Appears in *Proc. Int. Conf. Parallel and Distributed Computing Systems,* Dijon, France, 1996.

Performance of Hierarchical Load Sharing in Heterogeneous Distributed Systems

Michael Lo and Sivarama P. Dandamudi
School of Computer Science, Carleton University
Ottawa, Ontario K1S 5B6, Canada

TECHNICAL REPORT TR-96-22

ABSTRACT – Performance of distributed systems can be improved by load sharing (i.e., distributing load from heavily loaded nodes to lightly loaded ones). Dynamic load sharing policies take system state into account in making job distribution decisions. The state information can be maintained in one of two basic ways: distributed or centralized. Two examples of distributed policies are the sender-initiated and receiver-initiated policies. While distribution of state information makes the distributed policies suitable for large distributed systems, they do suffer in performance. The centralized single coordinator policy is the best policy from the performance point of view in the absence of contention for the coordinator node. However, for large systems, the coordinator may become a bottleneck limiting the performance benefits of such a policy. In addition, the single coordinator causes fault-tolerance problems as the load distribution is dependent on this single coordinator node. Furthermore, in large hierarchically distributed networks (e.g., several LAN clusters connected by a WAN), consulting the central coordinator is expensive and leads to performance problems. The hierarchical policy minimizes these performance bottlenecks. In this paper, we compare the performance of the hierarchical load sharing policy with that of the two distributed policies and the centralized single coordinator policy in heterogeneous distributed systems. In order to see how close the hierarchical policy performs in comparison to the single coordinator policy, we have considered the scenario where the bottleneck problem does not exist in the centralized policy. We show that the hierarchical policy performs very similar to the single coordinator policy for all the various system and workload parameters considered in this study.

Key words: Load sharing, Dynamic load distribution, Sender-initiated policy, Receiver-initiated policy, Heterogeneous distributed systems.

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 heterogeneous distributed systems. As in [15], we consider two types of heterogeneous systems: type I and type II. In type I systems, all nodes are identical but the job arrival rate at different nodes can be different; in type II systems, processing rates of nodes are also different.

Two important components of a dynamic policy are a transfer policy and a location policy [4]. 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 [11,18,21]. 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 [11].

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 all 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 [19]. 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 [22].

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 [4,5]. 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 [1]. Distributed policies are, however, scalable to large system sizes.

To overcome these problems, we have proposed a hierarchical load sharing policy that combines the merits of the centralized and distributed policies while eliminating/minimizing the disadvantages of these policies [2]. The performance of the hierarchical policy in homogeneous distributed systems has been reported in [2]. In this paper, we focus on the performance of the hierarchical policy in *heterogeneous* distributed systems. Since distributed systems are often heterogeneous in nature and load sharing policies are sensitive to heterogeneity, it is important to study the performance in a heterogeneous system. 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 a distributed policy. Note that the centralized policy is the best policy if there is no contention/bottleneck problem for the coordinator.

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

This report can be obtained from http://www.scs.carleton.ca.

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 [5]. 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 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 [17,18].

In the remainder of this section, we briefly review related work and outline the paper. There have been several studies published on this topic (an excellent overview of the topic can be found in [18]) [1-7, 9-22]. Here, for the sake of brevity, we only review a subset of this literature that is directly relevant to our study. Eager et al. [4,5] provide an analytic study of sender-initiated or 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 [17] 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 [3] 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.

Mirchandani, Towsley and Stankovic [15] consider the impact of delay on the performance of heterogeneous distributed systems. They consider two types of heterogeneous systems: in type I systems external job arrival rates at nodes may differ and in type 2 systems the processing rates of the nodes may differ. They use analytical models to estimate the performance. In order to facilitate analytical modeling, they assume Poisson arrival process (i.e., inter-arrival times are exponentially distributed) and the service times are exponentially distributed. The node scheduling policy is first-come/first-served with no preemption. This work is directly relevant to our study. As in their study, we consider the two types of heterogeneous systems. However, we focus on the performance of the hierarchical load sharing policy that has been proposed recently [2].

The remainder of the paper is organized as follows. Section 2 discusses the three load sharing policies against which the performance of the hierarchical policy is compared. The 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 for type I and type II heterogeneous systems are discussed in Sections 5 and 6, respectively. Conclusions are given in Section 7.

