| Bandwidth $[\frac{KB}{sec}]$ / # of Nodes | (0, 4] | (4, 16] | (16, 64] | (64, 256] |
|---|---|---|---|---|
| 40 | 2 | | | |
| 200 | 3 | 2 | 5 | |
| 400 | 6 | 3 | 11 | |
| 600 | 8 | 4 | 18 | |
| 800 | 10 | 5 | 4 | 21 |
| 1000 | 12 | 6 | 6 | 26 |
| 1200 | 12 | 8 | 8 | 32 |
| 1400 | 13 | 10 | 9 | 38 |
| 1600 | 16 | 10 | 10 | 44 |

**Figure 10**: **Intrusiveness on the LANs for the distributed algorithm with a heterogeneous load.**

## 7 Conclusions

We have shown that existing approaches to load sharing do not scale while supporting a rich set of policies and satisfying bounds on intrusiveness. Further, we have argued that in an owner-based distributed system, it is not sufficient to choose a load sharing algorithm which typically induces little overhead on the nodes and LANs; rather, the load sharing algorithm must be able to regulate and control the overhead. Finally, we outline the distributed clustering algorithm that provides scalable and non-intrusive load sharing for a rich set of sharing policies.

## References

[1] A. Barak and A. Litman. Mos: A multicomputer distributed operating system. *Software Practice and Experience*, 15(8):725–737, August 1985.

[2] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–153, February 1988.

[3] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software–Practice and Experience*, 21(8):757–85, August 1991.

[4] D. L. Eager, D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogenous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):–, May 1986.

[5] D. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic algorithms for load balancing. In *Proceedings of 8th International Conference on Distributed Computing*, pages 491–499, Los Alamitos, California, 1988.

[6] L. Kleinrock. *Queueing Systems, Volume 1: Theory*. John Wiley and Sons, Inc., 1975.

[7] P. Krueger and R. Chawla. The stealth distributed scheduler. In *Proceedings of 11th International Conference on Distributed Computing*, pages 336–343, Los Alamitos, California, 1991.

[8] P. Krueger and M. Livny. A comparison of preemptive and non-preemptive load distributing. In *Proceedings of 8th International Conference on Distributed Computing*, pages 123–30, Los Alamitos, California, 1988.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):95–114, July 1978.

[10] H.-C. Lin and C.S. Raghavendra. A dynamic load-balancing policy with a central job dispatcher. *IEEE Transactions on Software Engineering*, 18(2):148–58, February 1992.

[11] M. J. Litzkow, M. Livny, and M. W. Mutka. A hunter of idle workstations. In *Proceedings of 8th International Conference on Distributed Computing*, pages 104–111, Los Alamitos, California, 1988.

[12] D. A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the 11th ACM Symposium on Operating Systems Principals*, pages 5–12, 1987.

[13] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 6(2):33–44, December 1992.

[14] M. M. Theimer and K. A. Lantz. Finding idle machines in a workstation-based distributed system. In *Proceedings of 8th International Conference on Distributed Computing*, pages 112–122, Los Alamitos, California, 1988.

[15] C. Waldspurger, T. Hogg, B. Huberman, J. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, Feb 1992.

**Figure 4: Average response time.**

| # of Nodes | Intrusiveness |
|---:|---:|
| 40 | 0.035 |
| 200 | 0.179 |
| 400 | 0.370 |
| 600 | 0.583 |
| 800 | 0.770 |
| 1000 | 0.884 |
| 1200 | 1.117 |
| 1400 | 1.322 |

**Figure 5: Intrusiveness on the manager for the centralized algorithm with a heterogeneous load.**

| Intru-siveness / # of Nodes | (0.000, 0.067] | (0.067, 0.110] | (0.110, 0.135] | (0.135, 0.208] |
|---:|---:|---:|---:|---:|
| 40 | 2 | | | |
| 200 | 10 | | | |
| 400 | 19 | 1 | | |
| 600 | 28 | 2 | | |
| 800 | 37 | 2 | 1 | |
| 1000 | 41 | 8 | 1 | |
| 1200 | 45 | 13 | 2 | |
| 1400 | 48 | 16 | 4 | 2 |
| 1600 | 51 | 17 | 10 | 2 |

**Figure 6: Intrusiveness on managers for the DC algorithm with a heterogeneous load.**

| Intru-siveness / # of Nodes | (0.000, 0.067] | (0.067, 0.110] | (0.110, 0.135] | (0.135, 0.208] |
|---:|---:|---:|---:|---:|
| 40 | 40 | | | |
| 200 | 200 | | | |
| 400 | 490 | 20 | | |
| 600 | 600 | | | |
| 800 | 700 | 100 | | |
| 1000 | 910 | 90 | | |
| 1200 | 1140 | 60 | | |
| 1400 | 1040 | 160 | 200 | |
| 1600 | 1120 | 240 | 100 | 140 |

