A shorter version of this report appears in *Proc. IEEE MASCOTS'97, Haifa, Israel, January 1997.*

Hierarchical Load Sharing Policies for Distributed Systems

S. Dandamudi and Michael Lo
Centre for Parallel and Distributed Computing
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6, Canada

Technical Report SCS-96-1
January 1996 (revised November 1996)

ABSTRACT – Performance of distributed systems can be improved by load sharing. Dynamic load sharing policies take the system state into account in making load distribution decisions. The system state information can be collected in a distributed manner or by a single central controller node. In the distributed scheme, each node gathers the current system state information before making a decision on load distribution. Load sharing policies based on this strategy typically probe a few randomly selected nodes for their status in order to find a suitable partner for load distribution. Two principal policies that have been studied extensively in the literature are the sender-initiated and receiver-initiated policies. In the centralized scheme, a central node (called the “coordinator”) is assigned the responsibility of collecting the system state information. Any other node that needs to distribute load consults the coordinator node for a suitable partner. The distributed policies do not perform as well as the centralized policy. Performance of distributed policies is sensitive to variance in job service times and inter-arrival times. The distributed policies, on the other hand, are scalable to large systems whereas the centralized policy causes bottleneck problems for large systems. Another problem with the centralized scheme is that if the coordinator fails, the system defaults to “no-load sharing” scenario. Here we propose a hierarchical load sharing policy that minimizes the drawbacks associated with the distributed and centralized policies while retaining their advantages. We propose and study the performance of the hierarchical load sharing policies. The results presented here show that the hierarchical policies inherit the merits of the centralized and distributed policies while minimizing the disadvantages associated with them.

Index Terms: Dynamic load sharing, performance evaluation, hierarchical load sharing policies, homogeneous distributed systems, sender-initiated policies, receiver-initiated policies.

1. INTRODUCTION

Performance of distributed systems can be improved by load sharing. Load sharing attempts to distribute system workload from heavily loaded nodes to lightly loaded nodes in the system. Load sharing policies can be either static or dynamic. Static load sharing policies do not require system state information in making load distribution decisions. Dynamic policies, on the other hand, make their load distribution decisions based on the current (or most recent) system state. By reacting to the system state changes dynamically, dynamic load sharing policies tend to provide significant performance improvements compared to static policies and no load sharing. This paper considers dynamic load sharing in homogeneous distributed systems in which the nodes are identical and the workload of each node is similar.

Two important components of a dynamic policy are a transfer policy and a location policy [Eag86a]. The transfer policy determines whether a job is processed locally or remotely and the location policy determines the node to which a job, selected for possible remote execution, should be sent. Typically, transfer policies use some kind of load index threshold to determine whether the node is heavily loaded or not. Several load indices such as the CPU queue length, time averaged CPU length, CPU utilization, the amount of available memory etc. have been proposed/used [Zho88, Kun91, Shi92]. It has been reported that the choice of load index has considerable effect on the performance and that the simple CPU queue length load index was found to be the most effective [Kun91].

The dynamic policies can employ either a centralized or a distributed location policy. In the centralized policy, state information is collected by a single node (called the “coordinator”) and other nodes would have to consult this node for advice on the system state. In the distributed policy, system state information is distributed to all nodes in the system. The centralized policy has the advantage of providing near perfect load sharing as the coordinator has the entire system state to make a load distribution decision. The obvious disadvantages are that it suffers from diminished fault-tolerance and the potential for performance bottleneck. In this context, it should be noted that some studies have shown that the node collecting and maintaining the system state need not be a performance bottleneck for reasonably large distributed systems [The88]. However, if the system is geographically distributed (for example, several LAN clusters of nodes interconnected by a WAN), consulting a central node is very expensive and causes performance problems. Thus, the use of the centralized policy is often limited to a cluster of nodes in a large distributed system [Zho93].

The distributed policy eliminates these disadvantages associated with the centralized policy; however, distributed policy may cause performance problems as the state information may have to be transmitted to all nodes in the system. Previous studies have shown that this overhead can be substantially reduced by sampling only a few randomly selected nodes [Eag86a, Eag86b]. A further problem with the distributed policies is that the performance of such policies is sensitive to variance in service times as well as inter-arrival times [Dan95]. Distributed policies are, however, scalable to large system sizes.

In this paper, we propose a new hierarchical load sharing policy that combines the merits of the centralized and distributed policies while eliminating/minimizing the disadvantages of these policies. The performance of the proposed hierarchical policy along with two of its variants is studied extensively. The results reported here suggest that the hierarchical policy provides substantial performance improvements over the sender-initiated and receiver-initiated policies; its performance is closer to that of the centralized policy while providing scalability and fault-tolerance closer to that of the distributed policies.

Location policies can be divided into two basic classes: sender-initiated or receiver-initiated. In sender-initiated policies, congested nodes attempt to transfer work to lightly loaded nodes. In receiver-initiated policies, lightly loaded nodes search for congested nodes from which work may be transferred. It has been shown that, when the first-come/first-served (FCFS) scheduling policy is used, sender-initiated policies perform better than receiver-initiated policies at low to moderate system loads [Eag86b]. This is because, at these system loads, the probability of finding a lightly loaded node is higher than that of finding a heavily loaded node. At moderate to high system loads, on the other hand, receiver-initiated policies are better because it is much easier to find a heavily loaded node at these system loads. There have also been proposals to incorporate the good features of both these policies [Shi90, Shi92].

In the remainder of this section, we briefly review the related work and outline the paper. There is a lot of literature published on this topic (an excellent overview of the topic can be found in [Shi92]) [Alo88, Cho90, Eag86a, Eag86b, Hac87, Hac88, Kru88, Kun91, Lin92, Lit88, Liv82, Rom91, Sch91, Shi90, The88, Wan85, Zho88]. Here, for the sake of brevity, we only review a subset of this literature that is directly relevant to our study. Eager et al. [Eag86a, Eag86b] provide an analytic study of sender-initiated and receiver-initiated load sharing policies. They have shown that the sender-initiated location policy performs better at low to moderate system loads and the receiver-initiated policies perform better at high system loads. They have also shown that the overhead associated with state information collection and maintenance under the distributed policy can be reduced substantially (by probing only a few randomly selected nodes about their state). Shivaratri and Krueger [Shi90] have proposed and evaluated, using simulation, adaptive location policies that behave like a sender-initiated policy at low to moderate system loads and as a receiver-initiated policy at high system loads. Dikshit, Tripathi, and Jalote [Dik89] have implemented both sender-initiated and receiver-initiated policies on a five node system connected by a 10Mb/s communication network. They report performance of several load sharing policies.

The remainder of the paper is organized as follows. Section 2 discusses the three load sharing policies against which the performance of the proposed hierarchical policy is compared. The proposed hierarchical policy is described in Section 3. The next section describes the workload and system models that have been used in this study. The results are discussed in Section 5. Section 6 presents two variations of the hierarchical policy. This section also discusses the performance of these hierarchical policies. Fault-tolerance and other issues are discussed in Section 7. Conclusions are given in Section 8.

2. LOAD SHARING POLICIES

This section gives a brief description of the three load sharing policies used to compare the performance of the proposed hierarchical policy. These policies are the centralized policy (the single coordinator policy) and two distributed policies (sender-initiated and receiver-initiated policies). Throughout this paper we assume the first-come/first-served (FCFS) node scheduling policy. In addition, job transfers between nodes are done on a non-preemptive basis. The rationale for this choice is that it is expensive to migrate active processes and, for performance improvement, such active process migration is strictly not necessary. It has been shown that, except in some extreme cases, active process migration does not yield any significant additional performance benefit [Eag88, Kru88]. Load sharing facilities like Utopia [Zho93] will not consider migrating active processes.

In all the load sharing policies discussed in this paper, the following transfer policy is used. When a new job arrives at a node, the transfer policy looks at the job queue length at the node. This queue length includes the jobs waiting to be executed and the job currently being executed. The new job is transferred for local execution if the queue length is less than the specified threshold value T. Otherwise, the job is eligible for a possible remote execution and is placed in the job transfer queue. The location policy, when invoked, will actually perform the node assignment.

2.1. Sender-Initiated Policy

When a new job arrives at a node, the transfer policy described above would decide whether to place the job in the job queue or in the job transfer queue. If the job is placed in the job transfer queue, the job is eligible for transfer and the location policy is invoked. The location policy probes (up to) a maximum of probe limit P_l randomly selected nodes to locate a node with the job queue length less than T. If such a node is found, the job is transferred to that node for remote execution. The transferred job is directly placed in the destination node's job queue when it arrives. Note that probing stops as soon as a suitable target node is found. If all P_l probes fail to locate a suitable node, the job is moved to the job queue to be processed locally. When a transferred job arrives at the destination node, the node must accept and process the transferred job even if the state of the node at that instance has changed since probing.

