Computer Communications, 18(9):620–630, September 1995.

Scalable Approaches to Load Sharing in the Presense of Multicasting

Craig E. Wills
David Finkel

Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609

Abstract

This work examines policies based on multicasting for load sharing in a local area network environment. In these policies, lightly-loaded nodes join multicast groups to indicate their ability to accept additional work, and heavily-loaded nodes send multicast messages to locate these lightly-loaded nodes. Simulation is used to study the performance of these load sharing policies and compare them with previously proposed load sharing policies.

The results show that multicasting is an efficient method for locating lightly-loaded nodes, yielding better response time compared to previous policies. In addition, the results show that multicast-based policies can be used to lessen network traffic to busy nodes and nodes on remote LANs, while scaling to large numbers of machines.

KEYWORDS: load sharing, multicasting, distributed systems, local area networks

1 Introduction

As network environments grow to contain more machines, the possibility of transferring work from heavily-loaded to lightly-loaded machines is attractive. Studies have shown that in large networks there are many idle workstations at any time, even at peak usage periods in the day [1]. The process of redistributing workload in a network is the problem of *load sharing*, which has been examined in detail by much previous work. We previously examined approaches to load sharing based on multicasting in a small local area network environment [2]. This work specifically examines load sharing policies based on multicasting that can be scaled to networks with hundreds, even thousands of machines.

The use of multicasting is appealing because it supports efficient delivery of messages to a subset of machines, as opposed to broadcast, which delivers messages to all machines, or unicast, which delivers messages from one machine to another. Multicasting, which supports logical addresses, is used in our work to group lightly-loaded machines together in a single multicast address group. Requests to locate lightly-loaded machines can be multicast to this address rather than probing random machines, or broadcasting information between machines. There are three benefits to this approach. First, use of multicasting increases the chance that a lightly-loaded machine can be found efficiently even if the overall load of the network is high. Second, multicasting allows only lightly-loaded nodes to be recipients of load sharing messages. Most load sharing policies involve sending probe messages to potential transfer destinations, or broadcasting state information periodically. Either method interrupts both heavily- and lightly-loaded nodes. Because interrupt handling and context switching are relatively more expensive in modern architectures [3], avoiding interruptions to busy nodes is of even greater importance.

The third benefit is to limit traffic on shared resources such as backbone networks that interconnect individual local area networks (LANs). Although band-

width of the backbone network is a potential bottleneck, the ability of bridges to switch packets between a LAN and the backbone network is often a more serious problem. By using separate multicast addresses to represent lightly-loaded nodes on individual LANs, bridges can be configured to restrict multicast messages to the local LAN or to accept only multicast messages directed specifically to the LAN.

Our approach is to examine load sharing policies based on multicasting in such a way to make direct comparisons with other load sharing work. We examine how these policies work as we scale to networks of hundreds or thousands of machines using selective multicasting to a LAN. In addition to increasing the scale, we examine nonhomogeneous workloads and the effect of message costs on the policies. The policies are evaluated in terms of the overall task response time, the number of messages transmitted on the network and the number transmitted by each machine. We also examine the number of messages handled by busy machines, those not willing to accept transferred tasks, and the number of messages transmitted over the backbone network, which interconnects individual LANs.

In this paper we first describe related work done on load sharing and multicasting and then describe the system and policy model used in our analysis. The system model draws upon previous work by Eager, Lazowska, and Zahorjan [4], and our policy model is based on work by Shin and Chang [5]. After describing our model, we propose two load sharing policies that use multicasting along with variations of these policies for better scaling. These policies are compared with policies proposed in [4] and [5], along with another policy based on multicasting by Theimer and Lantz [6]. The policies are examined using simulation under varying load, scale, workload and message cost conditions. We conclude with a summary of our findings.

2 Related Work

An early work on load sharing in distributed systems [4] studied three abstract load sharing policies and concluded that load sharing has the potential to greatly improve system response time, and that moderately complex policies give almost as much performance improvement as more complex policies when compared to using no policy. This conclusion has been verified in a number of settings by other researchers [7, 8].

In particular, [8] examines a wide variety of load sharing policies. Key differences among these policies are how the load on a computer is measured and how the load information is exchanged among the nodes involved in the load sharing decisions. A paper by Shin and Chang [5] proposes a policy where state information is broadcast among a relatively small set of "buddy" nodes. These buddy nodes are then considered for transfer if one of the nodes is overloaded. Shivaratri and Krueger describe work on probe-based policies where information concerning which nodes are willing to receive task transfers and which are not accepting transfers is cached so that information is not lost [9]. They use this approach in investigating sender-initiated (in which receivers are found for senders), receiver-initiated (in which senders are found for receivers) and symmetric-initiated (in which both senders and receivers are found) load sharing policies.

Theimer and Lantz explored the use of multicasting for load sharing in the V-System [6]. They implemented and compared decentralized and centralized policies using multicasting. Our work examines a similar decentralized policy, but looks at a different centralized approach. We do not use a reserved central node, but allow the "leader" designation to move fluidly between nodes with less system disruption when the current leader node fails or becomes busy. Theimer and Lantz conjectured that their work would out-perform work by Eager, et al [4], but provided no comparison to it or other work and did not show expected response times. Our work compares multicast-based policies not only to each other, but

also to other load sharing policies. We also examine variations of the decentralized and centralized policies that exhibit better scaling characteristics.

Finding lightly-loaded machines for load sharing is an instance of the general problem of resource location, which has been approached using multicasting in other work. Cheriton and Mann [10] successfully combined caching along with multicasting in the study of a decentralized naming facility. They use a well-known multicast group that servers can dynamically join and leave.

In [11] we looked at an approach to map resource attributes to multicast addresses so that a machine joins a multicast address group only if it has the corresponding resource attribute. Servers on each machine dynamically join and leave multicast addresses as the resources on the machine change. Clients seeking information about a resource send a message to the multicast address corresponding to a resource attribute where, in the ideal case, a machine receives a request only when it has resources with the given attribute.

3 System Model

In investigating load sharing using multicasting, we use a system model based on that proposed in [4]. In that work the distributed environment is modeled as a collection of identical nodes that communicate via a local area broadcast network, such as Ethernet[1] [12]. Hardware support for multicasting has long been available on this type of network, but its support in higher level software has been limited. However, support for multicasting at the network layer for the standard Internet Protocol (IP) has been defined and implementations are now available [13]. A machine may selectively join and leave multicast groups by communicating with its network interface. The network interface knows which multicast addresses are currently in use and passes packets sent to these addresses to the kernel.

[1] Ethernet is a registered trademark of the Xerox Corporation.

As networks scale to larger sizes, the model of all hosts residing on a single LAN is not realistic. Rather, the approach commonly taken to scale networks to hundreds and thousands of machines is to interconnect LANs containing a few tens of machines to a backbone network using bridges as shown in Figure 1. This is the system configuration used in our work where we assume each LAN contains 20 machines. A distributed environment is scaled by increasing the number of LANs connected to the backbone network.

In terms of multicasting, we assume that a bridge can learn or be configured to recognize a few well-known multicast addresses associated with hosts on its respective LAN. For example, a request sent by a host on LAN-A to the well-known address for all lightly loaded machines on LAN-A will not be passed to the backbone network by the bridge for the LAN. A directed request sent by a host on LAN-A to lightly loaded machines on LAN-C will be passed to the backbone network by the bridge for LAN-A, but only the bridge for LAN-C will pass the request to its LAN.

