machine with the fewest outstanding jobs. However, in heterogeneous distributed environments, where not all nodes have the same parameters, selecting the optimal target machine requires more elaborate algorithms. Dynamic algorithms are employed more frequently because they offer a better efficiency, having the flexibility of job migration. In addition, usually, dynamic algorithms are neither very complicated to construct nor costly to run. Hence, dynamic algorithms have the potential to outperform static algorithms in general. The question, then, arises as to why should we go for adaptive policies?

An adaptive load distribution algorithm is a dynamic policy that is modifiable: as the system state changes, it, in turn, adjusts its activities based on the feedback it receives from the system. Typically, an adaptive algorithm consists of a collection of several load distribution schemes, which are chosen based on various system parameters [Lewis and El-Rewini, 1992].

Several adaptive load distribution (balancing or sharing) policies have been developed and used in recent years [Aweya et al, 2002; Corradi et al, 1999; Das et al, 2002; Hui and Chanson, 1999; Karatza, 2001; Liao and Chung, 1999; Touheed et al, 2000; Xu and Lau, 1997]. However, interest in adaptive approaches goes back, at least, two decades [Bokhari, 1981; Chou and Abraham, 1982; Chow and Kohler, 1977; Chow and Kohler, 1982; Eager et al, 1986; Kremien and Kramer, 1992; Krueger and Livny, 1988; Leff and Yu, 1991; Lin and Raghavendra, 1991; Xu and Hwang, 1993; Zhou, 1988]. The need for these techniques arises from the rapid growth of computer networks and of the complexity of the tasks required from them. Nowadays, many users do not rely on a single workstation with a single processor to serve their computational needs. The emergence of distributed systems and multiprocessor machines allows computer users to make use of the computing power of several workstations, each having several processors, to execute their computing tasks. The complexity and nature of the applications used represent another dimension in current and projected trends in computing environments. The increasingly dynamic and intelligent systems that are being or will be developed will add to the demand for adaptable policies.

This paper explores several adaptive load distribution algorithms for heterogeneous distributed computing systems. It then proposes and illustrates some modifications to these algorithms to improve the sharing of the load between the nodes of a network. Lastly, it presents some of the simulation results to observe the operation performance on heterogeneous distributed networks under varying load and either random or specific network topology. We then make some concluding remarks.

2. ADAPTIVE LOAD SHARING IN HETEROGENEOUS SYSTEMS

Real systems are often more likely to be heterogeneous than homogeneous [Hou and Shin, 1994; Shin and Hou, 1993]. Choosing the optimal target machine among a set of servers with different service rates is still an unsolved problem. In this paper, we choose to study the operation of adaptive load distribution algorithms while considering the effect of the tasks transfer delay. We will also discuss a simple model of computation with N exponential servers, each having a (perhaps different) service rate μ_k with $k \in [1,N]$. Each server has its own local queue and the number of jobs in each queue is denoted by x_k. Should we consider that job transfer is free and instantaneous, we would consolidate the system-wide arrival of jobs into a single Poisson stream of strength λ. However, since the communication cost is considered to be significant, the arrival stream at each node is $\lambda+\delta_k$, where λ is a constant rate at which tasks are initiated at any node and δ_k is the rate of transfer of jobs to node k. The problem reduces now to that of "joining the right queue." This is depicted in Figure 1.

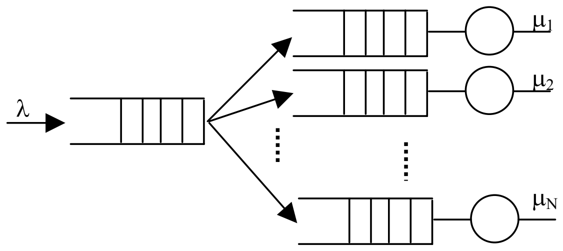


Figure 1. A Simplified Model of the System under Consideration

The following assumptions about the system that we will be considering are made:

- The architecture of the individual nodes includes a powerful bus interface unit (BIU), which is used to process most of the overhead generated by task or probe movement.
- The BIU has a direct memory access (DMA) capability to access memory without much interference to the CPU.

While the bulk of the processing overhead for task migration is transferred to the BIU, delays will nevertheless occur during this processing. There will also be network delays in the transmission of probes and tasks. The relative sizes of probes and tasks are generally quite different. While the physical transfer of a task may require moving tens of communication packets from a node to another, a probe or a response to a probe would in all likelihood need at most one packet. Consequently, it is quite safe to assume that the delays incurred by tasks in the BIU and the network will be significantly larger than those incurred by the probes. In our analysis, the delays incurred by probes will be assumed to be negligible when compared to those incurred by task transfers.

Next, we discuss three adaptive load sharing and distribution policies that have been proposed in the literature and present corresponding modifications we are proposing to each.

2.1. The Shortest Expected Delay (SED) Policy [Weinrib and Shenker, 1988]

This policy attempts to minimize each job's expected delay (until completion). The expected delay that the job will experience if sent to the kth server is $d_k = (1+x_k)/\mu_k$. The selected destination node will naturally be the one for which d_k is minimal. It is a greedy policy in that each job does what is in its own immediate best interest, and that is to join the queue which would minimize its expected delay of completion.

In fact, this policy optimally minimizes the average delay of a given batch of jobs with no subsequent arrival. However, with an ongoing arrival process, the SED policy no longer minimizes the average delay. Scheduling decisions affect not only the job being scheduled but will also potentially affect the subsequently arriving jobs, which may be forced to wait in the queue longer or use a slower server as a result of previous scheduling decisions.

This definition of the policy requires the source node to get state information from other nodes in order to make its location decision. A node may either choose to keep some static information regarding other nodes in the system in a state vector or it may choose to request information when the need arises. It may also use predictive analysis techniques to update the contents of the state vector. In any of these cases the information held or requested may pertain to the whole system or to a subset of the system. For scalability purposes, subsets are more often used, and certain criteria may be devised to specify the nodes to be included in the subset. Nodes may choose to disseminate their state information periodically or each time this information changes.