2. LOAD SHARING POLICIES

This section gives a brief description of the three load sharing policies used to compare the performance of the 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 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 [6,9]. Load sharing facilities like Utopia [22] 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. It transfers the new job 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 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 [5,18], we assume that $T = 1$. That is, load distribution is attempted only when a node is idle (Livny and Melman [14] 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 [10,13,16,19]. 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) [18] or if there is a high variance in job inter-arrival times [1]. 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 is a long gap 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 a *receiver* state if the job queue length of the node is less than T_l (low threshold);
- it is in a *sender* state if the job queue length of the node is greater than T_h (high threshold);
- otherwise, it is in an 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 [1].

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 to 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 [19]. 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 [22].

Thus we would like to have a policy that provides performance very close that of the centralized policy while inheriting the merits of the distributed policies. The hierarchical load sharing policy has been proposed to achieve this objective. We will show that the hierarchical policy, described next, accomplishes this objective.

3.2. The 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 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. Also note that the summary metric for Q6 stored in Q3 is +1 rather than +2. This is because the value stored in the parent node is +1 if the sum is positive and –1 if it is zero or negative. 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 needs to be changed. Since N5 is still a sender, no state change is necessary for Q3. Dandamudi and Lo [2] discuss the impact of maintaining a true summary metric. In addition, they also discuss two global hierarchical policies.

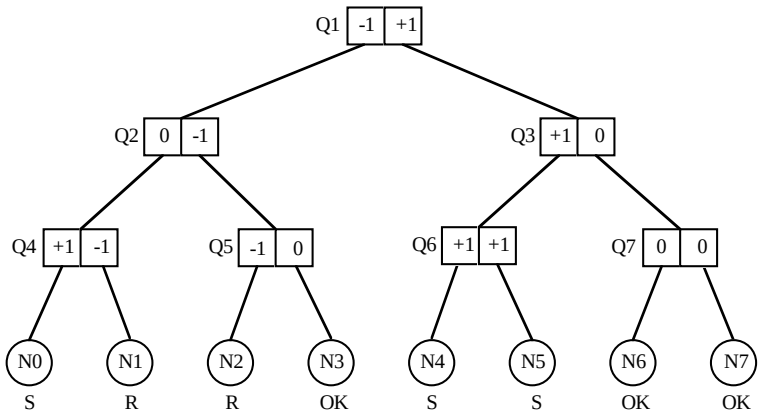


Figure 3.1 An example local hierarchical organization for an 8-node system with a branching factor of 2 (S = sender node, R = Receiver node, OK = OK node)

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.

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 is idle, it consults the hierarchy (explained next) for a sender node from which the receiver can get a job. The local hierarchical policy that has been 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 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. We 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 [15]. The CPU would give preemptive priority to communication activities (such as reading a message, 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 are 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 [3] uses serial probing.

The system workload is represented by four parameters. The job arrival process at each class i node is characterized by a mean inter-arrival time $1/\lambda_i$ and a coefficient of variation C_{ai}. Jobs are characterized by a processor service demand on a class i node (with mean $1/\mu_i$) and a coefficient of variation C_{si}. 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 use a two-stage hyperexponential model to generate service time and interarrival time CVs greater than 1 [8].

As in [15], we consider two types of heterogeneous systems. In type I systems, the nodes are homogeneous (i.e., have the same processing rate) but the job arrival rates at nodes are different. To reduce the complexity of the experiments, we consider two node classes as in [15]. In type II systems, the node processing rates are also different. We consider two node classes in type II systems as well. Nodes in each node class in type II system can have different job arrival rates and service rates.

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 of Type I Systems

This section presents the simulation results for type I heterogeneous systems. Unless otherwise stated, the following default parameter values are assumed. The distributed system has N = 32 nodes interconnected by a 10 megabit/second communication network. The number of class 1 nodes N_1 is 6 and the number of class 2 nodes N_2 is 26. Section 5.2 discusses the impact of node distribution between class 1 and class 2. The average job service time is one time unit (e.g., 1 second) for both classes. That is $\mu_1 = \mu_2 = 1$. 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. The threshold T_s (for the sender-initiated policies) is 2 and T_R (for the receiver-initiated policies) is 1 (as explained in Section 2.1). Similar threshold values are used in previous studies [3-5]. Probe limit P_l is 3 for the sender-initiated and receiver-initiated policies.