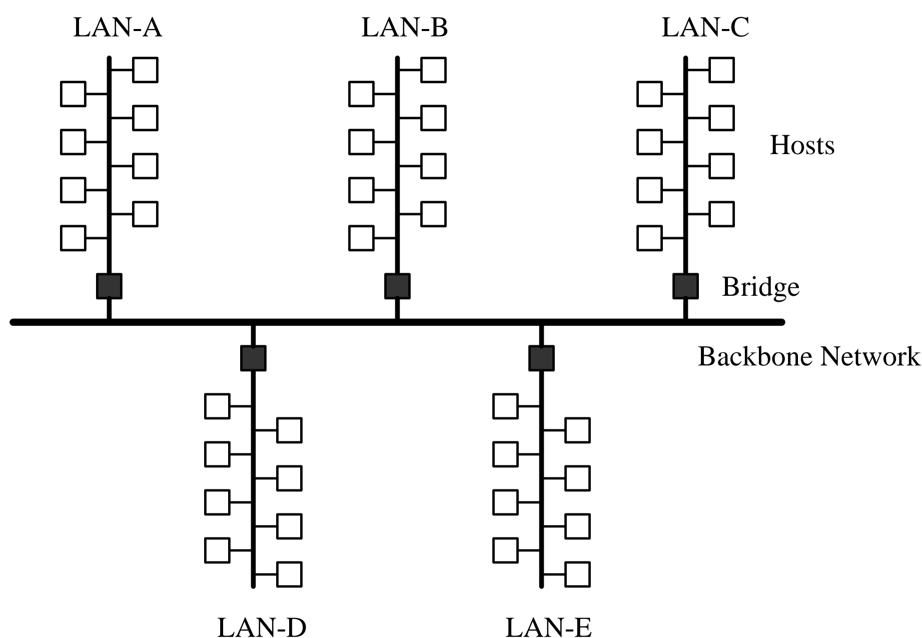

Figure 1: Network Model

In terms of node characteristics, we assume a base case where nodes are homogeneous with exponentially distributed task interarrival times with mean rate λ and exponentially distributed service times with mean S. As part of the study, we also examine the effect of having different mixes of nodes that are highly and lightly loaded. The average load at a node ($\rho = \lambda S$) is set to the arrival rate by fixing $S = 1$ for the analysis. All times given are in terms of the service time for a task. The process oriented simulation language CSIM[2] [14], which provides a convenient tool for modeling the system and collecting data, was used for the work. Simulation was used because it provides a a common platform to compare our work with other load sharing policies over a range of parameters to the model.

An important part of the work is not only to analyze the response time for different load sharing policies, but also to analyze the number of messages transmitted by each machine. Rather than simply count the number of "machine messages" (those messages sent and received by each machine) transmitted by each machine, we also count "busy machine" messages. A busy machine is one whose load is such that it is not accepting transferred tasks. We feel this is a more important measure of machine costs because the number of machine messages is only significant for machines doing useful work.

Similarly we are interested not only in the number of messages transmitted on any part of the network, but also in how many messages are transmitted on the backbone network. Thus we count "network messages" and "backbone network messages." The latter messages are important to minimize, as they must be processed by the bridges and handled by the shared backbone network.

The cost to transfer a task from one processor to another is represented as a processor cost rather than a communications cost as in [4]. This approach is based on the fact that the cost to pack and unpack data outweigh the communication costs. The transfer cost is represented in our analysis as a percentage of the

[2]CSIM is a copyright of Microelectronics and Computer Technology Corporation.

service time. For our analysis we use a value of 0.1, which, as pointed out in [4], is conservative, but is used for consistency with previous work. The transfer cost for a task is borne by the sending processor (versus the receiving processor) as a preemption of the currently executing task. Although processing is also done by the receiver nodes, it is less significant because these nodes are lightly loaded.

The cost of sending and receiving probe and state messages is not represented by processor costs in the base case for this analysis, but rather as the total number of messages transmitted. We feel this approach is appropriate because the messages themselves are small and processing costs are negligible. The important factor for message processing is the number of messages because the interrupt handling and context switching costs dominate actual processing costs. As part of the analysis we do consider the cost for sending and receiving messages using a processor cost of 0.005, which attempts to account for both processor and switching costs. We do not simulate contention on the network, but assume that all messages are delivered with negligible delay. As part of the analysis, we introduce a delay of 0.005 into messages transported across the backbone network to account not so much for network delay, but for processing and switching in the bridges.

4 Policy Model

All policies in this paper are described in terms of the three threshold parameters presented by Shin and Chang in [5]. They propose three threshold parameters TH_u, TH_f, and TH_v, which are used to represent the state of a node. They define four node states in terms of these thresholds and the queue length (QL), which includes the task in service, if any, at a node:

- underloaded if $QL \leq TH_u$,
- medium-loaded if $TH_u < QL \leq TH_f$,
- fully-loaded if $TH_f < QL \leq TH_v$, and

- overloaded if $QL > TH_v$.

All policies use TH_v as the transfer threshold—a node seeks to transfer a newly arriving task if the task would make QL at the node greater than TH_v. Note this threshold is used similarly as the threshold T defined in [4] except in that work a task is transferred if $QL \geq T$, but the QL does not account for the newly arrived task.

5 Multicast Policies for Load Sharing

In this section, we introduce two policies for load sharing that use multicasting. We also describe variations of these two policies that are designed to be more suitable for large scale environments. In the following section these policies will be compared with work done in [4], [5] and [6].

5.1 Multicast Threshold Policy

The simplest policy for using multicasting is to form a multicast address group in which any node that is underloaded ($QL \leq TH_u$) joins the group. We call this the "multicast threshold" (multithresh) policy. In this policy, state information is not expressed in messages exchanged among the nodes, but rather by nodes joining or leaving the multicast address group. The multithresh policy is essentially the same as the decentralized policy of Theimer and Lantz [6]

When the QL at a node grows to TH_f, the node removes itself from the multicast group. Overloaded nodes ($QL > TH_v$) wishing to transfer a job send a request to the multicast group and the first underloaded node that replies to the request is selected for transfer. Other replies are discarded by the requesting node. If no nodes are currently members of the underloaded multicast group, then the requesting node times out waiting for a response and then schedule the task on its local node.

The advantage of this approach is that it costs nothing in terms of network traffic to keep track of multicast group membership. In addition, a requesting node gets a fast reply to its request if any underloaded node exists in the network. There are two negative aspects to this approach. The first is that even though one reply is needed, the requesting node and the network will handle as many replies as underloaded nodes exist (although the node may start the transfer process as soon as the first reply arrives). The second is that if no underloaded nodes exist, then the requesting node must timeout waiting for a reply, which will not come; however, since this timeout occurs at a busy node, the node will be able to do other processing while waiting.

In comparing the multithresh policy for various threshold combinations we find the policy with the transfer threshold vector ($TH_u = 0$, $TH_f = 1$, and $TH_v = 1$, denoted as 011) works the best over low to medium system loads (details are given in [15]), while the threshold vector 012 works well over all loads. The 011 threshold vector does result in more busy machine messages because the threshold for defining a busy machine is lower. Overall, the best threshold vector is 012 because it gives good response time while exchanging fewer messages.

5.2 Scalable Multicast Threshold Policy

A potential problem with the multicast threshold policy is that as the number of machines increases the number of replies can grow large resulting in many wasted messages. As an alternative, we investigate a policy called multithresh20, which is so named because it uses the multithresh policy among the machines on a particular LAN. We assume that each LAN contains 20 machines.

This policy is identical to multithresh in terms of how nodes function except rather than one lightly-loaded multicast addresses there is a lightly-loaded multicast address for each LAN. Overloaded nodes wishing to transfer a task first send a request to the local LAN multicast group. If there is a reply then the first node

is chosen and the job transferred. However if no replies are received, the node randomly sends a request to the lightly-loaded multicast address of another LAN. Again the first reply is used if available otherwise the node randomly selects the lightly-loaded address of another LAN. This process continues until a node is found or a limit (defined as three in the study) is reached in which case the original node executes the task. This approach is analogous to probing used in [4], except with multicasting, an entire group of lightly-loaded nodes can be probed rather than one node at a time.