2.2. Receiver-Initiated Policy

When a new job arrives at node S, the transfer policy would place the job either in the job queue or in the job transfer queue of node S as described before. The location policy is typically invoked by nodes at times of job completions. The location policy of node S attempts to transfer a job from its job transfer queue to its job queue if the job transfer queue is not empty. Otherwise, if the job queue length of node S is less than T, it initiates the probing process as in the sender-initiated policy to locate a node D with a non-empty job transfer queue. If such a node is found within P_l probes, a job from the job transfer queue of node D will be transferred to the job queue of node S. In this paper, as in the previous literature [Eag86b, Shi92], we assume that $T = 1$. That is, load distribution is attempted only when a node is idle (Livny and Melman [Liv82] call it 'poll when idle' policy). The motivation is that a node is better off avoiding load distribution when there is work to do. Furthermore, several studies have shown that a large percentage (up to 80% depending on time of day) of workstations are idle [Mut87, Nic87, Lit88, The88, Kru91]. Thus the probability of finding an idle workstation is high.

Previous implementations of this policy have assumed that, if all probes fail to locate a suitable node to get work from, the node waits for the arrival of a local job. Thus, job transfers are initiated at most once every time a job is completed. This causes performance problems because the processing power is wasted until the arrival of a new job locally. This poses severe performance problems if the load is not homogeneous (e.g. if only a few nodes are generating the system load) [Shi92] or if there is a high variance in job inter-arrival times [Dan95]. For example, if four jobs arrive at a node in quick succession, then this node attempts load distribution only once after completing all four jobs. Worse still is the fact that if there are long gaps in job arrivals, the frequency of load distribution will be low. This adverse impact on performance can be remedied by reinitiating load distribution after the elapse of a predetermined time if the node is still idle. The receiver-initiated policy implemented here uses this reinitiation strategy.

2.3. Single Coordinator Policy

In this policy. there is a single node (called the "coordinator") that is responsible for collecting the system state information. Whenever that state of a node changes, it informs this change in state to the coordinator for updating purposes. A node can be in one of three states:

- it is in *receiver* state if the job queue length of the node is less than T_l (low threshold);
- it is in *sender* state if the job queue length of the node is greater than T_h (high threshold);
- otherwise, it is in OK state.

where $T_h \geq T_l$. In this paper, for reasons explained in Section 2.2, we assume that $T_h = T_l = T = 1$.

The load distribution is initiated by a receiver node (i.e., a node that is in the receiver state). Typically, at job completion times, if the state of the node changes to receiver, it consults the coordinator node for a node that is in the sender state. If a sender node is found, the coordinator informs the sender node to transfer a job to the receiver node. As in the receiver-initiated policy, reinitiation of load distribution is necessary in order to improve its performance under certain system and workload conditions.

3. Hierarchical Load Sharing Policy

3.1. Motivation

Most load sharing policies that have been proposed in the literature use distributed location policies. These include the sender-initiated and receiver-initiated policies discussed in the last section. Previous studies have shown that state information from a few randomly selected nodes is sufficient to provide substantial improvement in performance over no load sharing. Also, probing a significantly larger number of nodes yields only marginal additional improvement in performance. While the distributed policies are scalable to large systems, the performance of these policies is sensitive to variance in service times and inter-arrival times.

The centralized policy, described in the last section, provides near perfect load sharing. For example, this policy exhibits least sensitivity to variance in service times and inter-arrival times. However, this policy does not scale well for large systems. In addition, the single coordinator causes fault-tolerance problems. In this context, it should be noted that some studies have shown that the node collecting and maintaining the system state need not be a performance bottleneck for reasonably large distributed systems [The88]. However, if the system is geographically distributed (for example, several LAN clusters interconnected by a WAN), consulting a central node is very expensive and causes performance problems. Thus, the use of the centralized policy is often limited to a cluster of nodes in a large distributed system [Zho93].

Thus we would like to have a policy that provides performance very close to that of the centralized policy while inheriting the merits of the distributed policies. We propose the use of hierarchy to achieve this objective. We will show that the hierarchical policy, described next, accomplishes this objective.

3.2. Proposed Hierarchical Policy

In the hierarchical policy, instead of a single node maintaining the entire system state, a set of nodes is given this responsibility. The system is logically divided into clusters and each cluster of nodes will have a single node that maintains the state information of the nodes within the cluster. The state information on the whole system is maintained in the form of a tree where each tree node maintains the state information on the set of processor nodes in the sub-tree rooted by the tree node. Figure 3.1 shows an example hierarchical organization for 8 processor nodes with a branching factor B of 2. Recall that a node can be in one of three states: *sender* (overloaded), *OK* (normal load), or *receiver* (underloaded). We will use +1 to represent the sender state, 0 for the OK state, and -1 for the receiver state. For example, Q4 maintains the state information in nodes N0 ands N1.

It may be noted from Figure 3.1 that each node in the tree maintains the state information on all the nodes in the sub-tree rooted at this tree node. In other words, cluster size increases as we move up the tree. For example, Q2 maintains state information on nodes N0, N1, N2, and N3. However, in order to reduce the frequency of updates as well as to reduce the amount of information that has to be kept at higher tree nodes, only summary state information is maintained. The summary metric used in this policy is the arithmetic sum of the state information of the tree nodes below it. For example, summary state metric for Q4 is zero, which implies that the cluster represented by Q4 (i.e., nodes N0 and N1) is in OK state. Thus this policy encourages local load balancing. For this reason, this policy is called the *local hierarchical* policy. For example, the summary metric for Q6 stored in Q3 is +1 rather than +2. Note that the value stored in the parent node is +1 if the sum is positive (but not zero), -1 if negative, and 0 if zero. An advantage of this scheme is that it reduces the number of updates required to maintain the hierarchy. For example, if the system state changes and N4 moves to OK state, only the entry in Q6 need to be changed. Since N5 is still a sender, no state change is necessary for Q3. Section 7 discusses the impact of maintaining a true summary metric. In addition, two global hierarchical policies are discussed in Section 6.

Since load sharing can be done at various levels of the tree, the root node does not have to handle requests form all nodes in the system. Furthermore, it can also be seen that the set of nodes that form the tree can all be distributed to different system nodes so that no node is unduly overloaded with system state information. For a system with N nodes, the maximum number (N-1) of tree nodes will be required when the branching factor B is 2. Since there are N system nodes these (N-1) tree nodes can be distributed uniformly.

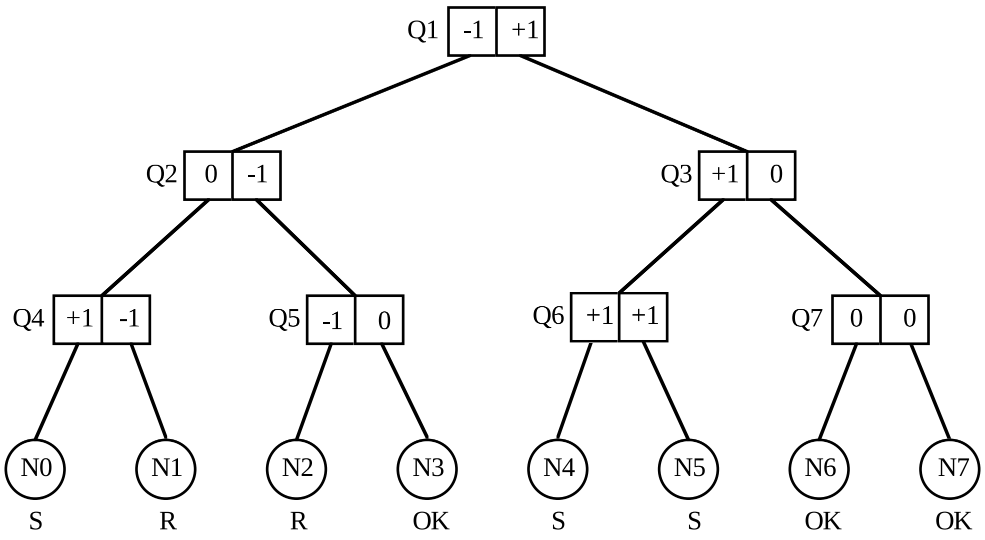


Figure 3.1 An example local hierarchical organization for an 8-node system with a branching factor of 2

The tree branching factor is a design issue in such an organization. We wish to keep the tree as "bushy" as possible (i.e., a tree with a large branching factor) because it allows better load sharing and results in better performance. Taken to the extreme this represents the centralized organization. Section 6.3 considers the impact of the branching factor on the performance of the hierarchical policy.