**Figure 7: Intrusiveness on nodes for the distributed algorithm with a heterogeneous load.**

| # of Nodes | Bandwidth $[\frac{KB}{sec}]$ |
|---:|---:|
| 40 | 1.8102 |
| 200 | 9.7180 |
| 400 | 20.7440 |
| 600 | 33.9083 |
| 800 | 44.6015 |
| 1000 | 47.6991 |
| 1200 | 66.1447 |
| 1400 | 87.3665 |

**Figure 8: Intrusiveness on the LAN of the manager for the centralized algorithm with a heterogeneous load.**

| Bandwidth $[\frac{KB}{sec}]$ / # of Nodes | (1, 4] | (4, 8] | (8, 16] |
|---:|---:|---:|---:|
| 40 | 2 | | |
| 200 | 9 | 1 | |
| 400 | 19 | 1 | |
| 600 | 27 | 3 | |
| 800 | 28 | 11 | 1 |
| 1000 | 20 | 29 | 1 |
| 1200 | 23 | 35 | 2 |
| 1400 | 27 | 41 | 2 |
| 1600 | 30 | 45 | 5 |

**Figure 9: Intrusiveness on the LANs for the DC algorithm with a heterogeneous load.**

**Figure 2: Intrusiveness on the nodes.**

**Figure 3: Intrusiveness on the LANs.**

distributed algorithms introduce very little overhead.

We now consider a heterogeneous environment in which location policy requires us to first search for an available processor within the local cluster and then to try all the processors in each neighboring cluster. Cluster size is fixed at 20 processors per cluster. The simulated receiving policies are such that half of the clusters will not accept any remote tasks, though they are still generating remote task requests at an average rate of one request per minute per node. The other half of the clusters are relatively idle, but their nodes are generating state update messages at the rate of one update per 10 seconds per node. The results (see Figure 4) show that both our DC approach and fully distributed load sharing scale beyond 1400 nodes; however, response times are an order of magnitude higher in the distributed case. For this workload, the centralized case fails to scale beyond about 1000 nodes.

The tables in Figures 5–7 report the fraction of compute cycles expended on load sharing overhead. The latter two tables report overhead distributions rather than simple averages, with each column representing an intrusiveness range. For example, in Figure 6, we see that for a 1400 node system, four cluster managers spent 11–13.5% of their cycles on load sharing overhead. The results for the centralized case show intrusiveness increasing with system size. In fact, load sharing overhead eventually overwhelms the central server, limiting scalability and producing intrusiveness ratings greater than 1. In contrast, the DC algorithm is holding up relatively well; in a 1600 node system only 2 cluster managers are expending 20% of their cycles on load sharing. In contrast, in the distributed algorithm, 140 nodes spend 20% of their cycles in load sharing overhead. Moreover, while load sharing overhead is confined to cluster managers in the DC algorithm, overhead in the fully distributed algorithm can perturb any node in the system.

Finally, the tables in Figures 8 to 10 report network bandwidth overhead. Again, for the DC and distributed algorithms, we report overhead ranges rather than averages. In a 1400 node system, the central server's LAN sees 87KB/sec of traffic. For the 1600 node DC case, assuming each cluster manager is on a different LAN, 94% of the LANs see at most 8KB/sec of load sharing traffic and all LANs see less than 16KB/sec. The distributed algorithm does a bit worse, with 50% of the LANs using 0.25MB/sec for load sharing traffic.

Thus, the simulation results confirm our qualitative assessments. A single centralized server does not scale to thousands of nodes for homogeneous or heterogeneous workloads while the fully distributed algorithm becomes intrusive under heterogeneous workloads. The DC algorithm scales to the system sizes in which we are interested while controlling intrusiveness and limiting overhead to a well-defined set of managers.

state will be stale before it is used, but broadcasting updates to keep this state coherent would introduce the problems of the centralized scheme.

## 6 Comparison of load sharing approaches

In this section, we compare the three load sharing algorithms discussed above with respect to their ability to scale with acceptable intrusiveness in an owned environment. Unfortunately, even under the simple workload and policy models, the fully distributed and the DC approaches are not amenable to analytic techniques. Consider the simple case when task arrivals and departures can be modeled with Poisson distributions. To evaluate scalability for each node in distributed load sharing, we must estimate scheduling response time, $\bar{x}_p$. But $\bar{x}_p$ depends on the number of nodes to be polled, which depends in turn on workload distribution, receiving policy, and the policies that determine location order. And, response time depends not only on the number of nodes polled but also on current message arrival rates ($\mu_p$) at these nodes. Given this complexity, we use simulation techniques to compare the different load sharing algorithms.