Batch strategy has been used to compute confidence intervals (at least 30 batch runs were used for the results reported here). This strategy 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 (in most cases, up to about 90%).

5.1. Performance as a function of system load

Figure 5.1 shows the mean response time as a function of offered system load. Note that the offered system load for class i is given by λ_i/μ_i. Since $\mu_1 = \mu_2 = 1$, offered system load on class i nodes is equal to λ_i. In this experiment we have fixed λ_1 at 0.9 and λ_2 is varied to see the performance impact of system load. The results in Figure 5.1a correspond to an inter-arrival CV (C_{ai}) and service time CV (C_{si}) of 1 (for both classes) and those in Figure 5.1b were obtained with the inter-arrival time and service time CVs 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 offered system load (on class 2 nodes) of up to about 80%. At higher loads, the receiver-initiated policy is better. This is a well-known result [5,18]. At low system loads, the sender-initiated policy is able to successfully transfer a job as the probability of finding a receiver node is high as there are 26 class 2 nodes.

At higher 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. For precisely the same reason, the

receiver-initiated policy provides a substantially better performance than the sender-initiated policy. However, its performance is still far below that of the hierarchical and single coordinator policies as shown in Figure 5.1a.

The hierarchical and single coordinator policies perform much better than the sender-initiated and receiver-initiated policies. This performance improvement increases with system load. Note that the minimum response time for this workload is 1 as that is the average job service demand used for these experiments. It can be seen from Figure 5.1a that both these policies provide performance very close to this value at low to moderate system loads. The main advantage of these two policies is that they can locate a sender node even if there is only one in the entire system as these policies maintain system state information on the entire system. The hierarchical policy performs marginally worse than the single coordinator policy as locating a sender involves sending more messages through the hierarchy. We should emphasize here that the single coordinator policy does not suffer contention problems. The rationale for this choice is that we would like to compare the performance of the hierarchical policy with that of the single coordinator policy, which provides the best performance in the absence of contention for the coordinator.

C_a = 4 and C_S = 4

As can be expected, higher service and inter-arrival CVs have a substantial impact on all policies. When C_{ai} = 4 and C_{Si} = 4 (for both classes), sender-initiated policies perform better at low system loads. The reason is that there are only 6 (out of 32) nodes operating at an offered system load of 0.9 while the remaining 26 nodes are operating at low system load. Since load sharing activity is initiated at most at $1/P_I$ = 1 rate when the nodes are idle, the 6 heavily loaded nodes do not get much help from the remaining 26 nodes. The following two factors (discussed in Sections 5.3 and 5.4) also contribute to this performance sensitivity: (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. Due to these interactions, the RI policy performs better than the SI policy for offered system loads on class 2 nodes higher than about 0.55.

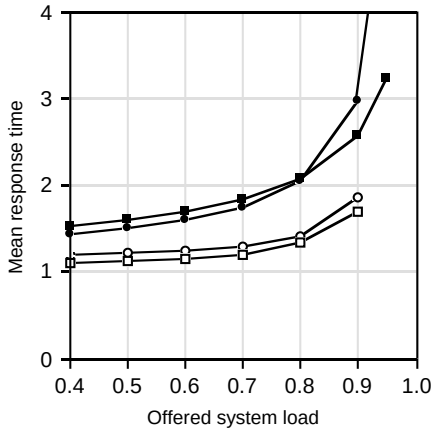
It may be noted from Figure 5.1b that the performance of the hierarchical policy is substantially better (particularly at moderate to high system loads) than both the sender-initiated and receiver-initiated policies. For example, the response time decreases from about 7 (with the receiver-initiated policy) to about 3 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 following sections discuss the sensitivity of the performance of the four load sharing policies to the degree of heterogeneity and variance in inter-arrival and service times.