5.3 Multicast Leader Policy

The second policy we investigated is designed to reduce the number of replies that a requesting node receives. In this "multicast leader" policy (multileader), we designate one of the nodes in the underloaded group as its leader. This node is responsible for selecting one of the nodes in the underloaded multicast group in reply to a request by an overloaded node to transfer a task. This policy uses two multicast address groups—one for nodes that are underloaded and one for the group leader.

When a node becomes underloaded ($QL \leq TH_u$) it multicasts a join message to the underloaded group address and joins this group itself. When a node that was underloaded has its QL increase to TH_f then it sends a remove message to the underloaded group and leaves the group itself. The result of these actions is that at all times each underloaded node knows all members of the underloaded group joining after it. The first node to join the underloaded group is designated as the leader and joins the group leader multicast address. A node will remain group leader until its QL grows to TH_f. At that time, it picks the next oldest node in the underloaded group and sends that node a message designating it as the new group leader. The old group leader then removes itself from the group leader and underloaded multicast groups. Because each node in the underloaded group knows

the members joining after it, no other work is necessary.

If there are no more nodes in the underloaded group when the leader becomes overloaded, then the node remains as the group leader and as a member of the group, although based on its QL it should no longer be a member. This approach is used so that a group leader always exists (assuming no machine failures). As soon as an underloaded node joins the underloaded multicast group then the old leader node passes on leadership to the new node and removes itself from the group.

Overloaded nodes wishing to transfer a job send a request to the leader address for the group. The leader node randomly selects a node in the underloaded group (other than itself if possible) and returns this node to the requesting node. The leader node tries not to select itself for transfer, if possible, so that leadership changes are minimized. If there are no nodes in the underloaded group then the leader sends a negative reply and the requesting node must perform the task locally.

The advantage of the multileader policy over the multithresh policy is that an overloaded node receives one and only one reply to its request rather than a variable number. There is increased overhead for underloaded group membership changes, but these costs are incurred by nodes that are underloaded.

There is also increased complexity if we take into account machine failures. If the group leader node fails (or the system starts up and there is no leader) then a requesting node will not receive a reply to its request. In that case it must send a message to the underloaded group address and receive multiple replies. If underloaded nodes do exist then the node knowing about the largest number of underloaded nodes will designate itself as the new leader. If no nodes exist then the requestor must temporarily become leader, even though it is not underloaded, until an underloaded node becomes available. We expect machine failure to be rare in the network and we do not account for it in our analysis.

The 012 threshold vector values provide the best response time for the multileader over almost the whole range of system loads. Almost as good as the 012 vector is the 022 vector. The effect of the 022 threshold vector is that a node will

join the underloaded group when it is idle ($QL = 0$) and leave it when $QL = 2$. A node will attempt to transfer jobs away if $QL \geq 2$ when a new job arrives. The effect of having $TH_f = 2$ is to reduce the frequency of membership change in the underloaded group; this factor has a significant effect on reducing the number of messages required.

This effect is pronounced when examining the number of network and machine messages, while it is more subtle when examining the number of busy messages. The number of busy messages is small at low to medium load and not until the system load is high does the multicast group leader become busy.

Overall, the best threshold vector for the multileader policy is 022. This vector gives almost the best response while having a moderate number of machine and network messages.

5.4 Scalable Multicast Leader Policy

A potential problem with the multicast leader policy is that as the number of machines increases the number of state change update messages can grow exponentially. As an alternative, we investigate a policy called multileader20, which is so named because it uses the multileader policy among the machines on a particular LAN. Again, we assume that each LAN contains 20 machines.

This policy is identical to multileader, except the leader for a group may actually exist on another LAN. That is if there are no more lightly-loaded nodes on the local LAN then the leader in another LAN is given responsibility. Recall that the leader designation is associated with a multicast address and while it is desirable for a node on the local LAN to receive messages sent to this address, it is not required. If there are no lightly-loaded nodes on the local network then a leader in another LAN is found to be the surrogate leader for the local network. Surrogate leaders are found by probing leaders in other LANs to see if lightly-loaded nodes exist in that LAN. If so, leadership for the local LAN is transferred to the remote

LAN; thus it is possible for one node to simultaneously be the leader of multiple LAN groups. When a lightly-loaded node again becomes available on a LAN, the group leadership responsibility is returned to that node.

6 Performance Analysis

The most interesting part of this work is to compare how these multicast-based policies compare against other load sharing policies across various workloads and conditions. In this section the four policies we have described are compared with policies proposed in [4] and [5], which do not use multicasting, and the centralized policy proposed by [6], which does use multicasting. A brief description of these three policies is included. Simulation results presented have less than 7% error at the 95% confidence level.

6.1 Threshold Policy

The threshold policy works by checking the QL of a node when a task arrives [4]. If $QL <= TH_v$ the task is done locally. Otherwise a probe is unicast to another randomly selected node to find out the QL of that node. If $QL <= TH_v$ at that node, then the task is transferred. If the task is not transferred, then the process is repeated for up to a probe limit value. As in their work, we used a probe limit of 3 as the default. Results with other probe limits are given in [4]. We use the threshold vector of 022 in the comparison, although the values of TH_u and TH_f are ignored for this policy. As a comparison with their work, our results for the threshold policy are the same as those simulation results reported in [4].

An alternate threshold policy described in [9] was considered for comparison, but results for it are not shown. This policy explicitly keeps track of which nodes are eligible to receive transferred tasks, and which nodes have indicated they are not willing to accept transferred tasks. When these latter nodes change state

to accepting tasks they notify other nodes, who were previously informed of no acceptance, that transfers will be accepted. This alternate threshold policy does work better than the threshold policy used when $TH_v = 1$ and the set of eligible machines for transfer changes frequently, but at this threshold the number messages generated is larger. At the threshold of $TH_v = 2$, used in our comparison, there is little difference between the policies.

6.2 Buddy Set Policy

Another policy that we simulated is the buddy set policy of Shin and Chang [5]. In this policy, each node is a member of a buddy set and each node within the buddy set has an ordered list of nodes to try when looking for a node to accept a transferred task. This list is called a node's *preferred* list. All nodes in a buddy set maintain the state of other nodes in its buddy set by broadcasting state change information whenever they enter or leave the underloaded state. Thus, deciding where a task should be transferred is done locally at each node by using the state information and a node's preferred list. A node broadcasts its availability to accept transfers when $QL \leq TH_u$; it broadcasts its unwillingness to accept transfers when $QL \geq TH_f$. A node seeks to transfer a newly arriving job if the job would make the QL of the node greater than TH_v. Our multileader policy is similar to the buddy policy in that state changes are sent, but in the case of the buddy policy the state change is broadcast to all nodes in the buddy set.

In their work, Shin and Chang found that there is little difference between buddy set of sizes of 10 and 15, and virtually no difference in performance for larger buddy set sizes. Therefore, rather than use a single buddy set of size 20 in our study, we simulated the policy with buddy sets of size 10, each with its own preferred list. Each LAN of 20 nodes contains two buddy sets. We chose to use the threshold vector of 022 in the comparison because these give the best performance with a reasonable number of state changes.

6.3 Centralized Policy

The last policy we compared against was the centralized policy described by Theimer and Lantz [6]. In their work they actually examined two policies based on multicasting—decentralized and centralized. Our multithresh policy is equivalent to the decentralized policy. The centralized policy is similar to our multileader policy, but rather than the leadership for the lightly-loaded group rotating amongst nodes, this policy uses a fixed node as the leader. In our simulation we fix one of the nodes as the leader for this policy and all updates and requests are directed to this node. In the simulation no task arrivals are given to this node (its sole duty is to service updates and requests), but the arrival rate to other nodes is increased proportionally to account for this node that is not servicing tasks. The thresholds are used in the same manner as described in previous policies with the specific threshold vector of 022 chosen giving the best performance with a reasonable number of state changes.