A slightly different algorithm based on minimizing the shortest expected delay is the DISTED algorithm (DISTED = distributed) introduced by Zhou [Zhou, 1988]. In this scheme, each node periodically broadcasts its local load to all other nodes, unless it has not changed since the last update. A data structure at each node holds load state information regarding all nodes in the system. Zhou's algorithm is not scalable, since state information is held regarding all other nodes, which makes it dependent on system size [Kremien and Kramer, 1992]. Also, if dissemination costs are high, periodic state updates may result in a high percent of wasted load exchanges or in outdated information if the update rate is too slow. However, with event-driven state update, a load message is sent only after a significant change has occurred.

After defining the SED policy, we still have to revise our transfer policy. Should we introduce a threshold T to trigger the location policy? Clearly, if we decide to choose a fixed threshold T below which each node chooses to locally process all jobs that are initiated at that node, this threshold may be too low for fast nodes or too high for slow nodes which would be saturated or overloaded before trying to resort to the SED policy to alleviate its load. If we decide, on the other hand, that each node k should have a certain threshold T_k on its queue length, beyond which any arriving task would search for another destination, this will clearly result in a loss of optimality of the algorithm. The optimality loss is since not all the jobs initiating at a node would try to minimize their expected delay.

The algorithms introduced in this paper make use of a threshold of zero in their transfer policies so that the location policy is triggered when any job initiates at any node.

2.2. Modifications to the SED Policy

A term representing the communication delay needs to be added to the expression of the expected delay that a job experiences if it joins the queue to the jth server. Thus, $d_j = [(1+x_j)/\mu_j] + t_{com}(i,j)$, where the second term is the mean communication delay experienced by a job sent from a source node i to a destination node j. If the decision made is to process the job locally, then $t_{com}(i,i) = 0$.

Since the queue size of each node is finite, the second modification made is to consider the scenario whereby, if the job is sent to a node whose queue is close to saturation, the job will not be able to enter the queue. The action to take in this case is to choose the "second best" node that has the second lowest expected delay. If the second node is also close to saturation, then the "third best" is chosen, and so on. If none of the nodes probed can qualify, then the job is sent anyway to the first node that was chosen. Upon its arrival, it will probably be unable to enter the queue and it will be stored temporarily at this node and attempt to enter the queue again after a fixed amount of time T_{ret}. Intuitively, T_{ret} should be higher than the average of all mean service times (possibly twice as high).

Normally, a queue should be considered to be saturated if its length is equal to q_{max}, and a job sent to a queue whose length is q_{max}, would most probably be unable to enter the queue (unless a job currently in service at the destination node completes before the current job reaches its destination.) However, even if the queue length is less than q_{max}, the corresponding node could be expecting other jobs that are being transferred to it and that would enter the queue before the current job reaches the queue. Thus, the current job would still be unable to enter. Under the assumption that the transfer time of jobs is relatively small compared to the inter- arrival time (inter- initiation time) of jobs, we have considered empirically that a node could be considered to be saturated when its queue length is equal to q_{max} - 1. The evaluation and simulation results of these modifications to the SED policy will be presented and discussed in Sections 3 - 5.

2.3. The Never Queue (NQ) Policy [Weinrib and Shenker, 1988]

Dynamic load distribution policies may be classified as *separable* or *nonseparable* policies. In separable policies, the sending server computes a cost or toll of sending a job to each eventual destination, or a subset of eventual destinations, and the job is placed on the server with the minimal cost. On the other hand, non-separable policies relax the individual cost constraint, allowing a decision to depend on the full set of state information. The Never Queue (NQ) policy [Weinrib and Shenker, 1988] is a separable policy in the sense described above. With this policy, a job is always sent to the fastest available server. This policy may appear at first ill advised, in that jobs are scheduled in a manner that increases their individual expected delays. However, the NQ policy minimizes the extra delay caused to the subsequently arriving jobs, so that the effect on the overall average delay may be reasonable. Furthermore, the NQ policy imposes a threshold T = 0 on the transfer policy. Hence, an incoming job at a server is always sent to another destination unless the source itself is the fastest available server.

The NQ policy [also Shenker and Weinrib, 1989] takes into consideration that the system behaved as a two-speed model with a finite number of fast servers and an infinite number of slow servers. Since the number of servers in that model is infinite, an idle server can always be found. The motivation behind the NQ policy is that at small loads, queuing for the fast servers minimizes the individual job's delay. Since the load is light, it is unlikely that the job will cause delays for future jobs. This is why the SED policy may yield near-optimal results. However, at high loads (high arrival rate λ), queuing for fast servers is inadvisable since the slower servers must be used in order to provide sufficient total processing power. Soon after a fast server becomes idle, a new job will arrive quickly and request service on it (the time elapsed will be roughly $1/\lambda$). In this case, as long as all the fast servers are utilized, it is better to immediately serve a newly arrived job at an idle slow server than to wait for a fast one.

However, the two-speed model is not always applicable since the total number of servers, N, is a finite number, and an idle server may not always be available. Thus, we have modified this algorithm to account for such a case where all servers are busy. In the latter situation, a greedy decision will be made whereby the job will be sent to the server that minimizes its individual delay (that is, the one with the smallest $(1+x_k)/\mu_k$). Note also that when using the NQ policy, the probe limit should be set high enough such that the probability of missing an idle server (because it was not probed for its state information) is made low.

2.4. Modifications to the NQ Policy

The Never Queue policy introduced above attempts to find the fastest idle server among the nodes that are probed, and sends the job to that server. If none of the servers is idle, the server with the shortest expected delay is selected. Again, as stated in the previous paragraph, if that server has a queue that is close to saturation, the server with the "second shortest" expected delay is selected, and so on. If no server, among those probed initially is eligible to receive a job, then that job is sent anyway to the server that would provide the shortest expected delay. Upon its arrival, that job will probably be unable to enter the queue and hence will be stored temporarily at this node and attempt to enter the queue again after a fixed amount of time T_{ret}.