With this hierarchical structure in place, various load sharing policies can be implemented on top of it. Here we describe a receiver-initiated policy that works with the hierarchy. As explained in Section 2.2, we assume a threshold *T* of 1. That is, whenever a node idle, it consults the hierarchy (explained next) for a sender node from which the receiver can get a job. The local hierarchical policy implemented works as follows.

When a job is complete at node S, if there are no jobs in its local job queue, it sends a message to its parent tree queue (i.e., to the processor node that is responsible for maintaining this tree node) for a sender node. If the tree node does not have any sender nodes in its sphere, it forwards the message to its parent tree node. This process is repeated up the tree until either a tree node with a sender node is encountered or the root node is reached. If the root node does not have a sender node implying that there is no sender node in the whole system, a “no job” message is sent back to the receiver node that initiated the request. If a tree node with at least one sender branch is found, the message follows a “downward” path along this branch until it reaches the sender node. If the tree node has more than one sender branch, one is selected at random.

When the message reaches the sender node, if that node is still a sender, it transfers a job to the receiver node. If, on the other hand, the node is no longer a sender (as it is in the process of updating its entry in the hierarchy), it sends a “false sender” message to the receiver indicating that there is no job to transfer. When a “no job” or a “false sender” message is received by the receiver node and if the node is still in the receiver state (i.e., there were no local job arrivals since the request) it updates it entry in the hierarchy (to receiver state) and waits for the corresponding reinitiation period and initiates another load distribution request (if it remains in the receiver state at the end of the reinitiation period).

4. System and Workload Models

In the simulation model, a locally distributed system is represented by a collection of nodes. In this paper we consider only homogeneous nodes. We also model the communication network in the system at a higher level. We model communication delays without modelling the low-level protocol details. An Ethernet-like network with 10 Mbits/sec is assumed. The communication network is modelled as a single server. Each node is assumed to have a communication processor that is responsible for handling communication with other nodes. Similar assumptions are made by other researchers [Mir89]. The CPU would give preemptive priority to communication activities (such as reading a message received by the communication processor, initiating the communication processor to send a probe message etc.) over the processing of jobs.

The CPU overheads to send/receive a probe and to transfer a job is modelled by T_{probe} and T_{jx}, respectively. Actual job transmission (handled by the communication processor) time is assumed to be uniformly distributed between U_{jx} and L_{jx}. Probing is assumed to be done serially, not in parallel. For example, the implementation in [Dik89] uses serial probing.

The system workload is represented by four parameters. The job arrival process at each node is characterized by a mean inter-arrival time $1/\lambda$ and a coefficient of variation C_a. Jobs are characterized by a processor service demand (with mean $1/\mu$) and a coefficient of variation C_s. We study the performance sensitivity to variance in both inter-arrival times and service times (the CV values are varied from 0 to 4). We have used a two-stage hyperexponential model to generate service time and inter-arrival time CVs greater than 1 [Kob81].

As in most previous studies, we model only CPU-intensive jobs in this study. We use the mean response time as the chief performance metric to compare the performance of the various load sharing policies.

5. PERFORMANCE ANALYSIS

This section presents the simulation results and discusses the performance sensitivity of the four load sharing policies. Unless otherwise specified, the following default parameter values are assumed. The distributed system is assumed to have N = 32 nodes interconnected by a 10 megabit/second communication network. The average job service time is one time unit. The size of a probe message is 16 bytes. The CPU overhead to send/receive a probe message T_{probe} is 0.003 time units and to transfer a job T_{jx} is 0.02 time units. The load distribution reinitiation period when a "no job" ("false sender") message is received is fixed at 1 (0.2). Job transfer communication overhead is uniformly distributed between L_{jx} = 0.009 and U_{jx} = 0.011 time units (i.e., average job transfer communication overhead is 1% of the average job service time). Since we consider only non-executing jobs for transfer, 1% is a reasonable value. Note that transferring active jobs would incur substantial overhead as the system state would also have to be transferred. A probe limit P_l of 3 is used for the sender-initiated and receiver-initiated policies. For the hierarchical policy, the default branching factor is fixed at 4. Section 6.3 discusses the impact of this parameter on the performance of the hierarchical policy.

Batch strategy has been used to compute confidence intervals (at least 30 batch runs were used for the results reported here). This strategy has produced 95% confidence intervals that were less than 1% of the mean response times when the system utilization is low to moderate and less than 5% for moderate to high system utilization.

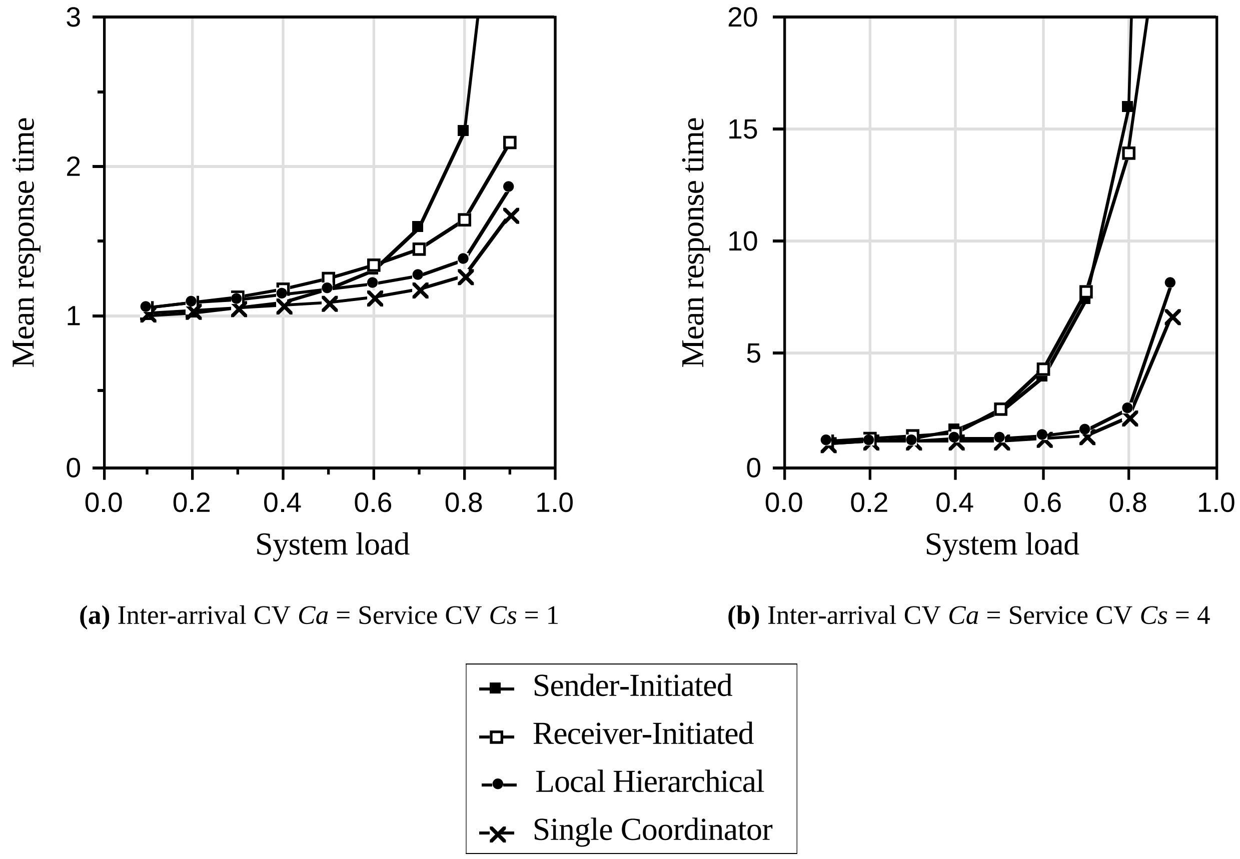
5.1. Performance as a function of system load

Figure 5.1 shows the mean response time of the four load sharing policies as a function of offered system load. Note that the offered system load is given by λ/μ. Since μ=1 for all the experiments, offered system load is equal to λ. The results in Figure 5.1a correspond to inter-arrival CV (C_a) and service time CV (C_s) of 1 and those in Figure 5.1b were obtained with the inter-arrival time CV and service time CV of 4.