6.4 Base Comparison of Policies

In the following we make a comparison of the seven policies we have discussed with a homogeneous workload, a fixed-size network of 100 nodes and no accounting for message costs other than counting their occurrence. We follow this comparison with results where each of these assumptions is selectively changed. In all cases the "best" set of threshold values were used for each policy.

Figure 2 shows a comparison of the mean response times for each policy. All the policies have nearly identical response times for light loads, but for system loads between 0.6 and 0.9 the two multithresh policies provide the best response times followed by the multileader and centralized policies. All of the multicast-based policies give better response time because at higher loads they have more nodes to consider in transferring a task. The multithresh policies are better because with a threshold vector of 012 they transfer to nodes that are less loaded than the

multicast leader policies with a threshold vector of 022. The non-multicast policies have higher response times because they have fewer nodes to consider in the load sharing decision. The threshold policy probes up to three nodes while the buddy set policy is limited to looking at its own buddy set.

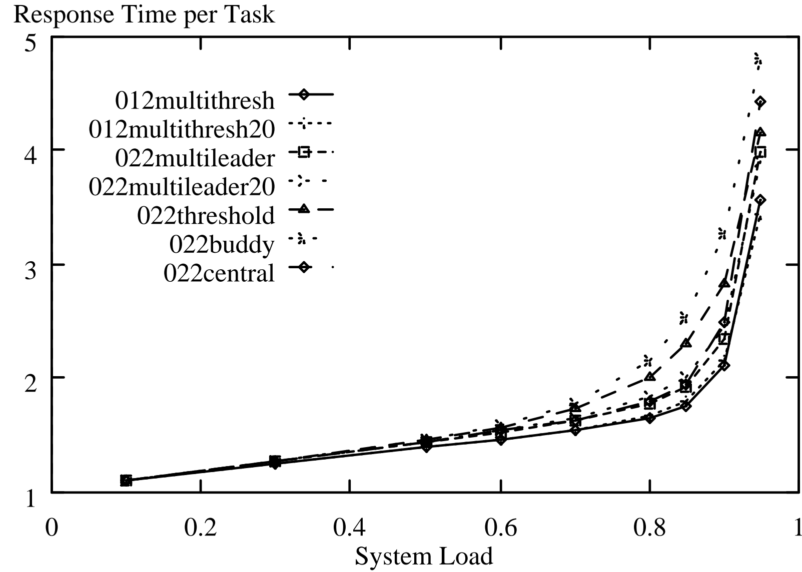

Figure 2: Response Time for Homogenous Workload and 100 Nodes (No Message Overhead)

Figure 3 shows a comparison of the network messages generated [3]. As shown, the multithresh policies generate the most messages at all low to medium loads because there are many requests to transfer a task with each request generating many replies. At high loads, all policies are comparable except for the buddy policy, which generates relatively few network messages. A more important measure is the number of messages sent over the backbone network. As shown in Figure 4, the straightforward multithresh policy generates a large number of backbone network messages, but the multithresh20 policy reduces this value to zero for low

[3]The graphs shown in this paper occasionally have lines clipped for policies that generate a large number of messages. Clipping occurs because we have chosen to use consistent scales across all graphs showing the same type of evaluation criteria. This approach emphasizes the display of useful information and allows it to be easily compared between graphs.

to medium loads and to a moderate level for high loads. This result is because most of the traffic is localized to each LAN. The two multileader policies show similar, if less pronounced, behavior with the multileader policy actually generating fewer backbone network messages at high loads. The centralized and threshold policies show a steady increase in backbone network traffic as the load increases while the buddy policy keeps all traffic local to a LAN.

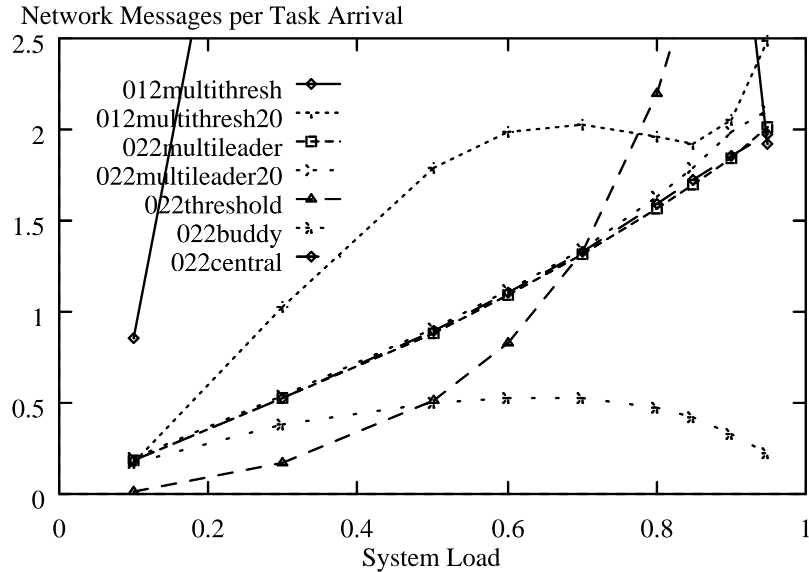

Figure 3: Network Messages for Homogeneous Workload and 100 Nodes

Also important is the number of machine messages, and especially the number of busy machine messages. Figure 5 shows the number of messages sent and received by machines. The multithresh and multileader policies, which do not localize their actions to a LAN, generate the largest number of machine messages, particularly at intermediate loads. Figure 6 shows the number of messages at busy machines; that is, at machines that are not accepting transfers. This measurement is significant because messages at idle nodes do not interfere with useful processing, while messages delivered to busy machines cause needless interruptions. The multithresh policy is the worst at low and medium loads because the requesting node, which is busy, receives many replies. At high loads, the threshold policy is

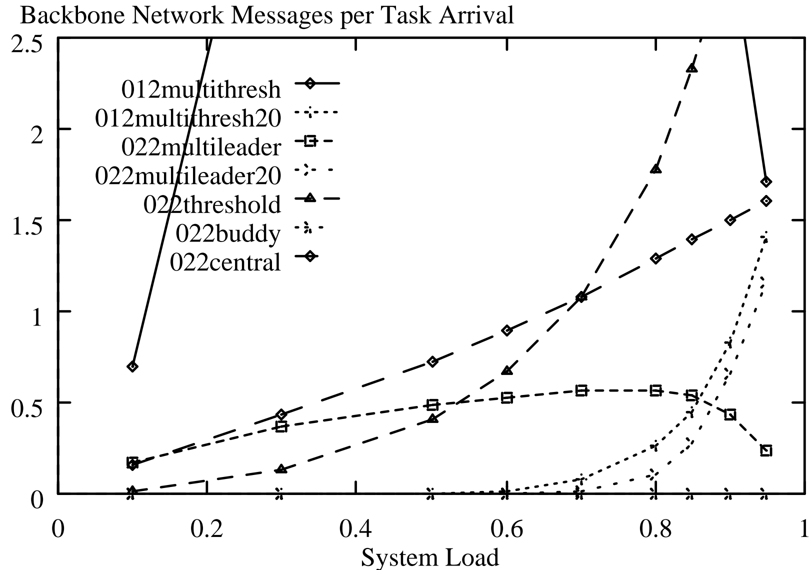

Figure 4: Backbone Network Messages for Homogeneous Workload and 100 Nodes

generating many probes that do not find willing destination nodes. However for this number of nodes, the number of busy messages does not appear to be significantly different for the various policies as was the case in [2] when only 20 nodes were used.

The significance of these comparative results is in the response times and number of backbone network messages with less significance in the number of busy machine messages. The multicast-based policies provide the best response times because they are able to interrogate all other machines (directly or indirectly) in making the transfer decision. The threshold policy selects a small subset, which works well under low loads at finding a transfer destination, but at higher loads only succeeds in generating many useless probes without finding a transfer node. The buddy set policy also limits the size of the buddy sets to reduce the number of nodes receiving broadcast information, but again decreases the chances of finding a transfer node.