To account for the communication delay between nodes, we will modify the policy as follows:

1. Find among the subset S_i of the nodes probed by the source node i, the nodes that are idle.
2. For each idle node j, the expected execution time of a job sent from i to be processed at j will be: $E[t_{exec}] = t_{com}(i,j) + (1/\mu_j)$, since the communication delay between i and j, $t_{com}(i,j)$, and the mean service time at j are uncorrelated. The node chosen will be the one with the least $E[t_{exec}]$.
3. If there are no idle nodes, then the node chosen will be the node j with the least $d_j = [(1+x_j)/\mu_j] + t_{com}(i,j)$. If the queue length of j is greater or equal to q_{max}-1, then the node chosen will be the one with the second smallest d_j, and so on.
4. If none of the nodes have a queue length less than q_{max}-1, then the job is sent anyway to the node with the smallest d_j.

The evaluation of this modified policy and its simulation results will be presented and discussed in Sections 3 - 5.

2.5. The Greedy Throughput (GT) Policy [Shenker and Weinrib, 1989]

The SED policy minimizes the expected delay of each arriving job, while the NQ policy can be seen as maximizing the instantaneous throughput rate, which is the total service rate of all occupied servers immediately following a scheduling decision. An alternative throughput optimization criterion is to maximize the expected number of job completions before the next job arrival, rather than to maximize the throughput rate only at the instant of scheduling. This policy will be called the Greedy Throughput (GT) policy.

When a job initiates at a node i, this node probes a subset of nodes by choosing them either randomly or by assigning to each node a "buddy set" of nodes to probe at each time. The information requested from the nodes probed includes the processing rate of the node and its queue length. The general formula for the implementation of this policy has been stated in [Weinrib and Shenker, 1988], and it consists of maximizing the quantity $[\mu_i / (\lambda + \mu_i)]^{1+n_i}$, where n_i is the total number of jobs in node i's queue, including any job in service.

2.6. Modifications to the GT Policy

The greedy throughput policy calls for maximizing the throughput of the system defined as the expected number of job completions before the next job arrival. This was found to be equivalent to maximizing the quantity $T(j) = [\mu_j / (\lambda + \mu_j)]^{1+nj}$ where n_j is the total number of jobs in the j^{th} queue including any job in service. For each node j, the ratio $\mu_j / (\lambda + \mu_j)$ is less than 1. Thus, the greater n_j is, the smaller T(j) will be, and the less "recommendable" (exponentially) the node j becomes to be selected as a destination node.

Since the decision criterion is dependent upon λ, the job initiation rate, the latter should be calculated with accuracy. The job initiation rate is not constant since jobs that cannot join a queue have to be stored and then re-initiated after a certain time interval T_{ret}. Thus, an average for λ has to be calculated empirically, as each job is initiated or re-initiated and its latest value should be broadcasted over the network, with perhaps a time-stamp packet or a counter indicating the number of the last job initiated. To calculate λ, we use the relationship

$$<\lambda>_{new} = (1-\alpha) <\lambda>_{old} + \alpha <\lambda>_{avg}$$

where the brackets represent an exponentially weighted average over past jobs for the estimated quantities. A server maintains only a single number to keep the average, updating it when each job completes. This occurs as a fixed size batch of jobs is initiated or re-initiated. The quantity $<\lambda>_{avg}$ is the average initiation rate over a fixed size batch of jobs (chosen in our simulations to be 150 jobs). The choice of α is a tradeoff between obtaining better averages and quickly responding to changes in load. In our analysis, we will adopt $\alpha = 0.05$.

The second modification to be made to the algorithm is in the calculation of the exponent in the formula $T(j)= [\mu_j / (\lambda + \mu_j)]^{1+nj}$, where $1+n_j$ should represent accurately the number of jobs in service and in queue at node j, before the next arrival. Intuitively, $1+n_j = 1+ s_j + x_j + t_j$, where x_j is the number of nodes in queue, and t_j is the number of jobs being transferred to node j, that are expected to arrive before the next job initiation in the system. Since the communication delay incurred by a job that is transferred to any node is assumed to be much smaller than the inter-arrival time of jobs ($1/\lambda$), all jobs being transferred to node j are expected to arrive to node j before the next job initiation. Thus, t_j is simply the exact number of jobs being transferred to node j at the instant that the decision is being made. In order for this information to be available to node j, we assume that as any job is transferred from a node to another, the source node informs the destination node of the incoming job by sending a small packet as a notice before sending the job.

As for s_j, it is equal to zero if node j is idle. If node j is busy, then s_j is equal to 1 if the job currently in service at node j is not expected to be complete before the next job initiation, i.e., if $(1/\mu_j)$ - (time already spent in service) $\geq (1/\lambda)$. Otherwise, s_j is equal to zero.

Again, as in previous paragraphs, if the server with the maximum T(j) has a queue that is close to saturation, the server with the "second highest" T(j) is selected, and so on. If no server, among those probed initially is eligible to receive a job, then that job is sent anyway to the server that would provide the shortest expected delay. Upon its arrival, that job will probably be unable to enter the queue and it will be stored temporarily at this node and attempt to enter the queue again after a fixed amount of time T_{ret}.

The evaluation and simulation results of these modifications to the GT policy will be presented and discussed in the following three sections.

3. CRITERIA FOR ALGORITHMS' PERFORMANCE EVALUATION AND COMPARISON

The performance analysis of the algorithms described in Section 2 is based on the queue length of the network's nodes. However, this can be accurate only if all tasks have identical, or identically distributed execution times, and the mean response time is used as the performance metric.

It must be noted here that if task execution times are neither identical nor identically distributed, the queue length (QL) can no longer be an adequate measure to characterize the load of a node. An alternative measure would be the cumulative execution time at each node (CET), and it would be used to determine whether a node can guarantee a task or not [Shin and Hou, 1993].

In our study of the SED, NQ and GT policies, the tasks at each node do not have identical execution times, since the network is heterogeneous in nature, but the task sizes are identical and the execution times are all assumed to be identically distributed, having an exponential distribution.