C_a =1 and C_s =1

When $C_a = C_s$ =1, the sender-initiated policy performs better than the receiver-initiated policy for system loads up to about 60%. At higher loads, the receiver-initiated policy is better. This is a well-known result [Eag86b,Shi92]. At low system loads, the performance of the sender-initiated policy is similar to that of the single coordinator policy. This is because the sender-initiated policy is able to successfully transfer a job as the probability of finding a node below the threshold (in our case, an idle node as the threshold is 1) is very high at these system loads. On the other hand, the performance of the hierarchical policy is closer to that of the receiver-initiated policy. The main reason for this performance deterioration is that, even though most nodes are idle, idle nodes seek work at the reinitiation period of 1. Thus, a node that has become a sender would have to wait some time before a receiver node requests transfer of this job. In contrast, the sender-initiated policy tends to transfer the job that has arrived at a busy node almost immediately (i.e., after finishing probing). In the single coordinator policy, the reinitiation period does not cause as much delay as in the hierarchical policy because the central node is consulted by all the nodes in the system.

At higher system loads, the performance of the sender-initiated policy deteriorates rapidly. This is because, it is harder to find a receiver node at these system loads. (At high system loads, the performance of the sender-initiated policy can be improved by increasing the threshold value, for example, to 2). For precisely the same reason, the receiver-initiated policy provides a substantially better performance than the



(a) Inter-arrival CV *Ca* = Service CV *Cs* = 1

(b) Inter-arrival CV *Ca* = Service CV *Cs* = 4

Figure 5.1 Performance sensitivity as a function of offered system load

(N = 32 nodes, μ = 1, λ is varied to vary system load, T = 1, P_l = 3, B = 4, transfer cost = 1%)

sender-initiated policy. However, its performance is still far below that of the single coordinator policy as shown in Figure 5.1a. The hierarchical policy, on the other hand, provides performance that is very close (less than 10%) to that of the single coordinator policy. At higher system loads, the performance of the hierarchical policy is substantially better than that of the receiver-initiated policy. For example, at offered system load of 80%, the receiver-initiated policy performs approximately 20% worse than the hierarchical policy.

C_a = 4 and C_s = 4

When C_a = C_s = 4, both sender-initiated and receiver-initiated policies provide similar performance as shown in Figure 5.1b (receiver-initiated policy performs slightly better than the sender-initiated policy at high system loads). This performance similarity is due to the following facts: (i) the sender-initiated policy is more sensitive (than the receiver-initiated policy) to the variance in service times, and (ii) the receiver-initiated policy is more sensitive to the variance in inter-arrival times [Dan95]. As the variance is high in both service times and inter-arrival times in Figure 5.1b, there is no significant difference in performance between these two policies. It should be noted that the performance of the sender-initiated policy will be better than the receiver-initiated policy if the threshold is increased to 2. However, our goal here is not to study such sensitivity of sender-initiated and receiver-initiated policies (see [Dan95] for details on this).

It may be noted from Figure 5.1b that the performance of the hierarchical policy is substantially better (particularly at moderate to high system loads) than both the sender-initiated and receiver-initiated policies. For example, the response time decreases from about 14 (with the receiver-initiated policy) to about 2.5 with the hierarchical policy when the offered system load is 80%. However, the difference between the response times of the single coordinator and hierarchical policies increases with the system load. The next two sections discuss the sensitivity of the performance of the four load sharing policies to variance in inter-arrival times and service times.

5.2. Sensitivity to variance in inter-arrival times

This section considers the impact of inter-arrival CV C_a on the performance of the four load sharing policies. Figure 5.2 shows the mean response time when the offered system load is fixed at 80% (i.e., λ = 0.8) and service time CV C_s at 1. Since the performance of the sender-initiated policy improves when the threshold value is increased from 1 to 2 at high system loads (here the system load is fixed at 80%), we have used T = 2 for the sender-initiated policy in this and the next section. All other parameter values are set as in Figure 5.1. The main observation from this figure is that the hierarchical policy exhibits a much more robust behaviour to variance in inter-arrival times than the sender-initiated and receiver-initiated policies.

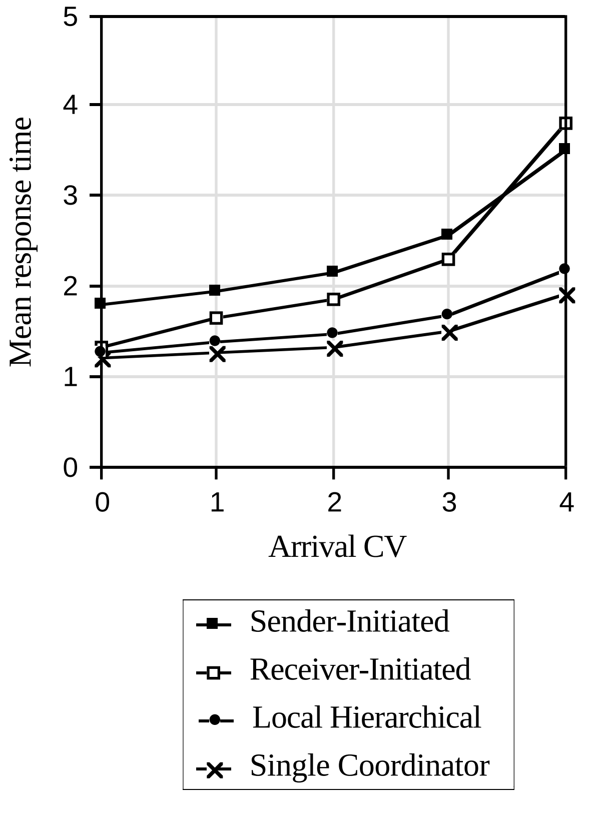


Figure 5.2 Performance sensitivity to variance in inter-arrival times (N = 32 nodes, λ = 0.8, Ca is varied from 0 to 4, μ = 1, Cs = 1, T = 2 for the sender-initiated policy and T= 1 for all other policies, P_l = 3, B = 4, transfer cost = 1%)

Note that higher inter-arrival CV implies clustered nature of job arrivals at each node in the system; the higher the CV the more clustered the job arrival process is. The sender-initiated policy is relatively less sensitive to the inter-arrival CV than the receiver-initiated policy. The reason for this sensitivity is that increased CV implies the clustered nature of job arrivals into the system and this decreases the probability of finding an idle node. Thus, increasing the inter-arrival CV moves the system based on the receiver-initiated policy towards a no-load sharing system. On the other hand, clustered arrival of jobs fosters increased load sharing in sender-initiated policies (see [Dan95] for a more detailed explanation). The performance difference between sender-initiated and receiver-initiated policies at low CV values is due to the the use two different threshold values.

The hierarchical policy is less sensitive than the sender-initiated and receiver-initiated policies and its performance is closer to that of the single coordinator policy. However, as the inter-arrival CV increases, the difference between these two policies increases. This is mainly due to the fact that increased CV results in a more imbalanced system and, therefore, increases load distribution activity in all policies. Since the hierarchical policy involves more messages per job transfer than the single coordinator policy, the difference in response times increases with inter-arrival CV.

5.3. Impact of variance in service times

This section considers the impact of service time CV Cs on the performance of the four policies. In this context, it should be noted that the service time distribution of the data collected by Leland and Ott [Lel86] from over 9.5 million processes has been shown to have a coefficient of variation of 5.3 [Kru88].

Figure 5.3 shows the mean response time when the offered system load is fixed at 80% (i.e., λ = 0.8) and the inter-arrival time CV Ca at 1. All other parameter values are set as in Figure 5.2. The sender-initiated policy exhibits a high degree of sensitivity to the service time variance. At high Cs (for example, when Cs = 4) receiver-initiated policy performs substantially better than the sender-initiated policies as shown in Figure 5.3. The reasons for the behaviour of the sender-initiated and receiver-initiated policies have been documented in [Dan95]. Briefly, with the FCFS node scheduling policy, larger jobs tend to monopolize the processing power, which causes increased queue lengths resulting in increased probing activity under the sender-initiated policy. However, this increase in probing activity is useless because, at high system loads, the probability of finding a receiver node is low. On the other hand, the receiver-initiated policy can successfully locate a sender because, at high system loads, the probability of finding an overloaded node is high. Thus, in this case, increased probing activity results in increased job transfers (thereby increasing load sharing). This results in better performance and reduced sensitivity to service time variance Cs.

The hierarchical policy provides further improvements in performance over that of the receiver-initiated policy. The hierarchical policy exhibits less sensitivity to service time CV than the receiver-initiated policy and its performance is closer to that of the single coordinator policy as shown in Figure 5.3. The performance difference between the hierarchical and single coordinator policies increases with service time CV for reasons discussed in Section 5.2.