Figure 3 shows significant differences among the number of network messages generated by each policy, but based on results of Ethernet capacity reported by

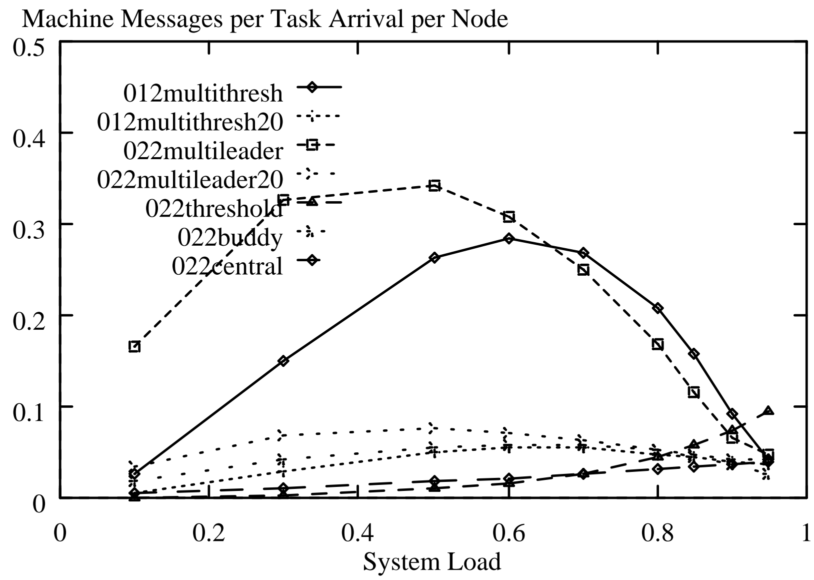

Figure 5: Machine Messages for Homogeneous Workload and 100 Nodes

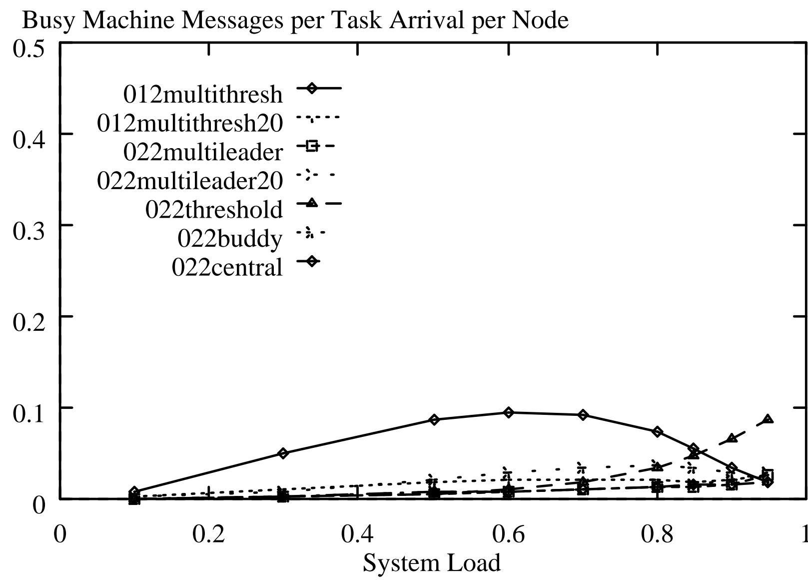

Figure 6: Busy Machine Messages for Homogeneous Workload and 100 Nodes

Boggs, Mogul, and Kent [16], these messages are a small fraction of network capacity. The more important problem is the number of messages that must be handled by bridges in switching them to and from the backbone network. In this case the two localized multicast-based policies work well in limiting this traffic. The number of machine and busy machine messages appears to show less variation between the policies.

6.5 Effect of Message Cost and Network Delay

To investigate the importance of busy machine messages and delay caused by sending messages over the backbone network, the next phase of our analysis was to repeat the previous scenario with a processor cost of 0.005 for sending/receiving messages and a delay of 0.005 for messages sent on the backbone network. Our experience indicates there is actually little cost for processing messages or delay because of lack of network bandwidth. Rather, the costs are due to switching—both by the processors to switch contexts to receive and multiplex messages and by bridges to switch messages from one network to another.

The response time for each of the policies over the various loads with the message overhead and network delay is shown in Figure 7. When compared with Figure 2, the multithresh policy shows comparatively worse performance at low and medium loads due to the large number of busy messages and of messages transferred across the backbone network. On the other hand, the multithresh20 policy shows the best performance at all loads because it is generally able to make good load sharing decisions and still keep traffic localized to non-busy machines on a single LAN. At high loads, the threshold policy shows comparatively worse behavior when the message overhead is taken into account.

The results for the other evaluation criteria are similar to those given in Figures 3–6 and are not shown for this analysis. Overall the results are as expected in that policies which transmit more network traffic to busy machines and switch more

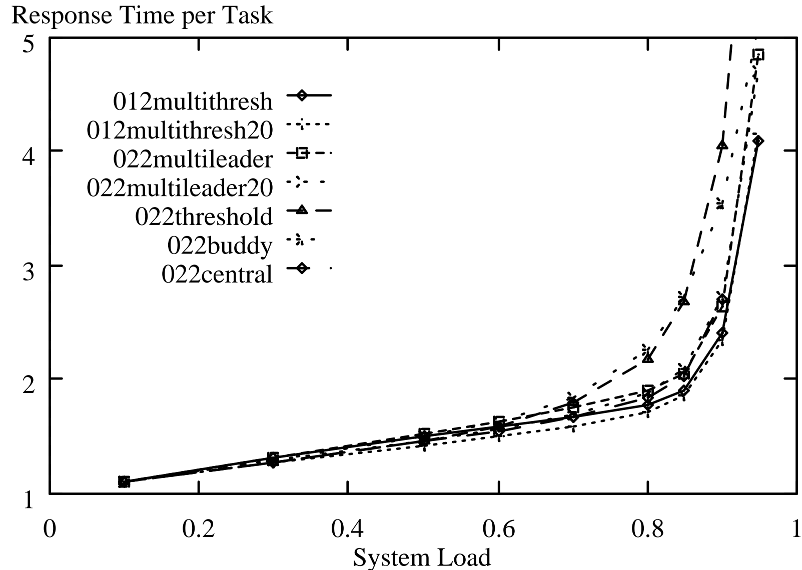

Figure 7: Response Time for Homogenous Workload and 100 Nodes (Message Cost and Delay of 0.005)

messages through bridges exhibit correspondingly worse behavior when costs are associated with these operations. It is important that these evaluation criteria be considered as we examine load sharing under different conditions.

6.6 Effect of Nonhomogeneous Workloads

We also investigated the policies using nonhomogeneous workloads with a fixed number number of nodes and no message overhead. Our approach was to define two types of nodes: lightly-loaded, having a load of 0.1 and heavily-loaded, having a load of 0.9. The policies were tested by varying the ratio of lightly- and heavily-loaded nodes in the system, which had the side-effect of varying the overall load of the network. Alternately we could have maintained a constant system load with different ratios by varying the load of the heavily-loaded nodes as done in [9]. This approach was considered, but it only served to exaggerate the effects of nonhomogeneity without changing the overall evaluation of the policies.

We did examine two approaches for how the ratio of nodes was varied to produce

a heterogeneous workload. In the first analysis, we specified the overall ratio of lightly- and heavily-loaded nodes and then applied this ratio to each LAN of nodes in the system. For example if the ratio was 20% lightly-loaded and 80% heavily-loaded then the nodes in each LAN would have this ratio. The relative results of this analysis are similar to those found for the homogeneous workload case and are thus not shown.