5.2. Impact of degree of heterogeneity

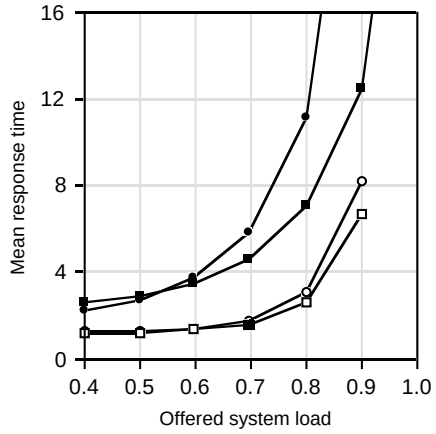
The results presented in the last section assumed that the class 1 nodes N_1 = 6 and class 2 nodes N_2 = 26, where the 6 class 1 nodes are heavily loaded (offered load of 90%). This section considers the impact of degree of heterogeneity on the performance. In this experiment, the offered system loads for class 1 and 2 are fixed as 0.1 and 0.9 (i.e., λ_1 = 0.1 and λ_2 = 0.9). Figure 5.2 shows the results. In this graph, the x-axis gives the number of class 2 nodes N_2 and remaining nodes belong to class 1 group. Since class 1 nodes are lightly loaded, the mean response time increases as we increase N_2.

When the number of class 2 nodes N_2 is smaller, SI performs better than RI because class 1 nodes are lightly loaded and the probability of finding a receiver is high in this case. As N_2 increases, the probability of finding a sender increases. As a result, the RI performs better and the performance of SI deteriorates rapidly at high N_2 values.

The hierarchical policy exhibits a more robust behaviour as N_2 increases. In particular, its performance is substantially better than both the SI and RI policies. The performance of the hierarchical policy is very close to that of the centralized policy. The difference between these two policies increases with N_2 (particularly noticeable at N_2 = 31). This is because, as N_2 increases, the system moves to a higher system load state as the number of class 2 nodes are at 90% offered system load. At these system loads, the additional delay associated with querying the hierarchy for a sender node increases as there are fewer sender nodes in the system. Thus, such queries traverse more of the hierarchy causing the delay to increase.



(a) Inter-arrival CV *C*ai = Service CV *C*si = 1 for both classes



(b) Inter-arrival CV *C*ai = Service CV *C*si = 4 for both classes

Figure 5.1 Performance sensitivity as a function of offered system load (N = 32 nodes, N_1 = 6, N_2 = 26, $\mu_1 = \mu_2$ = 1, λ_1 = 0.9 and λ_2 is varied to vary system load on class 2 nodes, B = 4, T_S = 2, T_R = 1, P_I = 3, transfer cost = 1%)

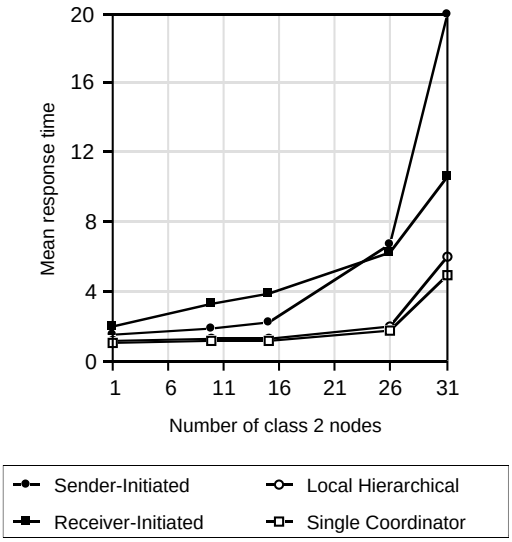


Figure 5.2 Performance sensitivity to degree of heterogeneity (N = 32 nodes, $N_1 = N - N_2$, N_2 is varied, $\mu_1 = \mu_2 = 1$, $C_{s1} = C_{s2} = 4$, $\lambda_1 = 0.1$, $\lambda_2 = 0.9$, $C_{a1} = C_{a2} = 4$, $B = 4$, $T_S = 2$, $T_R = 1$, $P_l = 3$, transfer cost = 1%)