The main performance metric that we will use in evaluating the performance of the algorithms is the mean response time. We will also compute the system utilization measures n_{idle} and n_{busy} to compare the utilization of the system under a certain load using a specific algorithm. The parameter n_{busy}

is the time-averaged number of busy nodes in the system. A busy node is defined as a node having one job in service and zero or more jobs in queue. The metric $n_{congested}$ is also computed. Since every node has a different service rate, all nodes cannot be considered as congested if their queue length exceeds a fixed threshold. We will assume that the fastest server, having a service rate μ_{max}, is congested when 80% of its job queue is full. Since all nodes will be assumed to have the same maximum queue length q_{max}, we will introduce the threshold $th_{congest}$ as: $th_{congest} = 0.8\ q_{max}$. Similarly, every node i in the system will have a different threshold, th_i, on its queue length beyond which it is considered as congested. This threshold will be assumed to be inversely proportional to the service rate of the node.

$$th_i = \lceil th_{congest} / (\mu_{max} / \mu_i) \rceil$$

Note here that this threshold is not used for decision making in the algorithms. It is merely used to compute $n_{congested}$. Also, in what follows, we will consider that the congested nodes are a subset of the set of busy nodes. Thus,

$$n_{idle} + n_{busy} = 1$$

The metric Φ, which is a measure of the absolute deviation from the average queue length, will not be computed in the following sections as is usually the case in similar studies, since it only applies to homogeneous systems, where the queue lengths of the nodes in the system would gain to be identical or almost identical for adequate load balancing.

4. THE SIMULATION MODEL

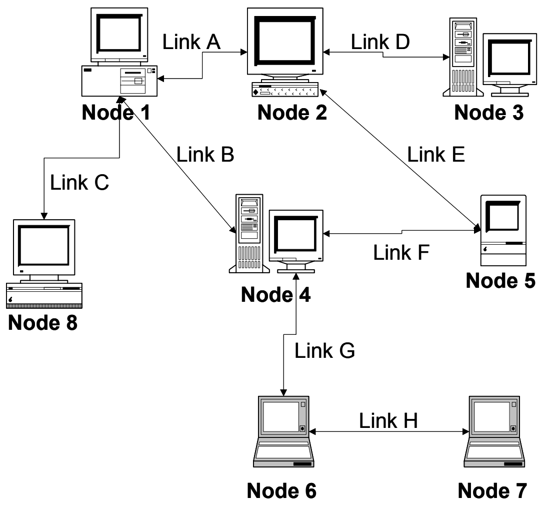
Three simulation programs were implemented to simulate the operation of the modified SED, NQ and GT policies in a heterogeneous network. Several assumptions were devised for the simulation model. These are:

1. The system consists of N nodes where each node has one server and a fixed size queue.
2. All jobs initiated in the system have a fixed size.
3. Jobs are initiated at an initiation rate of λ. The time intervals between job initiations are exponentially distributed.
4. A job is initiated at a node i with a fixed probability p = 1/N.
5. All nodes do not have the same service rate. For $i \neq j$, μ_i may be different from μ_j. The service times of jobs at every node i are all exponentially distributed with mean service time of $1/\mu_i$.
6. The network topology is specific to the case under study. Nodes are linked to each other using (perhaps) different communication media and every link i has a weight w_i. All communication links are full duplex. The communication delay incurred when transferring a node depends on the path taken by the job but we assume that in the initial stage, each node has a routing table indicating the shortest path from it to every other node. This shortest path is the one taken when transferring a job from a node to another. Moreover, the communication delay is assumed to be relatively small compared to other system time parameters like the mean inter- arrival time $1/\lambda$ and the mean service times $1/\mu_i$.
7. If a job attempts to join a queue that is saturated, it is stored locally at the node (perhaps in a swap-space on the node's hard drive) and temporarily before it tries to join the queue again after a fixed interval of time T_{ret}.
8. As each job is initiated at a node i, this node executes the corresponding location algorithms to find a destination for the job.
9. In order to make a decision on a potential destination, a node i probes at fixed number L_p of nodes chosen at random. Every node in this subset of nodes has to have a valid path from it to the source node. If the source node is unable to find L_p nodes that have a path to it, then the number of nodes probed will be the number of nodes having a path to the source node. The number of nodes probed may be zero, in which case the destination and source nodes will be the same.
10. The number of packets exchanged during a probe being negligible (as explained in Section 5.1), the time required to probe a node is assumed to be negligible relative to the transfer time of a job. Thus, no state change is assumed to take place at a node during probing.

Bearing in mind these assumptions, the algorithms introduced in Section 2 need to be modified to reflect the actual conditions of the system under study. The three (unmodified) algorithms, SED, NQ and GT, were described and simulated in the literature [Weinrib and Shenker, 1988] for a two- speed system where there are an infinite number of nodes (a fixed number of fast servers with comparable processing power and an infinite number of slower servers) and no communication delay when transferring nodes. Also, these algorithms were devised with the assumption that all nodes in the system would be considered during decision-making, which is not the case in our system, where only a subset of the nodes is probed so as to improve the scalability of the algorithms. Moreover, the three algorithms were tested and simulated in previous researches on systems with nodes having an infinite queue length, whereas in our system, a node is saturated if its queue length reaches a fixed limit q_{max}.

5. SIMULATION ON HETEROGENEOUS DISTRIBUTED NETWORKS

The system that we will simulate is an 8-node heterogeneous system, where each node consists of one server and one FIFO queue with a maximum capacity of 25 jobs. We will assume in the first stage of the simulation that before making a decision on a destination, a source node probes up to L_p= 6 nodes, that is 75% of the total number of nodes. All jobs are assumed to have similar sizes and equal priority. Jobs are processed on a First Come First Served basis. The network has a fixed topology whereby the 8 nodes are connected to each other as shown in Figure 2.