However our second type of heterogeneous workload did show interesting results. This approach is motivated by the observation that all nodes on a particular LAN may be lightly or heavily-loaded. For example a lab of machines interconnected by one LAN may be busy with activity while another lab of machines connected by another LAN may be relatively idle. Thus in the approach we varied the ratio of lightly/heavily-loaded nodes, but did so by giving all nodes on the same LAN the same load.

The response time results for the various policies over a range of load ratios are shown in Figure 8. The buddy policy exhibits poor performance because nodes on heavily-loaded LANs do not consider lightly-loaded nodes on other LANs. In addition, the multithresh20, and particularly the multileader20, policies exhibit slightly worse performance at low proportions of loaded nodes lower loads because these policies are not as efficient in locating lightly-loaded nodes in other LANs. By taking a global perspective, the other policies give slightly better performance at smaller ratios.

In Figure 9, the multithresh policy generates more network messages than can shown on this scale. enormous number of network messages in this scenario. The multithresh20 policy is relatively not as good at mixed ratios. Only the buddy policy shows a low number of messages at all proportions. The number of backbone network messages is mixed among the policies as shown in Figure 10. Again the multithresh policy exhibits poor performance (beyond the scale of the graph) with the multileader policies showing the best performance of the multicast-based policies.

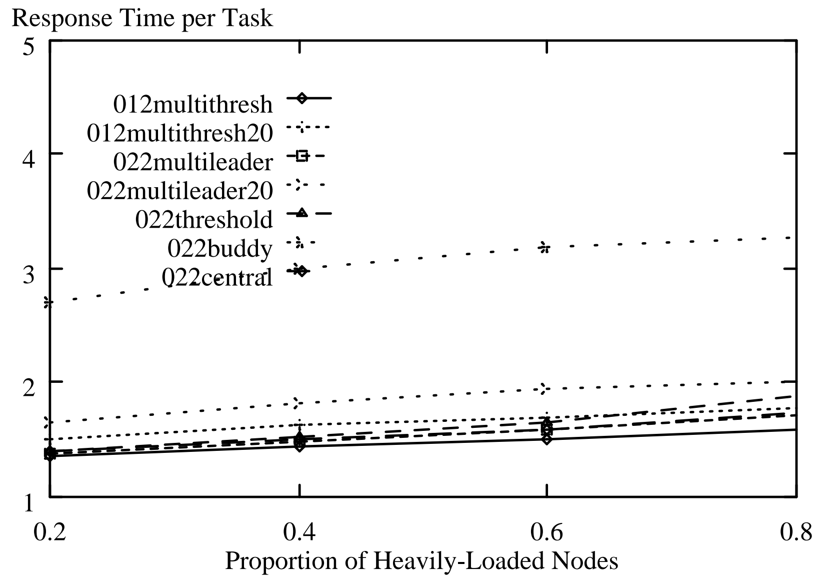

Figure 8: Response Time for Nonhomogenous Workload and 100 Nodes (No Message Overhead)

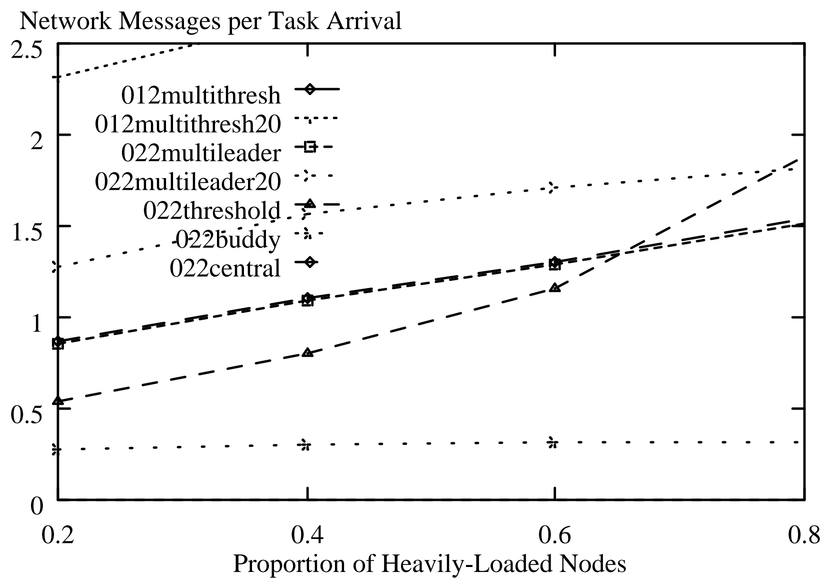

Figure 9: Network Messages for Nonhomogeneous Workload and 100 Nodes

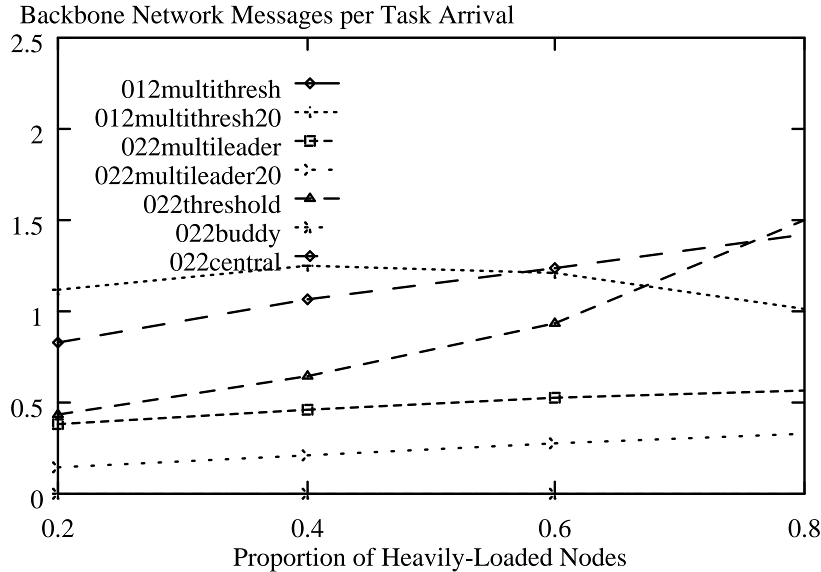

Figure 10: Backbone Network Messages for Nonhomogeneous Workload and 100 Nodes

Figures 11 and 12 show the number of machine and busy messages. The results are not particularly revealing other than to show that the multithresh and multi-leader policies generate the most machine messages with many of these messages to busy machines with the multithresh policy and low to medium proportions of heavily-loaded nodes. This latter result is because many lightly-loaded nodes exist and swamp a busy node wishing to transfer a task with replies.

The results of this analysis show that the multithresh20 policy does not offer better response in all cases and in fact provides slightly worse performance at low proportions of heavily-loaded nodes. In terms of messages the policy is comparable to the other policies. A slight modification in the policy could be made to better adjust when the workload is skewed. This approach would cause each node to not always request a lightly-loaded node in the local LAN as the first request, but rather to request a lightly-loaded node in the last LAN used for task transfer. This modification would still use nodes in the local LAN if they were lightly-loaded, but would more efficiently find other nodes if the local nodes were not accepting transfers.