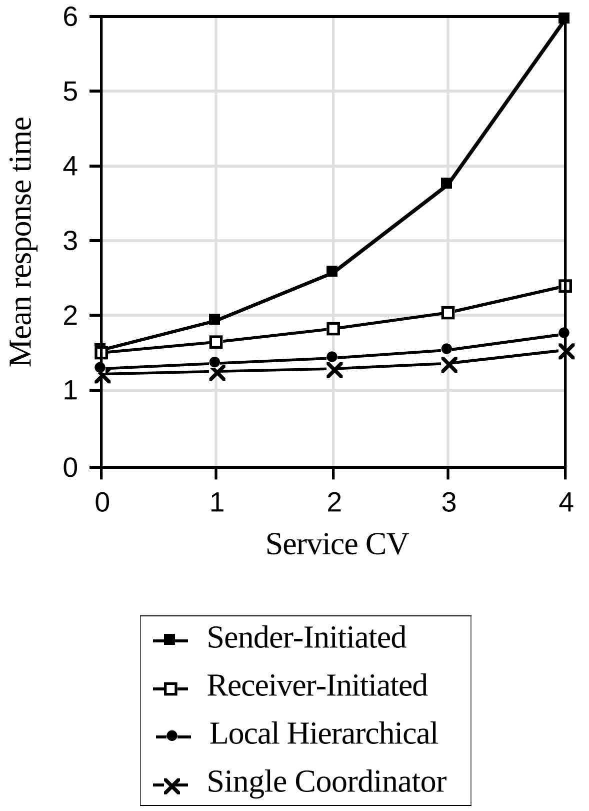


Figure 5.3 Performance sensitivity to service time CV (N = 32 nodes, λ = 0.8, Ca = 1, μ = 1, Cs is varied from 0 to 4, T = 2 for the sender-initiated policy and T= 1 for all other policies, P_l = 3, B = 4, transfer cost = 1%)

6. TWO VARIATIONS OF THE HIERARCHICAL POLICY

The hierarchical policy described in Section 3 tends to achieve load sharing by sharing load locally. This section describes two global hierarchical policies and compares the performance of the three hierarchical policies.

6.1. Global Hierarchical Policies

The scheme described in Section 3 encourages local load sharing. For example, in Figure 3.1, the fact the node N0 is a sender is not known beyond the cluster of N0 and N1 nodes. Thus, if for example, there is a receiver node that is looking for a sender, it will not be informed of the existence of N0. Note, however, that this is a transient situation as the receiver node N1 eventually gets a job transferred from node N0. This job transfer might be delayed for some time depending on the reinitiation period. If node N0 remains a sender after the job transfer, this information will be made available to other nodes in the system. The net effect is that the local policy sometimes delays informing other nodes of the existence of a sender node by giving priority to local load sharing. This type of load sharing is preferred when the system consists of LAN clusters connected together by a WAN. If, on the other hand, the system is simply a single cluster of nodes connected by a LAN, performance improvements can be obtained by reducing this delay. This section proposes a variant of the local hierarchical policy that achieves this objective.

In the global hierarchical policy, load distribution is initiated by a receiver node as in the local hierarchical policy. It, however, differs from the local policy in the type of information maintained in the hierarchy. In the global policy, the existence of a sender node is publicized as widely as possible so that any receiver node in the whole system can initiate a job transfer. This is suitable for systems that do not have inherent hierarchical structure in the communication network (as indicated before, the system is assumed to be a single cluster of nodes). Figure 6.1 shows the information maintained by the hierarchy in the global policy for the same node states as in Figure 3.1.

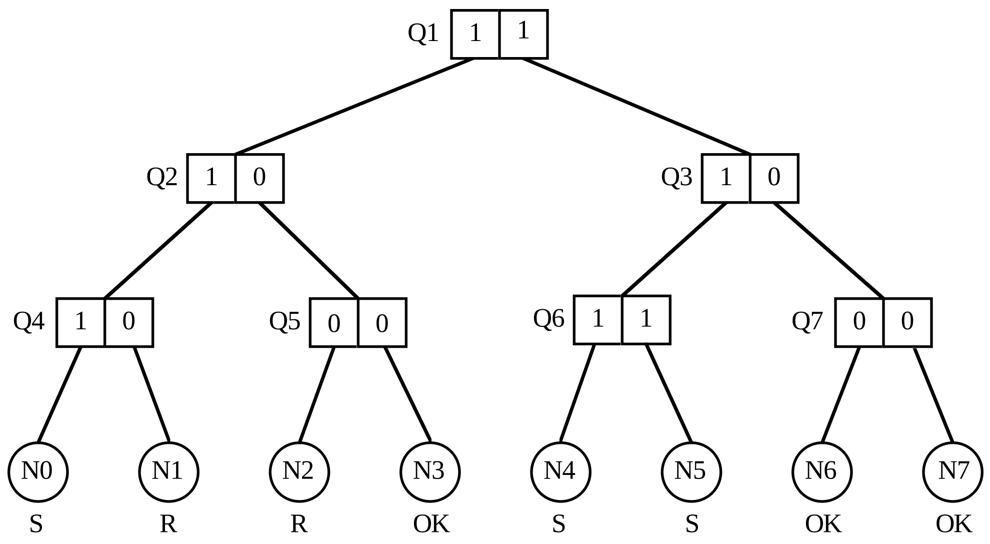


Figure 6.1 An example global hierarchical organization for an 8-node system with a branching factor of 2

It may observed from this figure that the hierarchy maintains state information about two states only: the sender state is indicated by a 1 and the non-sender state (receiver or OK) is represented by a 0. Thus, the semantics of the state information kept in the tree nodes are: if there is at least one sender in that branch, a 1 is stored to indicate this fact; otherwise a 0 is stored. For example, the first value in Q2 is 1, which represents the fact that the left sub-tree has at least one sender.

In this and the local policy described in Section 3, a receiver initiates load distribution by requesting a job from a sender node. This request flows through the hierarchy (up the tree and then down the tree) and finally terminates at a sender node (if there is one). This sender node directly replies to the receiver node that initiated the load distribution by either sending a job (if the sender node is in the sender state) or sends a "false sender" message if the state has changed from the sender to OK.[1] We refer to this as the false sender scenario. In the recursive version of the global policy, when the message ends up with a false sender, the search process continues by back tracking until a "true sender" is found[2]. Thus the recursive version improves performance further at the expense of increased message activity. The next section compares the performance of the three hierarchical policies.

[1]The state information in the hierarchy might be out of date temporarily because of message delays associated with updating the hierarchy.

[2] Unless thare is no sender node in the entire system; in this case, the root tree node sends a "no job" message to the originator of the request.

6.2. Performance Comparison

Figure 6.2 shows the performance of the three hierarchical load sharing policies. The results are for a system with 32 nodes and the branching factor is fixed at 4. The service time CV and the inter-arrival time CV are both 4 for these results. It can be seen from these data that the performance difference between the local policy and non-recursive global policy is marginal. The local policy suffers in performance due to the delay (discussed in Section 6.1) in making the sender state information globally available. This delay becomes less critical at high system loads.

The performance difference between the two global policies is marginal at low to medium system loads. At high system loads, the recursive global policy provides better performance. The performance difference between the two global policies increases with system load. The difference is a function of the reinitiation period. As explained in Section 6.1, in case of false sender, the non-recursive version waits for the corresponding reinitiation period before initiating another request for load distribution. The recursive version, on the other hand, continues searching the hierarchy. Thus, the recursive version provides better performance than the non-recursive version at high system loads. At low system loads, however, the recursive version provides marginally worse performance than the non-recursive policy. In Figure 6.2, the difference is not noticeable at low loads but evident at system load of 50%. The reason for this is that the recursive search is useless at low to moderate system loads as the probability of a sender node existing is low.