Link	From Node	To Node	Weight
A	1	2	2
B	1	4	1
C	1	8	1
D	2	3	1
E	2	5	3
F	4	5	1
G	4	6	2
H	6	7	1

Figure 2. Heterogeneous Network with Specific Topology

From \ To	1	2	3	4	5	6	7	8
1	0	2	3	1	2	3	4	1
2	2	0	1	3	3	5	6	3
3	3	1	0	4	4	6	7	4
4	1	3	4	0	1	2	3	2
5	2	3	4	1	0	3	4	3
6	3	5	6	2	3	0	1	4
7	4	6	7	3	4	1	0	5
8	1	3	4	2	3	4	5	0

Table 1. Shortest Path Matrix for the System of Figure 2

The weight of each link is a comparative measure of the communication delay from the source of the link to its destination. We will assume that the size of the jobs in the system is such that it takes 1% of a unit of time for a job to go from a source node to a destination node through a link of weight 1. The shortest path matrix of the system depicted in Figure 2, is as shown in Table 1. All three modified algorithms (SED, NQ and GT) were simulated for 100,000 processed jobs. Thus, the simulation stops when 100,000 jobs have been processed. To eliminate the start-up transients, we will ignore the first 10,000 jobs.

Number of nodes: N = 8		
Maximum Queue Size: q_{max} = 25		
Probe Limit: Lp = 6		
Number of jobs in simulation run = 100,000 jobs		
Transients: 10% => 10,000 jobs		
Node	**Mean service time = 1/μ**	**Fastest/slowest**
1	3	
2	12	
3	1	fastest
4	1	fastest
5	10	
6	15	slowest
7	15	slowest
8	8	
μ_{avg} = 0.3469		

Table 2. Highly heterogeneous system ($\mu_{fastest}/\mu_{slowest}$) = 15

1/λ	**λ / $N\mu_{avg}$**	**t_{resp}**		
		SED policy	**NQ policy**	**GT policy**
3.00	0.12	1.1790	1.2038	1.1730
2.50	0.14	1.2129	1.2763	1.2012
2.00	0.18	1.2644	1.4017	1.2518
1.50	0.24	1.3522	1.6443	1.3404
1.00	0.36	1.5980	2.2764	1.5420
0.80	0.45	1.8446	2.7651	1.7719
0.70	0.51	2.0926	3.0423	1.9880
0.60	0.60	2.5770	3.3322	2.3843
0.50	0.72	3.8613	3.8082	3.1982
0.40	0.90	9.8399	6.2446	6.2296

Table 3. Mean Response Time of a Highly Heterogeneous System

First, we will simulate the operation of the three algorithms for a system that has a relatively large heterogeneity. The mean processing times of the 8 nodes are shown in Table 2. The highest mean processing time (the slowest server) is 15 times higher than the lowest mean processing time (the fastest server). We vary the load by varying the mean inter-arrival time (initiation time) of the jobs, $1/\lambda$. The higher the load, the higher the mean response time t_{resp} of any of the three algorithms, as depicted in Table 3. We can conclude from observing Figure 3, that the NQ algorithm behaves poorly in a highly heterogeneous system. This conclusion is due to the fact that the location policy chooses the fastest idle server, which can be much slower in a highly heterogeneous system than a very fast server that has only a small amount of jobs in queue (or that is currently processing a job and has an empty queue). Thus, the choice of the destination is not always a very successful one. The SED policy performs better than the NQ policy in such a system, except at very high load ($1/\lambda$ = 0.5 or 0.4), where all the nodes of the system are nearly always busy and the NQ policy resorts to finding the node that minimizes the expected delay of the incoming job. The GT policy performs better than the other two policies under any load.

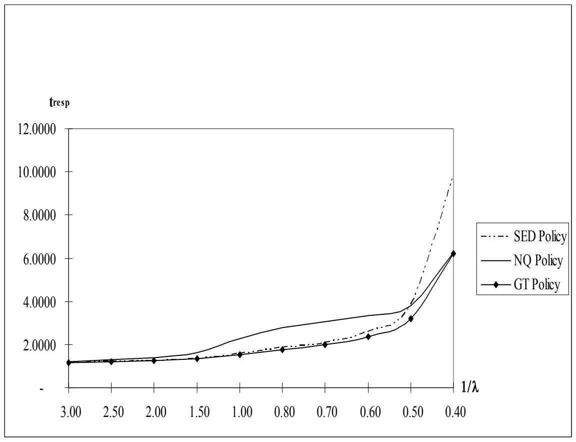


Figure 3. Mean response time of a highly heterogeneous system

We will next observe the operation of the three modified algorithms in a less heterogeneous system than previously. The mean processing times of the 8 nodes are listed in Table 4. The mean processing rate of the fastest server is 8 times higher than the mean processing rate of the slowest server. By observing the mean response time of the three algorithms as depicted in Table 5 and Figure 4, we can conclude that the NQ algorithm behaves better in this case than in a highly heterogeneous system. Under a small load ($1/\lambda$ = 2.5 or 2.0), the NQ policy performs better than both the SED and the GT policies. Under an average load, the GT policy yields lower response times than the NQ policy. As the load gets higher, the NQ policy outperforms the GT policy again. In such a system, the NQ policy is also sometimes outperformed by the SED policy. At a high load, the NQ policy also results in a busier system than the other policies. When $1/\lambda$ = 0.4 for instance, a run of simulation performed for each algorithm, shows that the time-averaged number of busy nodes when applying the NQ policy is 5.84, while it is 4.70 for the GT policy and 3.66 for the SED policy. This is an indication that, when applying the NQ policy, the nodes are perhaps more efficiently used than when the other two policies are employed. This observation results in a lower average response time as shown in Table 5.