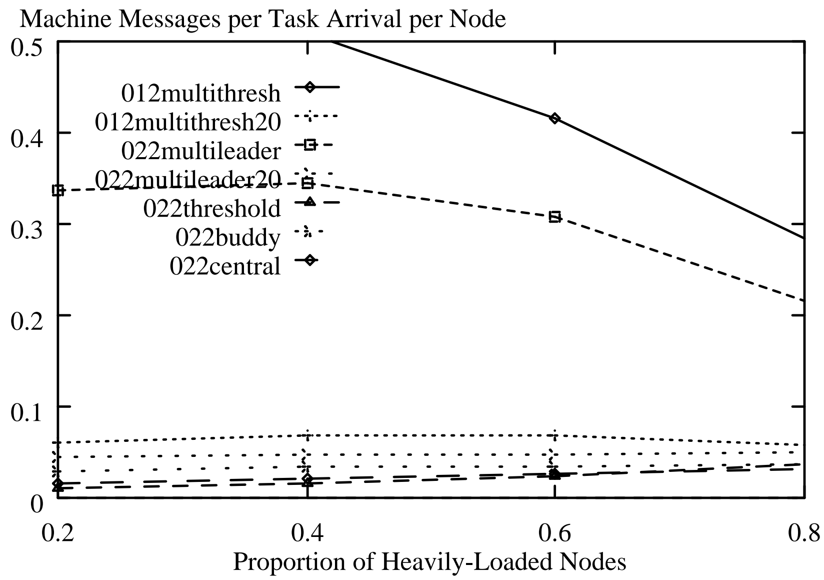

Figure 11: Machine Messages for Nonhomogeneous Workload and 100 Nodes

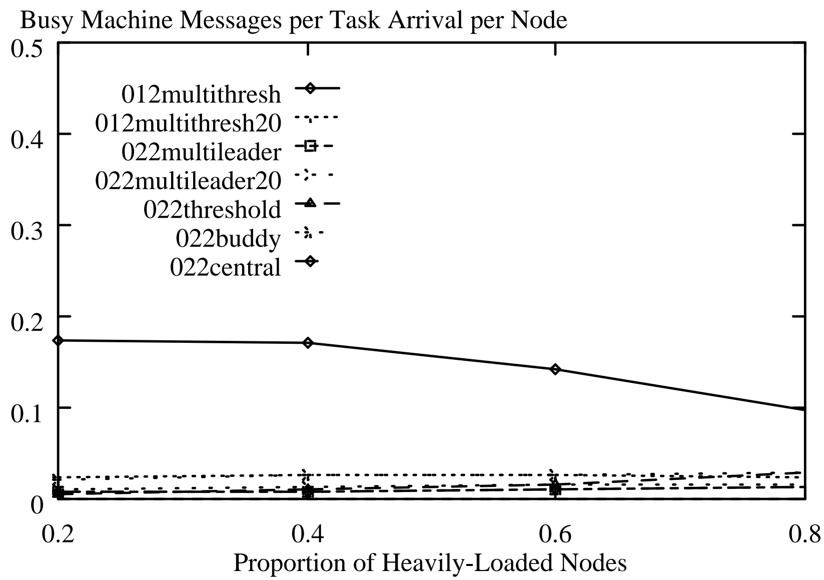

Figure 12: Busy Machine Messages for Nonhomogeneous Workload and 100 Nodes

6.7 Effect of Scale

The final analysis we performed was to test how well the various policies scaled to larger numbers of nodes. In this analysis a homogeneous workload with a load of 0.7 was used along with no message overhead. The number of nodes in the system was varied. Figure 13 shows essentially no variation in response time for the various policies. Thus, all the policies we considered scale equally well in that respect.

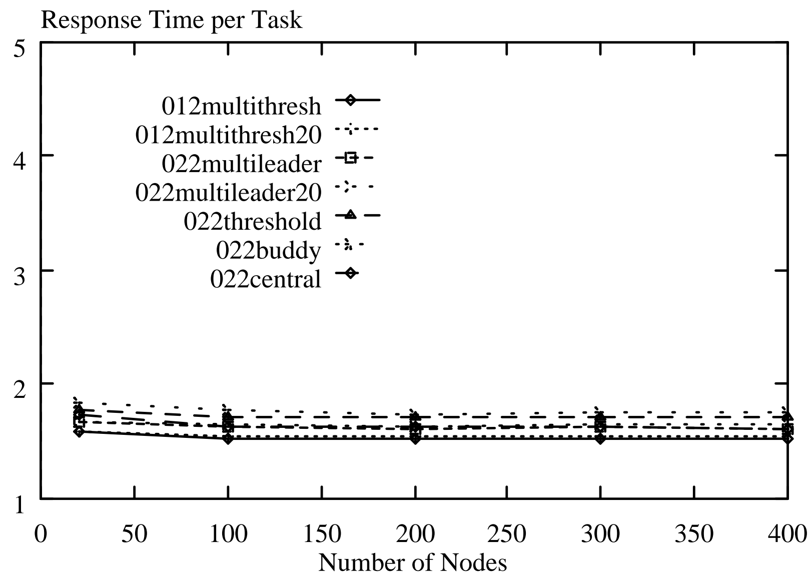

Figure 13: Response Time for Homogenous Workload of 0.7 and Variable Number of Nodes (No Message Overhead)

Figure 14 shows that there is little variation in the number of network messages among the policies except the multithresh policy, which does not scale well. Figure 15 shows the same lack of variation for backbone network messages, although the multithresh20 and multileader20 policies constantly exhibit the best performance.

The final criteria are shown in Figures 16 and 17. The multithresh and multileader policies exhibit consistently poorer performance in terms of machine messages and the multithresh policy shows worse performance for busy machine messages across different scales.

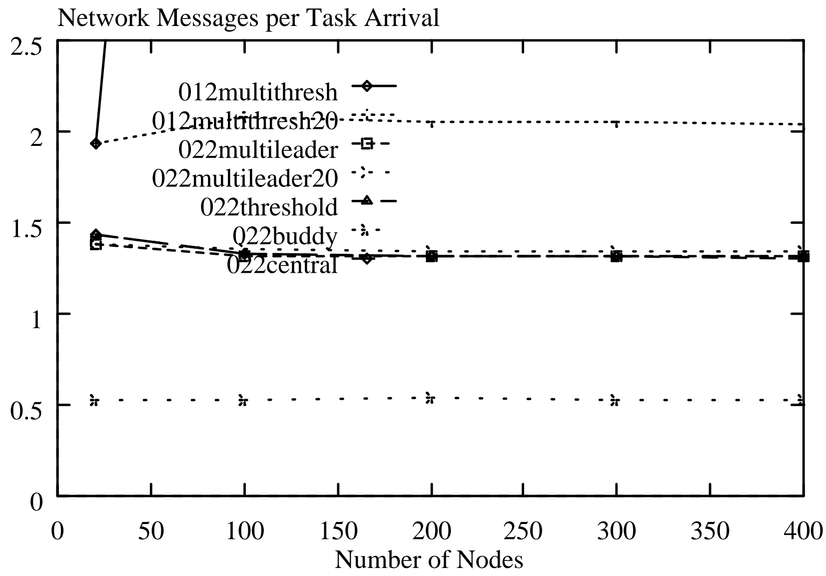

Figure 14: Network Messages for Homogeneous Workload of 0.7 and Variable Number of Nodes

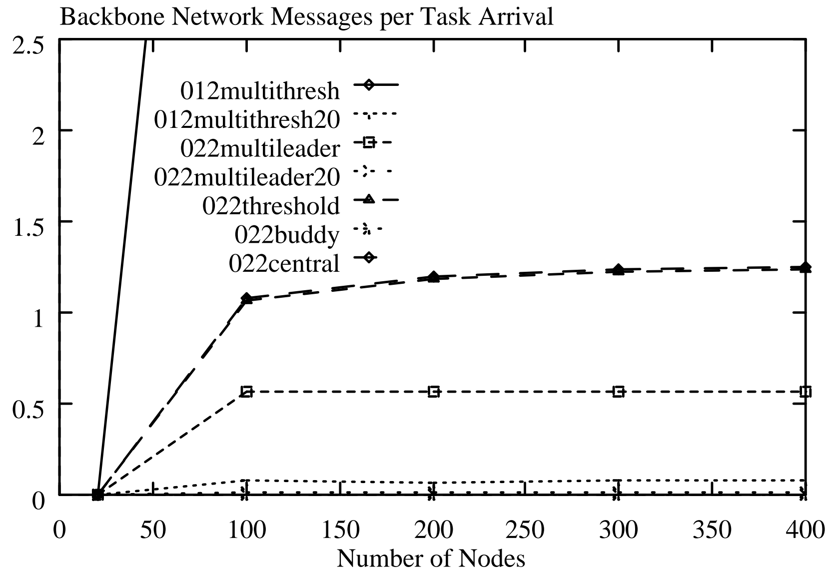

Figure 15: Backbone Network Messages for Homogeneous Workload of 0.7 and Variable Number of Nodes