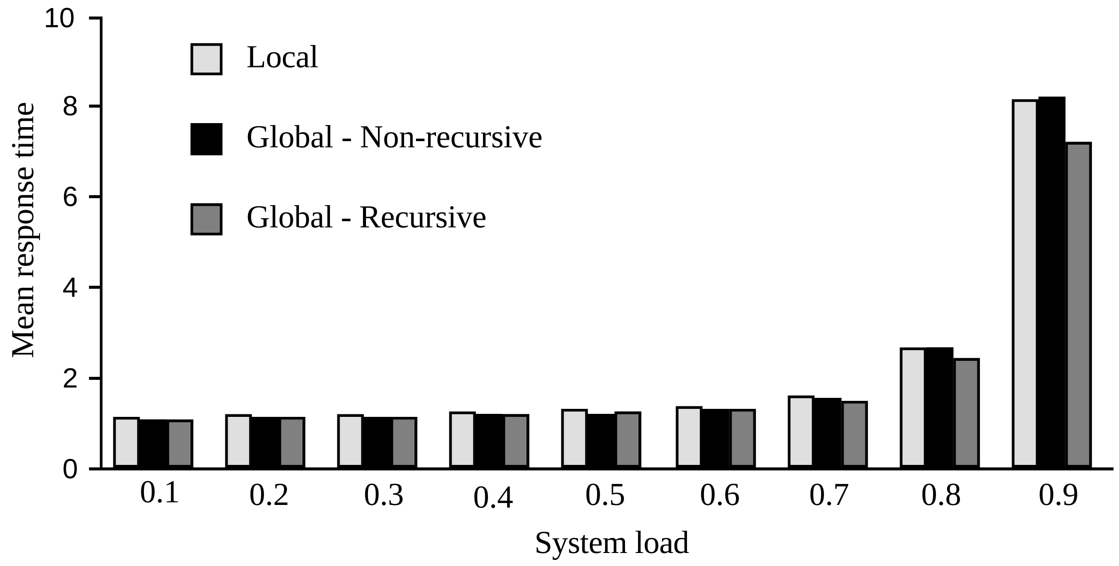


Figure 6.2 Performance of the three hierarchical policies as a function of the offered system load (N = 32 nodes, μ = 1, Cs = 4, λ is varied to vary system load, Ca = 4, T = 1, B = 4, transfer cost = 1%)

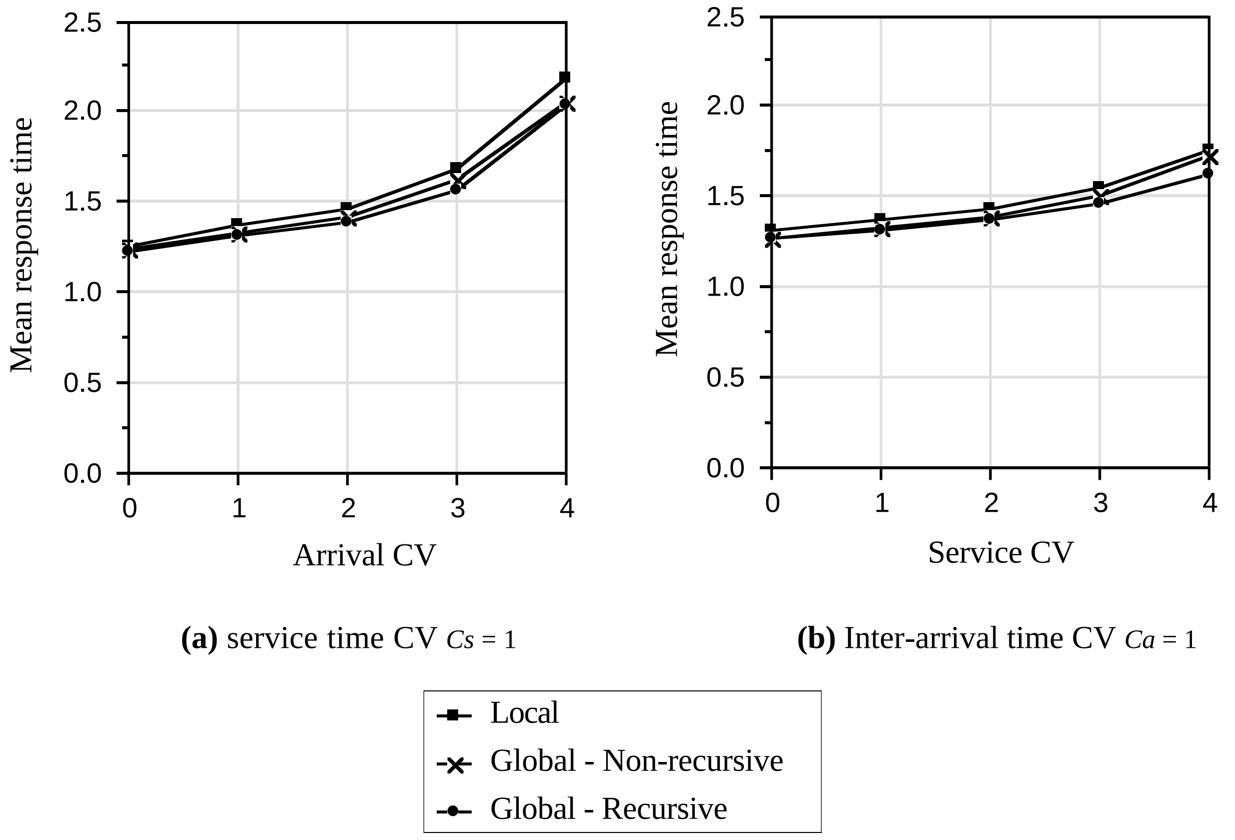


Figure 6.3 Performance of the three hierarchical policies to variance in inter-arrival and service times (N = 32 nodes, λ = 0.8, μ = 1, T = 1, B = 4, transfer cost = 1%)

Although we have not shown here, when the service time CV and inter-arrival CV are both 1, the performance difference among the three polices is even smaller than that shown in Figure 6.2. The impact of service time CV and inter-arrival time CV is shown in Figures 6.3. As shown in Figure 6.3, the performance difference among the three policies increases with the variance in both service times and inter-arrival times. All three policies show more sensitivity to inter-arrival CV than the variance in service times for reasons discussed in Section 5.2. The performance difference between the two global policies is noticeable only at high variance in service and inter-arrival times.

6.3. Sensitivity to Branching Factor

Performance of the hierarchical policies is sensitive to the branching factor of the hierarchy. All results presented thus far have been obtained using a branching factor of 4. This section discusses the sensitivity of the hierarchical policies to this design parameter. Figure 6.4 presents the result for various branching factors for a system consisting of 32 nodes (with an offered system load of 80%). As in the last section, the CV values of service time and inter-arrival times are both fixed at 4.

As can be expected, the performance difference among the three policies reduces with increasing branching factor value. In particular, when the branching factor is 32, the behaviour of the hierarchical policy is similar to that of the single coordinator policy. Notice that the recursive global policy at branching factor of 32 provides marginally worse performance. Even though there is only a single tree node in this case as in the single coordinator policy, due to the recursive search strategy used by the recursive global policy, there is more overhead. Since the probability of the hierarchy not reflecting the up-to-date system state is small, such recursive search is fruitless. However, when the branching factor is small (e.g., 2), recursive policy provides substantial improvement in performance compared to the other two policies (as the probability finding out-of-date state information in the hierarchy discussed before will be high).

6.4. Reduction in Message Handling Rate

One of the motivations for proposing the hierarchical policy is that the coordinator could potentially cause bottleneck problems. (Fault tolerance is another issue that is discussed in the next section.) In this section we demonstrate that this bottleneck problem is reduced by using the hierarchy. To show this we plot the average message handling rate of the tree nodes at each level. This rate is defined as follows:

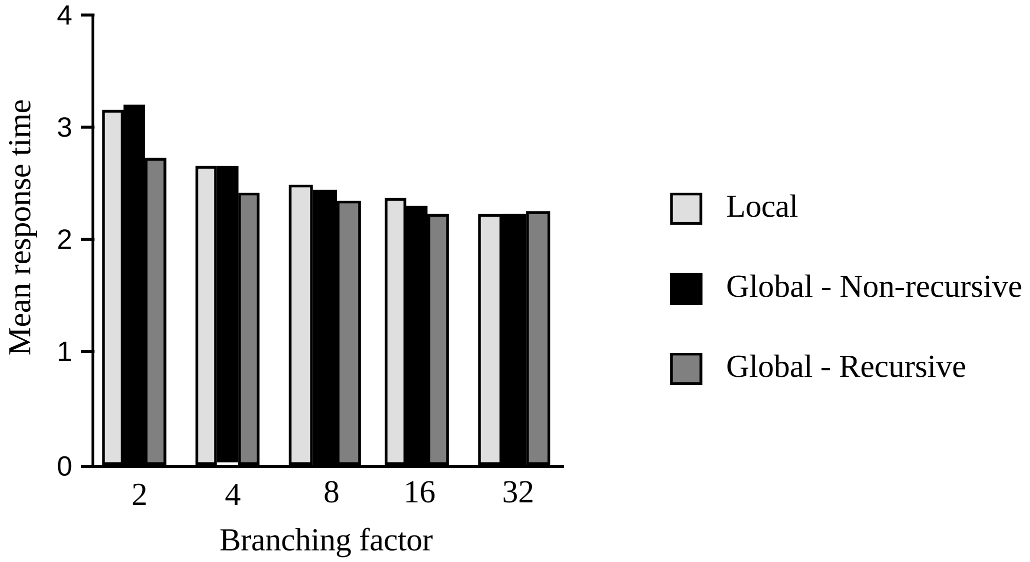


Figure 6.4 Sensitivity of the three hierarchical policies to the branching factor

(N = 32 nodes, λ = 0.8, Ca = 4, μ = 1, Cs = 4, T = 1, transfer cost = 1%)

average message handling rate of a level *i* node =

$$\frac{\text{number of mesages received + number of messages sent out by all nodes at level } i}{\text{number of tree nodes at level } i \text{ * number of jobs completed}}$$

Figure 6.5 shows the message handling rate when the CV values of service times and inter-arrival times are both 1 and the system load is 80%. The results are for a branching factor of 4. For reference, the message handling rate of the single coordinator policy is shown at each level. Of course, only the root node experiences this rate in the single coordinator policy. The rate is the highest at the middle level. At this level the hierarchical policies reduce the message rate by a factor between 2 and 2.5 (with the local policy providing the most reduction in message handling rate). The message handling rate of leaf nodes is low (between 15% and 18% of the single coordinator rate) because there are a large number of them. For our example, there are 8 leaf level tree nodes whereas only two middle level nodes. The root node in the hierarchical policies handles only about 25% to 35% of the single coordinator rate. Similar trends have been observed with other parameter values (details can be found in [Lo96]).