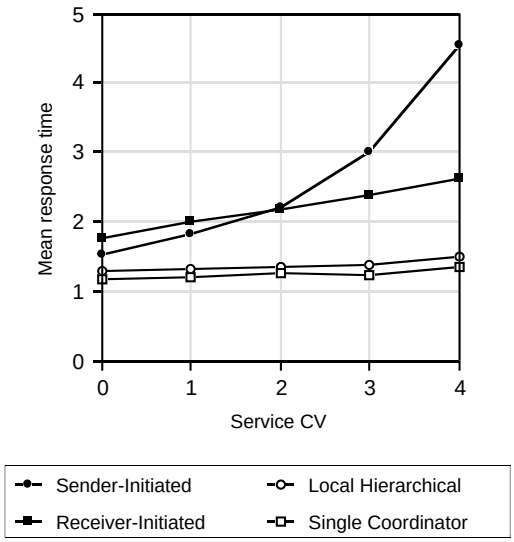


Figure 5.3 Performance sensitivity to service time CV (N = 32 nodes, $N_1 = 6$, $N_2 = 26$, $\lambda 1 = 0.1$, $\lambda 2 = 0.9$, $C_{a1} = C_{a2} = 1$, $\mu_1 = \mu_2 = 1$, $C_{s1} = C_{s2}$ = varied from 0 to 4, $B = 4$, $T_S = 2$, $T_R = 1$, $P_l = 3$, transfer cost = 1%)

5.3. Impact of variance in service times

This section considers the impact of service time CV Csi 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 [12] from over 9.5 million processes has been shown to have a coefficient of variation of 5.3 [9].

Figure 5.3 shows the impact of variance in service time. For this experiment, the inter-arrival time CV for both classes is fixed at 1. The sender-initiated policy exhibits more sensitivity to Csi than the receiver-initiated policy. At high Csi (for example, when Csi = 4) receiver-initiated policy (RI) performs substantially better than the sender-initiated policy (SI). The sensitivity of the FCFS policy to service time CV is an expected result because larger jobs can monopolize the processing power that results in increased number of probes in the SI policy. However, this increase in probing activity is useless because, at high system loads, the probability of finding a receiver node is small. At high Csi, RI performs substantially better than SI mainly because, at high system loads, the probability of finding an overloaded node is high. Thus, for the RI policy, increased probing activity results in increased job transfers (thereby increasing load sharing). This results in better performance and reduced sensitivity to Csi.

The hierarchical policy provides performance similar to that of the single coordinator policy. The advantage of the hierarchical and single coordinator policies is that they can successfully locate a sender node even if there is only one in the entire system as they both maintain the entire system state. As they successfully transfer jobs to lightly loaded nodes, these two policies exhibit far less sensitivity to service time variance.

5.4. Sensitivity to variance in inter-arrival times

The impact of inter-arrival CV C_{ai} is shown in Figure 5.4. For both classes, the service time CV is fixed at 1. The inter-arrival time CV is set to the same value for both classes and is shown on the x-axis. The receiver-initiated policies are relatively more sensitive to the inter-arrival CV than the sender-initiated policies. The reason for this sensitivity is that increased CV implies clustered nature of job arrivals into the system; this results in reduced probe rate. For example, assume that Csi = 0 for both classes. If Cai is also zero, each node invokes the probing process to locate a sender node after the completion of each job because local job arrival would be some time after the completion of the previous job (because of no variations in either service or inter-arrival times). Now increasing C_{ai} means clustered arrival of jobs. Suppose four jobs have arrived into the system as a cluster. Then, the node would not initiate any load sharing activity until it completes these four jobs. Although not shown here, probing activity decreases with increasing inter-arrival CV. On the other hand, clustered arrival of jobs fosters increased load sharing activity in sender-initiated policies. Thus, the sender-initiated policies

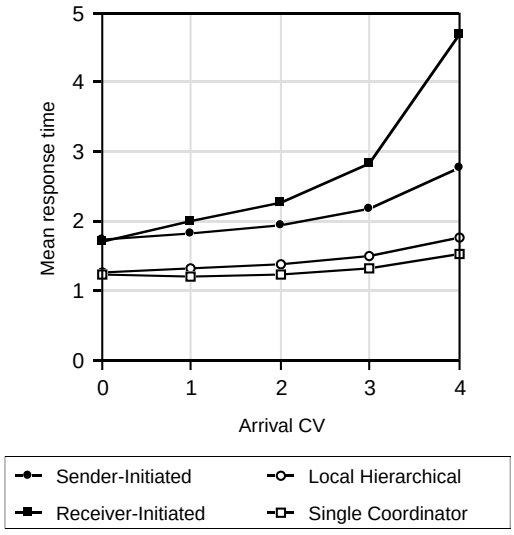


Figure 5.4 Performance sensitivity to inter-arrival time CV (N = 32 nodes, $N_1 = 6$, $N_2 = 26$, $\lambda 1 = 0.1$, $\lambda 2 = 0.9$, $C_{a1} = C_{a2}$ = varied from 0 to 4, $\mu_1 = \mu_2 = 1$, $C_{s1} = C_{s2} = 1$, $B = 4$, $T_S = 2$, $T_R = 1$, $P_l = 3$, transfer cost = 1%)

tend to transfer more jobs than the receiver-initiated policies and achieve better load sharing.