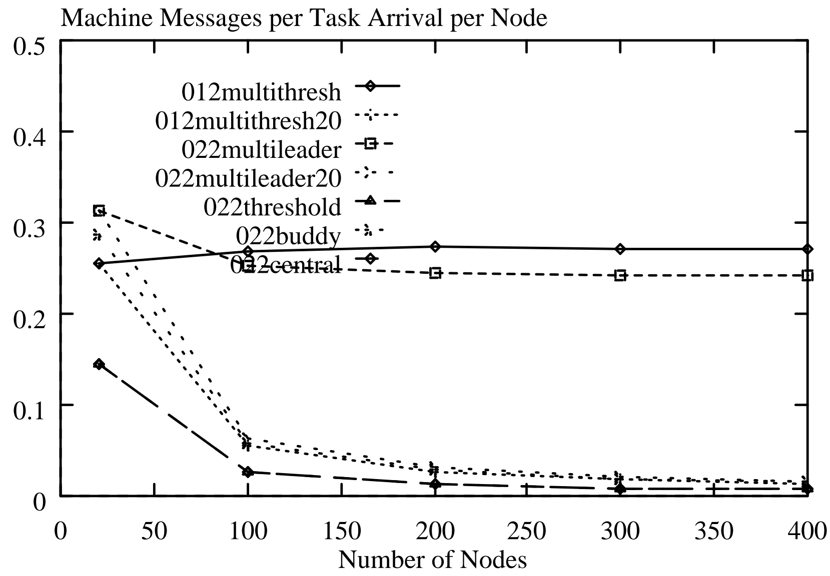

Figure 16: Machine Messages for Homogeneous Workload of 0.7 and Variable Number of Nodes

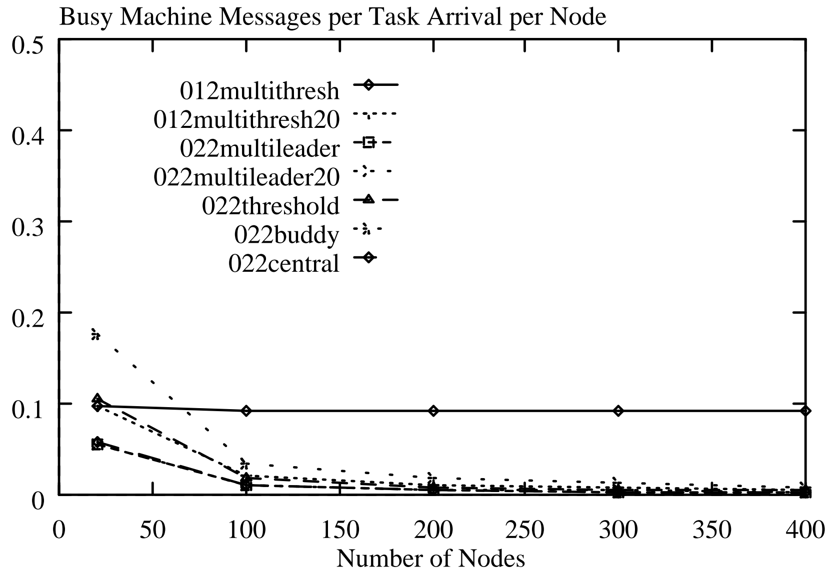

Figure 17: Busy Machine Messages for Homogeneous Workload of 0.7 and Variable Number of Nodes

In summary, the multithresh policy exhibits poor scaling characteristics just as found by Theimer and Lantz in their work [6]. However the multithresh20, along with the other policies exhibit desirable scaling characteristics and can be used in large environments.

7 Conclusion

In this work we have shown that the use of multicasting to locate lightly-loaded machines works well in networks that support efficient delivery of such messages. Our work uses simulation to compare the use of multicasting for load sharing directly with other load sharing approaches in a controlled environment. The results show that multicasting is an efficient method for locating lightly-loaded nodes, yielding better response time compared to previous policies. In addition, the results show that multicast-based policies can be used to lessen network traffic to busy nodes and nodes on remote LANs.

The work makes a number of contributions. The primary contribution is the introduction and study of multicast-based load sharing policies that can scale to large numbers of machines. Theimer and Lantz had previously proposed policies based on multicasting, but their study was limited to measuring network traffic for two straightforward implementations. The policies we propose scale better, are more resilient to failed nodes and exhibit better performance. Another contribution of our work is that we compare performance of our policies with several other well-known policies not based on multicasting.

As part of this comparison, we offer a more realistic model of how machines are interconnected as number of machines grows. Rather than assume all machines reside on the same LAN, we model groups of machines to reside on LANs which themselves are interconnected with a backbone network. This model allows us to not only examine policies based on the standard measure of response time, but to also introduce new criteria based on examining the number of messages handled

by busy machines and the number that are transported on the backbone network.

Finally, we make contributions in examining the policies under different scenarios by varying the message overhead, workload and scale. Our network model allows us to examine nonhomogeneous workloads partitioned not only by heavily and lightly-loaded machines, but also by heavily and lightly-loaded LANs, which are individually homogeneous.

In conclusion, we found the multithresh20 policy, where lightly-loaded nodes are grouped by LANs, to generally be the best performing policy with the least amount of message traffic to busy machines and on the backbone network. The policy also exhibits excellent scaling characteristics as opposed to the basic multithresh policy, which can result in many messages for a large number of nodes. The policy is simple, yet it is able to make good load sharing decisions, and is resilient to node failures by not maintaining any centralized state.

References

[1] M. W. Mutka and M. Livny. Profiling workstations' available capacity for remote execution. In *Performance '87, Proceedings of the 12th IFIP WG 7.3 Symposium on Computer Performance*, 1987.

[2] Craig E. Wills and David Finkel. Load sharing using multicasting. In *Proceedings of the Twelfth Annual IEEE International Phoenix Conference on Computers and Communications*, pages 303–309, March 1993.

[3] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[4] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[5] Kang G. Shin and Yi-Chieh Chang. Load sharing in real-time systems with state-change broadcasts. *IEEE Transactions on Computers*, C-38(8):1124–1142, August 1989.

[6] Marvin M. Theimer and Keith A. Lantz. Finding idle machines in a workstation-based distributed system. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 112–122, June 1988.

[7] David Finkel. Modeling dynamic load sharing in distributed computer systems. *Computer Systems: Science and Engineering*, 5:89–94, 1990.

[8] S. Zhou. A trace driven simulation study of dynamic load balancing. Technical Report UCB/CSD 87/305, University of California at Berkeley, September 1986.

[9] Niranjan G. Shivaratri and Phillip Krueger. Two adaptive location policies for global scheduling algorithms. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 502–509, May-June 1990.

[10] David R. Cheriton and Timothy P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[11] Craig E. Wills. Strategies for using multicasting to locate resources. In *Proceedings IEEE 16th Conference on Local Computer Networks*, pages 589–598, October 1991.

[12] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[13] S. Deering. Host extensions for IP multicasting, August 1989. RFC 1112.

[14] Herb Schwetman. *CSIM Users' Guide (Revision 14)*. Microelectronics and Computer Technology Corporation, 1990.

[15] Craig E. Wills and David Finkel. Load sharing using multicasting. Technical Report WPI-CS-TR-91-15, Worcester Polytechnic Institute, December 1991.

[16] David R. Boggs, Jeffrey C. Mogul, and Christopher A. Kent. Measured capacity of an Ethernet: Myths and reality. In *Proceedings of ACM SIGCOMM '88*, pages 222–234, August 1988.