7. DISCUSSION

The results presented in Section 6 indicate that the local policy performs the worst (albeit marginally) and the recursive global policy the best. These conclusions should be viewed against the system model used in the simulation experiments. The nodes of the system are assumed to be interconnected by an Ethernet-like LAN. Therefore, global policies do not suffer from the additional message overhead. However, when we are considering a large system with hierarchical communication network (e.g., several LAN clusters interconnected by a WAN), the local policy proves to be advantageous as it tries to promote load sharing locally before widening the sphere of load sharing to include other clusters. Thus, the local policy tends to minimize the expensive inter-cluster communication. In this situation, the local policy might perform better than the global policy. The improvement depends on the severeness of the penalty in using the inter-cluster communication. Also note that the single coordinator scheme also suffers in this case. This scenario, however, is not reflected in the simulation model.

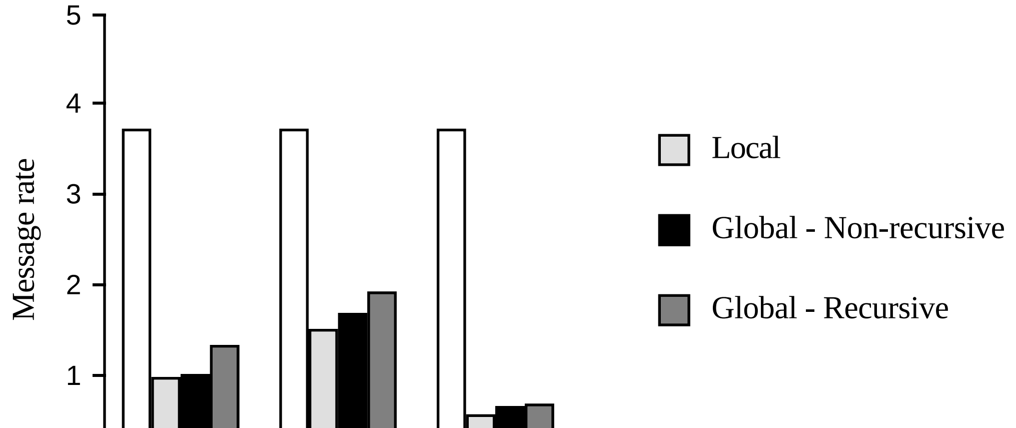


Figure 6.5 Message handling rates of nodes at the three levels of the hierarchy (for comparison, the message handling rate of the single coordinator policy (shown in white) is included for each level)

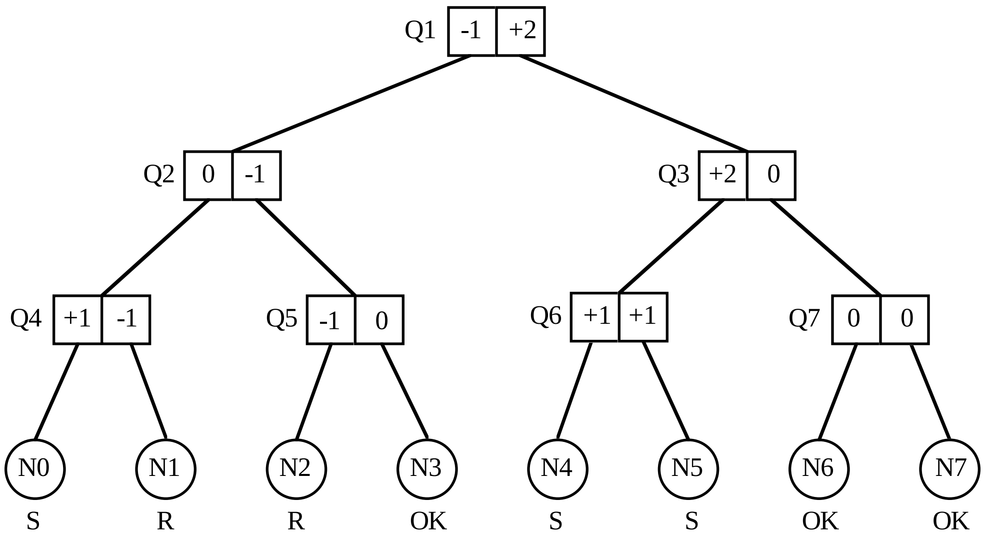


Figure 7.1 An example local hierarchical organization (local-v2) for an 8-node system with a branching factor of 2

Fault-tolerance is important in large distributed systems. The single coordinator policy does not provide fault-tolerance as the failure of the coordinator leaves the system in no-load sharing mode. The distributed policies provide excellent fault-tolerance as the failure of a single node only eliminates that particular node from further consideration in making load distribution decisions. The hierarchical policy provides fault-tolerance that is intermediate between these two extremes. If a single node fails, the corresponding tree node state information is lost. However, by keeping suitable pointers, nodes can successfully skip the failed node and operate in a degraded load sharing mode. The degree of degradation depends on the level of the tree node failure in the hierarchy. For example, if tree node Q2 fails (assume that Q2 is mapped to node N3), nodes N0 and N1 can still perform local load sharing using Q4. Similarly, even though it is not applicable for branching factor of 2, for larger branching factors, second cluster of nodes can also achieve local load sharing. However, the first two clusters (cluster consisting of N0/N2 and N3/N4) could not load share locally due to the failure of node Q2. But, for example, if Q4 and Q5 maintain a pointer to grandparent Q1, load sharing can be achieved with the rest of the system.

A variation of the global policy that we have not considered in this study works as follows: Each tree node in the hierarchy maintains individual node state information about part of the system for which the tree node is the root. Thus, in Figure 6.1, Q1 maintains state information for each individual node in the whole system much like in the single coordinator policy. Node Q2 maintains state information only on nodes N0 through N3 and node Q3 on nodes N4 through N7. The advantage of this scheme is that there is no need for downward movement of the load distribution message. The message at most goes to the root node in order to get the id of a sender node. However, this does not serve our purpose as every state change would have to update all tree node along the path to the root node. Thus, the root node gets the same number of update messages as in the single coordinator policy. It only cuts down the number of query messages for the id of a sender node. For this reason, we have not considered this scheme.

Finally, we briefly discuss a variation of the local hierarchical policy that maintains the true summary metric value in each tree node. This is shown in Figure 7.1 for the same node states as in Figure 3.1. Simulation experiments have shown that the average response time with this version is slightly higher than the version discussed in Section 3. It can also be seen that this scheme generates more state update messages than the other scheme (Figure 7.2).

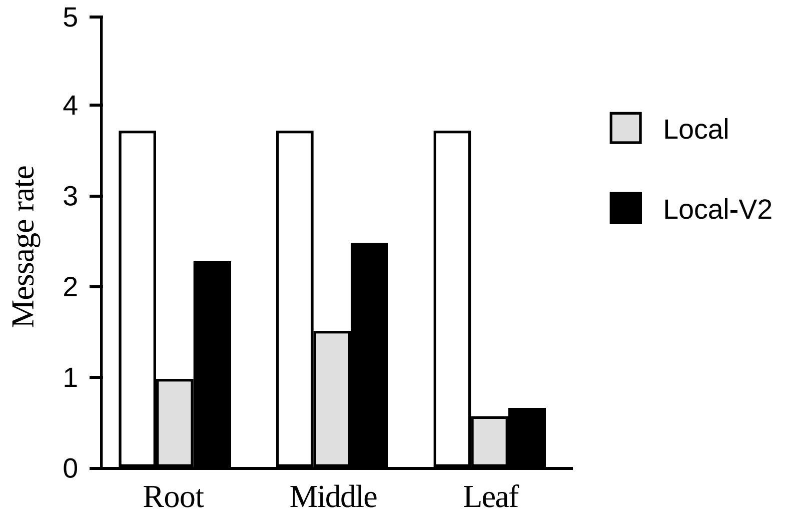


Figure 7.2 Message handling rates of the two local hierarchical policies (for comparison, the message handling rate of the single coordinator policy (shown in white) is included for each level)

8. CONCLUSIONS

Load sharing is a technique to improve the performance of distributed systems by distributing the system workload from heavily loaded nodes to lightly loaded nodes in the system. Dynamic load sharing policies take the system state into account in making load distribution decisions. The system state information can be collected in a distributed manner or by a single central controller node. In the distributed scheme, each node gathers the current system state information before making a decision on load distribution. Load sharing policies based on this strategy typically probe a few randomly selected nodes for their status in order to find a suitable partner for load distribution. Two principal policies that have been studied extensively in the literature are the sender-initiated and receiver-initiated policies.