Number of nodes: N = 8		
Maximum Queue Size: q_{max} = 25		
Probe Limit: Lp = 6		
Number of jobs in simulation run = 100,000 jobs		
Transients: 10% => 10,000 jobs		
Node	**Mean service time = $1/\mu$**	**Fastest/slowest**
1	2	
2	7	
3	1	fastest
4	1	fastest
5	5	
6	8	slowest
7	8	slowest
8	5	
μ_{avg} = 0.4116		

Table 4. Medium heterogeneity system ($\mu_{fastest}/\mu_{slowest}$) = 8

$1/\lambda$	**$\lambda / N\mu_{avg}$**	**t_{resp}**		
		SED policy	**NQ policy**	**GT policy**
2.50	0.12	1.2122	1.1498	1.1985
2.00	0.15	1.2608	1.2121	1.2414
1.50	0.20	1.3452	1.3305	1.3136
1.00	0.30	1.5333	1.6179	1.4791
0.80	0.38	1.7059	1.8740	1.6155
0.65	0.47	2.0026	2.1656	1.8433
0.50	0.61	2.7608	2.5960	2.3334
0.40	0.76	4.3442	3.2179	3.2332
0.32	0.95	10.3105	8.2349	9.2351

Table 5. Mean response time of a medium heterogeneity system

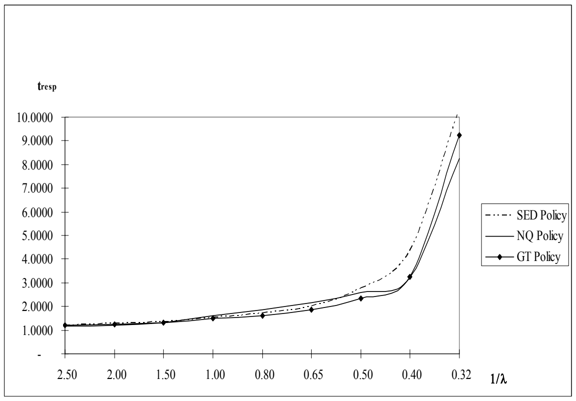


Figure 4. Mean response time of a system with medium heterogeneity

We will now focus our analysis on the case where the system is much less heterogeneous. The nodes have comparable service rates (see Table 6), and the mean processing rate of the fastest server is only 3 times as high as the mean processing rate of the slowest server. By observing the mean response time of the three modified algorithms, as shown in Table 7 and Figure 5, we can conclude that the NQ policy behaves better than in the previous two systems. The response time of the system that results from applying the NQ algorithm is lower than the response times of the other two algorithms under all loads. The GT policy still outperforms the SED policy for this system. Again, the time-averaged number of busy nodes under the NQ policy is higher than the number of busy nodes under the GT and SED policy. For instance, when $1/\lambda = 0.3$, a run of simulation performed for each algorithm indicates that the time-averaged number of busy nodes when applying the NQ policy is 4.80, while it is 4.60 for the GT policy and 4.06 for the SED policy.

Number of nodes: N = 8		
Maximum Queue Size: $q_{max} = 25$		
Probe Limit: Lp = 6		
Number of jobs in simulation run = 100,000 jobs		
Transients: 10% => 10,000 jobs		
Node	**mean service time = $1/\mu$**	**Fastest/slowest**
1	1	fastest
2	3	slowest
3	1	fastest
4	1	fastest
5	2	
6	3	slowest
7	3	slowest
8	2	
$\mu_{avg} = 0.6250$		

Table 6. Low heterogeneity system ($\mu_{fastest}/\mu_{slowest}$) = 3

$1/\lambda$	$\lambda / N\mu_{avg}$	t_{resp}		
		SED policy	**NQ policy**	**GT policy**
1.50	0.13	1.2289	1.0597	1.2070
1.00	0.20	1.3253	1.1157	1.2901
0.80	0.25	1.3978	1.1677	1.3258
0.65	0.31	1.4880	1.2339	1.3808
0.50	0.40	1.6742	1.3513	1.4039
0.40	0.50	1.9214	1.4966	1.6526
0.30	0.67	2.4538	1.8163	2.0222
0.25	0.80	3.0871	2.3339	2.5720
0.21	0.95	6.2518	5.7121	6.0921

Table 7. Mean response time of a low heterogeneity system

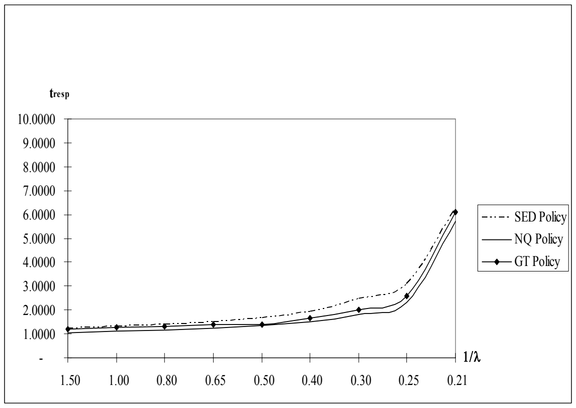


Figure 5. Mean response time of a system with low heterogeneity

Now, we examine the effect of varying the probe limit L_p on the overall response time of the system. We performed several rounds of simulation on the same system with low heterogeneity, similar to what was done in the previous paragraph, as is summarized in Table 8, and we changed the probe limit from $L_p = 6$ to $L_p = 4$ (up to 50% of the nodes in the system are probed this time). The simulation results are compiled in Table 9 and Figure 6. The response times for all algorithms under any load have all increased because the decision criterion is applied on a lesser number of nodes, which makes the decision less likely to be optimal or near-optimal. The mean response times for the NQ policy have increased in a higher proportion than the ones for the other two policies, since the decision criterion for the NQ policy is to join the fastest idle server. The less the number of nodes probed, the less likely it is for the actually fastest idle server to be among those nodes. The response times for the NQ policy are more affected at a low load when most nodes are idle and the “Never Queue” policy is applied.