Both hierarchical and single coordinator policies, while providing much better performance than the SI and RI policies, exhibit more sensitivity to arrival CV than to service CV. The main reason for their sensitivity is that they both implement the receiver-initiated load distribution and suffer from the problem discussed in the last paragraph.

6. PERFORMANCE OF TYPE II SYSTEMS

In type II heterogeneous systems, nodes may have different processing rates in addition to seeing different job arrival rates as in type I systems. As in the type I systems, we consider a system with N = 32 nodes consisting of two node classes with processing rates of μ_1=0.5 and μ_2=1. Due to space limitations, we will only present a sample of the simulation results here.

6.1. Performance as a function of system load

This section presents the results as a function of offered system load. For this experiment, we have divided the 32 system nodes equally between the two node classes (i.e., $N_1 = N_2$ =16). The offered system load is maintained the same for both classes. That is, the arrival rate of class 1 is maintained at half of that for class 2 nodes because class 2 nodes are twice as fast (i.e., $\lambda_1 = \lambda_2/2$). We have run several experiments to determine the best threshold value for each class of nodes. In general, the threshold values T1 = 2 and T2 = 3 are found to be good choices. The results are presented in Figure 6.1. The inter-arrival and service time CVs for both classes are fixed at 4 (i.e., Cai = Csi = 4).

Both sender-initiated and receiver-initiated policies perform the same until the offered system load is about 0.6. Beyond that the receiver-initiated policy performs significantly better than the sender-initiated policy. The performance difference between SI and RI also depends on the number of class 1 and class 2 nodes.If there are more class 1 nodes (which are slow nodes in our case), the performance difference increases. This issue is discussed in the next section.

Analysis of simulation data for each class suggests that, in SI policy, more jobs are transferred from class 2 to slower class 1 nodes. Since the job arrival rate of class 2 nodes is twice that of the class 1 nodes and T_1 is 2 and T_2 is 3, the probability of finding a sender node in class 2 is higher than in class 1. Since the probing process is unbiased, more jobs are transferred to class 1 nodes. As a result, the response time of class 1 jobs is substantially higher than that of class 2 jobs.

In RI policy, job movement across the class boundary is more balanced than that in SI policy. This is because class 2 nodes have a higher probability of being in the sender state than class 1 nodes. As a result of this balanced job movement, the difference between the response times of the two classes is smaller than in SI policy.

As in the type I systems, the hierarchical policy provides a much better performance than the SI and RI policies. Its performance is close to that of the single coordinator policy up to about 0.7 offered system load. The difference between these two policies is noticeable only at high system loads for reasons given in Section 5.

6.2. Impact of degree of heterogeneity

In this section we will look at the impact of degree of heterogeneity on the system performance. Figure 6.2 shows the response time as a function of number of class 1 nodes N_1. Note that the number of class 2 nodes (i.e., faster nodes) N_2 is given by 32–N_1. Thus as we move from left to right in Figure 6.2, the number of slower nodes increases in the system. As a result all policies experience increase in response time. However, only the SI policy exhibits more sensitivity to N_1 than the other three policies. This is because the sender-initiated policy finds it difficult to locate a receiver as there are fewer receivers than senders (because the system load is 80%). As a result, the SI policy moves the system closer to no load sharing than the RI policy. This coupled with the fact that there are more and more slower nodes as we move right in Figure 6.2 causes the response time to increase much more quickly with SI than RI policy.