In the centralized scheme, a central coordinator node is assigned the responsibility of collecting the system state information. Any other node that needs to distribute load consults the coordinator node for a suitable partner. The distributed policies do not perform as well as the centralized policy. Performance of distributed policies is sensitive to variance in job service times and inter-arrival times. The distributed polices, on the other hand, are scalable to large systems whereas the centralized policy causes bottleneck problems for large systems. Another problem with the centralized scheme is that if the coordinator fails, the system defaults to "no-load sharing" scenario.

We have proposed a hierarchical load sharing policy that minimizes the drawbacks associated with the distributed and centralized policies while retaining their advantages. We have presented a local hierarchical policy and two global hierarchical policies. Performance of these three hierarchical policies has been compared to the centralized single coordinator policy and the distributed sender-initiated and receiver-initiated policies. We have shown that the hierarchical policy provides substantial performance improvements over the sender-initiated and receiver-initiated policies; its performance is closer to that of the centralized policy while providing scalability and fault-tolerance closer to that of the distributed policies. Note that, due to the length of simulation experiments, we have restricted our experiments to a 32-node system. For this system size, the single coordinator does not experience the bottleneck problems.

We have not considered the impact of system and workload heterogeneity [Mir90]. System heterogeneity refers to non-homogeneous nodes (nodes with different processing speeds, for example) and the workload heterogeneity refers to non-homogeneous job characteristics. We intend to extend our work to study these issues in the near future.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support provided by the Natural Sciences and Engineering Research Council of Canada and Carleton University.

REFERENCES

[Alo88] R. Alonso and L. L. Cova, “Sharing Jobs Among Independently Owned Processors,” *IEEE Int. Conf. Dist. Computing Systems,* 1988, pp. 282-288.

[Cho90] S. Chowdhury, “The Greedy Load Sharing Algorithm,” *J. Parallel and Distributed Computing,* Vol. 9, 1990, pp. 93-99.

[Dan95] S. P. Dandamudi, “Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneous Distributed Systems,” *IEEE Int. Conf. Dist. Computing Systems,* 1995.

[Dik89] P. Dikshit, S. K. Tripathi, and P. Jalote, “SAHAYOG: A Test Bed for Evaluating Dynamic Load-Sharing Policies,” *Software - Practice and Experience,* Vol. 19, No. 5, May 1989, pp. 411-435.

[Eag86a] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Software Engng.*, Vol. SE-12, No. 5, May 1986, pp. 662-675.

[Eag86b] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Performance Evaluation,* Vol. 6, March 1986, pp. 53-68.

[Eag88] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing," *ACM Sigmetrics Conf.*, 1988, pp. 63-72.

[Hac87] A. Hac and X. Jin, “Dynamic Load Balancing in a Distributed System Using a Decentralized Algorithm,” *IEEE Int. Conf. Dist. Computing Systems,* 1987, pp. 170-177.

[Hac88] A. Hac and T. J. Johnson, “Dynamic Load Balancing Through Process and Read-Site Placement in a Distributed System,” *AT&T Technical Journal,* 1988, pp. 72-85.

[Har90] A. J. Harget and I. D. Johnson, “Load Balancing Algorithms in Loosely-Coupled Distributed Systems: A Survey,” in *Distributed Computing Systems,* (Ed) H. S. M. Zedan, Butterworths, London, 1990, pp. 85-108.

[Kob81] H. Kobayashi, *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology,* Addison-Wesley, Reading 1981.

[Kru88] P. Krueger and M. Livny, “A Comparison of Preemptive and Non-Preemptive Load Distributing,” *IEEE Int. Conf. Dist. Computing Systems,* 1988, pp. 123-130.

[Kru91] P. Krueger and R. Chawla, “The Stealth Distributed Scheduler,” *IEEE Int. Conf. Dist. Computing Systems,* 1991, pp. 336-343.

[Kun91] T. Kunz, “The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme,” *IEEE Trans. Software Engng.*, Vol. SE-17, No. 7, July 1991, pp. 725-730.

[Lel86] W. E. Leland and T. J. Ott, “Load Balancing Heuristics and Process Behavior,” *Proc. PERFORMANCE 86* and *ACM SIGMETRICS 86*, 1986, pp. 54-69.

[Lin92] H.-C. Lin and C. S. Raghavendra, “A Dynamic-Load Balancing Policy With a Centralized Dispatcher (LBC),” *IEEE Trans. Software Engng.*, Vol. SE-18, No. 2, February 1992, pp. 148-158.

[Lit88] M. J. Litzkow, M. Livny, and M. W. Mutka, “Condor - A Hunter of Idle Workstations,” *IEEE Int. Conf. Dist. Computing Systems,* 1988, pp. 104-111.

[Liv82] M. Livny and M. Melman, “Load Balancing in Homogeneous Broadcast Distributed Systems, *Proc. ACM Computer Network Performance Symp.*, 1982, pp. 47-55.

[Lo96] M. Lo, *Performance of Load Sharing Policies in Distributed Systems*, MCS Thesis, School of Computer Science, Carleton University, Ottawa, Canada 1996.

[Mir89] R. Mirchandaney, D. Towsley, J. A. Stankovic, “Analysis of the Effects of Delays on Load Sharing,” *IEEE Trans. Computers.*, Vol. 38, No. 11, November 1989, pp. 1513-1525.

[Mir90] R. Mirchandaney, D, Towsley, and J. A. Stankovic, “Adaptive Load Sharing in Heterogeneous Distributed Systems,” *J. Parallel and Distributed Computing,* Vol. 9, 1990, pp. 331-346.

[Mut87] M. Mutka and M. Livny, “Profiling Workstation’s Available Capacity for remote Execution,” *Proc Performance 87*, Brussels, Belgium, 1987, pp. 529-544.

[Nic87] D. Nichols, “Using Idle Workstations in a Shared Computing Environment,” Proc. ACM Symp. Operating System Principles, Austin, Teas, 1897, pp. 5-12.

[Rom91] G. Rommel, “The Probability of Load Balancing Success in a Homogeneous Network,” *IEEE Trans. Software Engng.*, Vol. SE-17, No. 9, September 1991, pp. 922-933.

[Sch91] M. Schaar, K. Efe, L. Delcambre, L. N. Bhuyan, “Load Balancing with Network Cooperation,” *IEEE Int. Conf. Dist. Computing Systems*, 1991, pp. 328-335.

[Shi90] N. G. Shivaratri and P. Krueger, “Two Adaptive Location Policies for Global Scheduling Algorithms,” *IEEE Int. Conf. Dist. Computing Systems*, 1990, pp. 502-509.

[Shi92] N. G. Shivaratri, P. Krueger, and M. Singhal, “Load Distributing for Locally Distributed Systems,” *IEEE Computer*, December 1992, pp. 33-44.

[Sue92] T. T. Y. Suen and J. S. K. Wong, “Efficient Task Migration Algorithm for Distributed Systems, *IEEE Trans. Parallel and Dist. Syst.*, Vol. 3, No. 4, July 1992, pp. 488-499.

[The88] M.M. Theimer and K. A. Lantz, “Finding Idle Machines in a Workstation-Based Distributed System,” *IEEE Int. Conf. Dist. Computing Systems*, 1988, pp. 112-122.

[Wan85] Y. T. Wang and R. J. T. Morris, “Load Sharing in Distributed Systems,” *IEEE Trans, Computers*, Vol. C-34, March 1985, pp. 204-217.

[Zho88] S. Zhou, “A Trace-Driven Simulation Study of Dynamic Load Balancing,” *IEEE Trans. Software Engng.*, Vol. SE-14, No. 9, September 1988, pp. 1327-1341.

[Zho93] S. Zhou, X. Zheng, J. Wang, and P. Delisle, “Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems,” *Software - Practice and Experience*, Vol. 23, No. 12, December 1993, pp. 1305-1336.