Number of nodes: N = 8		
Maximum Queue Size: q_{max} = 25		
Probe Limit: Lp = 4		
Number of jobs in simulation run= 100,000 jobs		
Transients: 10% => 10,000 jobs		
Node	**Mean service time = 1/μ**	**Fastest/slowest**
1	1	fastest
2	3	slowest
3	1	fastest
4	1	fastest
5	2	
6	3	slowest
7	3	slowest
8	2	
μ_{avg} = 0.6250		

Table 8. Low heterogeneity system ($\mu_{fastest}/\mu_{slowest}$) = 3 (with Lp = 4)

1/λ	**λ / Nμ_{avg}**	**t_{resp}**		
		SED policy	**NQ policy**	**GT policy**
1.00	0.20	1.3533	1.2168	1.3272
0.80	0.25	1.4364	1.2781	1.3800
0.65	0.31	1.5436	1.3576	1.4526
0.50	0.40	1.7299	1.4695	1.5857
0.40	0.50	1.9652	1.6104	1.7575
0.30	0.67	2.4753	1.9557	2.1496
0.25	0.80	3.0830	2.5173	2.7524
0.22	0.91	4.5134	3.8363	4.1186
0.21	0.95	6.3382	6.0973	6.2644

Table 9. Mean response time of a low heterogeneity system (with Lp = 4)

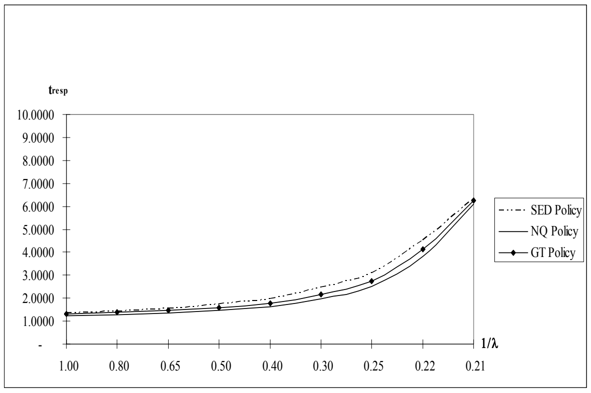


Figure 6. Mean response time of a system with low heterogeneity (and Lp = 4)

6. CONCLUSIONS

Several proposed adaptive load sharing algorithms from the literature are discussed and analyzed in this work. Three such algorithms are then modified to reflect the assumptions and more realistic operating conditions of the system. The modified solutions are simulated to compare their processing time, resource allocation, communication costs and sensitivity to parameter changes. Consequently, we can conclude from the results of the simulation runs that simple algorithms, that do not require much computational overhead or implementation complexity, can perform very well under most operating conditions, such as the Never Queue policy. Future work in this context could include adapting these algorithms to the case where the inter-arrival times and processing times are randomly distributed or time varying. It would also be interesting to try to modify these algorithms, or to construct new algorithms, to account for the case where certain tasks may have deadlines or may have priority over other tasks. One more aspect that can be studied in the future would be that, in most real time systems, the probability that a task initiates at certain node is not always the same for all nodes of the system. In fact, in many cases, it is always true that few nodes generate the jobs and have to redistribute them among the other nodes of the system.

REFERENCES

Aweya J., Oullette M., Montuno D.Y., Doray B. and FelskeK. 2002, "An Adaptive Load Balancing Scheme for Web Servers," *International Journal of Network Management*, vol. 12, Pp3-39.

Baumgartner, K.M. and Wah, B.W. 1991, “Computer Scheduling Algorithms: Past, Present, and Future,” *Information Science*, 57-58, Pp319-345.

Bokhari, S.H. 1981, “On the Mapping Problem,” *IEEE Trans. on Computers*, 30, 3, Pp550-557.

Casavant, T.L., and Kuhl, J.G. 1994, “A Taxonomy of Scheduling in General Purpose Distributed Computing Systems.” In T.L. Casavant and M. Singhal, ed., READINGS IN DISTRIBUTED COMPUTING SYSTEMS, IEEE Computer Society Press.

Chou, T.C.K. and Abraham, J.A. 1982, “Load Balancing in Distributed Systems,” *IEEE Trans. on Software Engineering*, 8, 4, Pp401-412.

Chow, Y.-C. and Kohler, W.H. 1977, “Dynamic Load Balancing in Homogeneous Two-Processor Distributed Systems,” In K.M. Chandy and M. Reiser, eds., COMPUTER PERFORMANCE, North Holland Publishing Company, Pp39-52.

Chow, Y.-C. and Kohler, W.H. 1982, “Models for Dynamic Load Balancing in Homogeneous Multiple Processing Systems,” *IEEE trans. on Computers*, 36, Pp .

Corradi A., Leonardi L and Zambonelli F. 1999, "Diffusive Load-Balancing Policies for Dynamic Application," *IEEE Concurrency*, Pp22-31.

Das S.K., Harvey D.J. and Biswas R. 2002, "Adaptive Load-Balancing Algorithms Using Symmetric Broadcast Networks," *J. Parallel and Distributed Computing*, vol. 62, Pp1042-1068.

Eager D., Lazowska E., and Zahorjan, J. 1986, "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Trans. Software Eng.* , vol. SE-12, Pp662-675.

Efe, K. 1982, "Heuristic Models of Task Assignment Scheduling in Distyributed Systems," *IEEE Computer*, 15, 6, Pp50-56.

Hac A. and Jin X. 1990, "Dynamic Load Balancing in a Distributed System using a Sender Initiated Algorithm", *J. Systems Software*, Pp79-94.

Harget, A.J. and Johnson, I.D. 1990, "Load Balancing Algorithms in Loosely-coupled Distributed Systems: A Survey." In H.S.M. Zedan, ed., DISTRIBUTED COMPUTER SYSTEMS: THEORY AND PRACTICE, Butterworths, Pp85-108.

Hou C.J. and Shin K.G. 1994, "Load Sharing with Consideration of Future Task Arrivals in Heterogeneous Distributed Real-Time Systems," *IEEE Trans. Computers*, vol. 43, Pp1076-1091.

Hu, Y.F. and Blake, R.J. 1999, "An Improved Diffusion Algorithm for Dynamic Load Balancing," *Parallel Computing*, 25, Pp417-444.

Hui C.-C. and Chanson S.T. 1999, "Improved Strategies for Dynamic Load Balancing," *IEEE Concurrency*, Pp58-67.

Karatza, H.D. 2001, "Job Scheduling in Heterogeneous Distributed Systems," *J. of Systems and Software*, vol. 56, Pp203-212.