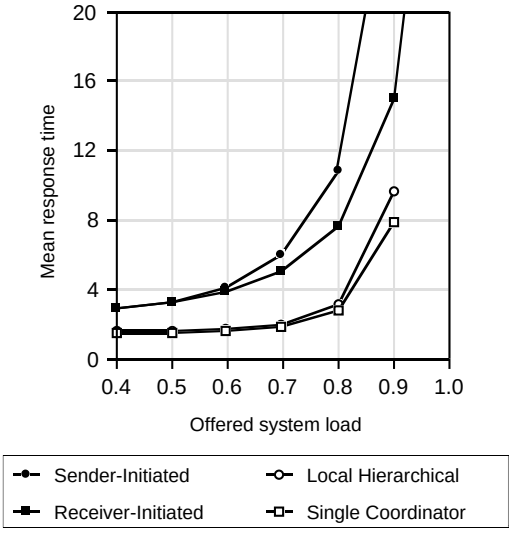


Figure 6.1 Performance sensitivity to offered system load (N = 32 nodes, $N_1 = N_2 = 16$, $\lambda 1 = \lambda 2/2$, $\lambda 2$ is varied, $C_{a1} = C_{a2}$ = 4, μ_1 = 0.5, μ_2 = 1, $C_{s1} = C_{s2}$ = 4, B = 4, T_{S1} = 2, T_{S2} = 3, $T_{R1} = T_{R2}$ = 1, P_l =3, transfer cost = 1%)

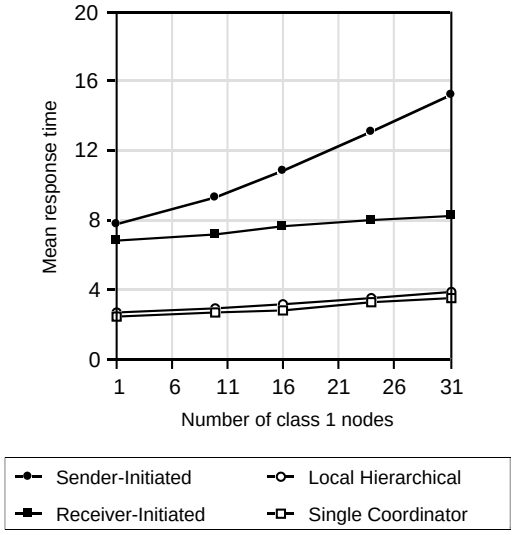


Figure 6.2 Performance sensitivity to the threshold parameter (N = 32 nodes, N_1 is varied, $N_2 = N - N_1$, $\lambda 1$ = 0.4, $\lambda 2$ = 0.8, $C_{a1} = C_{a2}$ = 4, μ_1 = 0.5, μ_2 = 1, $C_{s1} = C_{s2}$ = 4, B = 4, T_{S1} = 2, T_{S2} = 3, $T_{R1} = T_{R2}$ = 1, P_l =3, transfer cost = 1%)

The hierarchical policy exhibits a more robust behaviour and its performance is similar to that of the single coordinator policy. Again, this demonstrates that maintaining global system state information in the form of hierarchy is beneficial in improving the overall system performance.

7. 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. Previous studies have considered two dynamic load sharing policies: sender-initiated and receiver-initiated. In the sender-initiated policy, a heavily loaded node attempts to transfer work to a lightly loaded node and in the receiver-initiated policy a lightly loaded node attempts to get work from a heavily loaded node. These two are distributed policies as the system state information is distributed across the system. While such distribution of state information makes these policies suitable for large distributed systems, they do suffer in performance.

The centralized single coordinator policy is the best policy from the performance point of view in the absence of contention for the coordinator node. However, for large systems the coordinator may become a bottleneck limiting the performance benefits of such a policy. In addition, the single coordinator causes fault-tolerance problems as load sharing is dependent on this single coordinator node. Furthermore, in large hierarchically distributed networks (e.g., several LAN clusters connected by a WAN), consulting the central coordinator is expensive and leads to performance problems.

The hierarchical policy minimizes these performance problems. We have compared the performance of the hierarchical load sharing policy with that of the two distributed policies and the centralized single coordinator policy. In order to see how close the hierarchical policy performs in comparison to the single coordinator policy, we have considered the scenario where the bottleneck problem does not exist in the centralized policy. We have shown that the hierarchical policy performs very similar to the single coordinator policy (which provides the best performance in the absence of contention) for all the various system and workload parameters considered in this study.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] S. P. Dandamudi, “Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneous Distributed Systems,” *IEEE Int. Conf. Dist. Computing Systems*, 1995, pp. 484-492.

[2] S. P. Dandamudi and M. Lo, *Hierarchical Load Sharing Policies for Distributed Systems*, Technical Report SCS-96-1, School of Computer Science, Carleton University, Ottawa, Canada (This technical report can be obtained from http://www.scs.carleton.ca)

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

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

[11] 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.

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

[13] 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.

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

[15] 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.

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

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

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

[19] 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.

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

[21] 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.

[22] 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.