Kremien O. and Kramer J. 1992, "Methodical Analysis of Adaptive Load Sharing Algorithms", *IEEE Trans. Parallel Distrib. Syst.* , vol. 3, Pp747-760.

Krueger P. and Livny M. 1988, "A Comparison of Preemptive and Non-Preemptive Load Distributing", *IEEE DCS-88*, Pp123-130.

Leff A. and Yu P.S. 1991, "An Adaptive Strategy for Load Sharing in Distributed Database Environments with Information Lags," *J. Parallel and Distributed Computing*, vol. 13, no. 1, Pp91-103.

Lewis, T.G. and El-Rewini, H. 1992, INTRODUCTION TO PARALLEL COMPUTING, Prentice-Hall, Inc..

Liao C.-J and Chung Y.-C. 1999, "Tree-Based Parallel Load-Balancing Methods for Solution-Adaptive Finite Element Graphs on Distributed memory Multicomputers," *IEEE Transactions on parallel and Distributed Systems*, vol. 10, no. 4, Pp360-370.

Lin F.C. and Killer R.M. 1986, "Gradient Model: A Demand-Driven Load Balancing Scheme", *IEEE Conf. On Distributed Computing Systems* , Pp329-336.

Lin H.C., and Raghavendra C.S. 1991, "A Dynamic Load Balancing Policy with a Central Job Dispatcher", *IEEE Conf. On Distributed Computing Systems* , Pp264-271.

Lo, V.M. 1988, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans. on Computers*, 31, 11, Pp1384-1397.

Mirchandaney R., Towsley D. and Stankovic J. 1989, "Analysis of the Effects of Delays on Load Sharing," *IEEE Trans. Computers*, vol. 38, Pp1513-1525.

Pingali S., Towsley D and Kurose J.F. 1994, "A Comparison of Sender Initiated and Receiver Initiated Reliable Multicast Protocols*", The ACM Sigmetrics Corp.,* Pp221-230.
Rotithor H.G. 1994, "Taxonomy of Dynamic Task Scheduling Schemes in Distributed Computing Systems", *IEEE Proc.- Comput. Digit. Tech.* , vol. 141, Pp1-10.

Schaar M., Efe K., Decambre L. and Bhuyan L.N. 1991, "Load Balancing with Network Cooperation", *IEEE Conf. DCS*, Pp328-335.

Shenker S. and Weinrib A. 1989, "The Optimal Control of Heterogeneous Queuing Systems: A Paradigm for Load Sharing and Routing", *IEEE Trans. Computers*, vol. 38, Pp1724-1735.

Shin K.G. and Chang Y.C. 1995, "A Coordinated Location Policy for Load Sharing in Hypercube-Connected Multicomputers," *IEEE Trans. Computers*, vol. 44, Pp669-682.

Shin K.G. and Hou C.J. 1993, "Analytic Models of Adaptive Load Sharing Schemes in Distributed Real-Time Systems", *IEEE Trans. Parallel Distrib. Syst.* , vol. 4, Pp740-747.

Shin K.G. and Hou C.J. 1996, "Evaluation of Load Sharing in HARTS with Consideration of Its Communication Activities", *IEEE Trans. Parallel Distrib. Syst.* , vol. 7, Pp724-739.

Sofianopoulou, S. 1992, "The Process Allocation Problem: A Survey of the Application of Graph-Theoretic and Integer Programming Approaches," *J. of Operational Research*, 43, 5, Pp407-413.

Touheed N., Selwood P, JimackP.K. and Berzins M. 2000, "A Comparison of Some Dynamic Load-Balancing Algorithms for a Parallel Adaptive Flow Solver," *Parallel Computing*, vol. 26, Pp1535-1554.

Weinrib A. and Shenker S. 1988, “Greed is not enough: Adaptive Load Sharing in Large Heterogeneous Systems”, *INFOCOM*, Pp986-994.

Xu C.-Z. and Lau F.C.M. 1997, LOAD BALANCING IN PARALLEL COMPUTERS: THEORY AND PRACTICE, Kluwer Academic Press.

Xu J. and Hwang K. 1993, "Heuristic methods for Dynamic Load Balancing in a Message-Passing Multicomputer," *J. Parallel and Distributed Computing*, vol. 15, no. 1, Pp1-13.

Zhou S. 1988, “A trace-driven simulation study of dynamic load balancing”, *IEEE Trans. Software Eng.*, vol. 14, 9, Pp1327-1341.

Zhou S. and Ferrari D. 1987, “A Measurement Study of Load Balancing Performance”, *IEEE Conf. On Distributed Systems*, Pp490-497.

KARIM Y. KABALAN was born in Jbeil, Lebanon. He received the B.S. degree in Physics from the Lebanese University in 1979, and the M.S. and Ph.D. degrees in Electrical Engineering from Syracuse University, in 1983 and 1985, respectively. During the 1986 Fall semester, he was a visiting assistant professor of Electrical Engineering at Syracuse University. Currently, he is a Professor of Electrical Engineering with the Electrical and Computer Engineering Department, Faculty of Engineering and Architecture, American University of Beirut. His research interests are numerical solution of electromagnetic field problems and software development.

WALEED W. SMARI received his Ph.D. degree in Electrical and Computer Engineering from Syracuse University, Syracuse, NY, in 1996. He has a B.S. in Electrical Engineering, M.S. in Electrical Engineering, and M.S. in Computer Engineering from Syracuse University. During his last year at Syracuse, he was a faculty member at the Department of Engineering at the State University of New York, New Paltz, NY. Currently, he is an Associate Professor at the Department of Electrical and Computer Engineering, University of Dayton, Dayton, OH. His technical interests and specialties include performance evaluation methods and modeling techniques of computing systems, parallel and distributed processing and networking, digital systems design, scheduling theory and algorithms, and computer engineering education. He is a member of the IEEE, ACM, ASEE, IASTED, ISCA, SCS, and MASIC-VIUF.

JACQUES Y. HAKIMIAN was born in Lebanon in 1974. He received his B.E. and MS. degrees in Electrical and Computer Engineering from the American University of Beirut in 1996