### 6.1 Simulation model

Each node has its own local and remote task submission rates. In order to model a realistic sending policy, we assume that when a task is submitted for remote execution, it will consume non-negligible computation cycles. This assumption motivates us to bound the frequency with which remote tasks are submitted by a processor to one per minute. Every local and remote submission has an associated CPU utilization and communication bandwidth usage. These task submissions and their corresponding task completions change the state of the node and the state of the LAN to which the node is connected.

The frequency of state changes depends upon the policies specified by owners, as well as the workload. For a location policy that tries to balance load or reduce average response time by assigning a remote task to the "least busy" processor in the system, each state change due to local task arrivals and departures must be considered. Alternatively, if the location policy simply searches for an available processor and the receiving policy is to accept tasks on weekends only, then state changes occur only at the beginning and end of the weekend. To support a broad range of policies,

**Figure 1: Average response time.**

we use the "least busy" processor policy to motivate an upper bound on state change frequency of once per node per 10 seconds.

Finally, we consider both a homogeneous and a heterogeneous workload. Details of the workloads and the simulation results are reported in the section below.

### 6.2 Comparison

In our simulation of a homogeneous environment, the mean rate at which processors submit remote tasks is one task per 60 seconds. For the centralized and DC cases, each node sends a state update message to its manager at a rate of one in every 10 seconds. Recall from Section 4 that under homogeneous conditions we expect centralized load sharing to have poor scalability. This qualitative conclusion is supported by the simulation results in Figure 1 where we plot average response time $\bar{x}$ vs. number of processors $N$. Both the DC algorithm and the distributed approach scale well beyond 1400 nodes, whereas the centralized scheme requires about five seconds to service a request when there are 600 nodes and becomes unstable beyond 856 nodes. Figures 2 and 3 display results on the overhead of the load sharing algorithms as a function of the number of nodes. These graphs respectively plot the maximum fraction of compute cycles dedicated to load sharing overhead at any node and the maximum network overhead in any LAN. As predicted earlier, the intrusiveness of the centralized scheme scales linearly with the number of nodes while the DC and then a search of all the nodes in each "nearby" LAN. If the workload is such that half the LANs are idle and the other half are busy, then the overhead of $N$ nodes searching for an idle machine can quickly generate $O(N^2)$ messages and increase response time. Thus, we cannot predictably bound the intrusiveness of the scheme.

Various researchers have proposed randomized polling to avoid this behavior. Under homogeneous conditions with uniform task arrival rates and uniform node service rates, it has been shown that for reasonable system utilizations, a node will find an appropriate available remote node with 2 to 5 random probes [4][2]. For a shared pool of processors, this randomized polling approach seems perfectly reasonable. But, in an owner-based system, processors will be partitioned into political units and location and receiving policies may encourage or even require a processor to poll in a particular order. For example, in a university setting we may first try to exploit idle cycles in our own research group, then in a related research group, then within the department and finally throughout the university. Applying such a scheme would naturally produce the workload described in the previous paragraph. Quantitative results about the high message traffic and scheduling overhead for this workload are given in Section 6.

## 5 DC algorithm

In the last section, we noted serious shortcomings in both the centralized and the fully distributed load sharing algorithms. The centralized algorithm does not offer the scalability we require, while the distributed algorithm is intrusive. We can overcome these pitfalls by using a *parameterized* hybrid distributed clustering (DC) approach. We group nodes into clusters. A manager is assigned to each cluster to maintain the state of the cluster. As stated above, the definition of state depends on the policies we are implementing. Nodes within a cluster send state update messages and remote task requests to their cluster manager. The size of a cluster is a parameter which is determined with respect to the scalability and intrusiveness measures of the system. Nodes on a particular LAN may be partioned into more than one cluster. Alternatively, a cluster manager may be responsible for nodes connected to several LANs.

[2]The effectiveness of the randomized algorithm depends on its homogeneity assumptions. If, for example, only one node in a large system is idle but that node is extremely fast, then system utilization can be moderate but pure randomized polling will still generate $O(N^2)$ messages to locate the idle cycles on the fast node.

The set of cluster managers may change over time to handle changes in bounds on intrusiveness and fairness. In the simplest case, an owner of a node which has been running as a cluster manager may require exclusive use of his node and the manager will migrate.

When a manager receives a remote task request and the request cannot be satisfied by any node within the cluster, the cluster manager polls other cluster managers. As alluded to in our discussion of information and location mechanisms, the number of cluster managers that are polled depends on the amount of state information maintained about other clusters and the polling order. While coherent global state would allow us to minimize polling, maintaining global state would introduce the scalability problems inherent in the centralized approach. On the other hand, if managers do not maintain any state information about other clusters, then the intrusiveness of the load sharing algorithm on the networks and managers cannot be controlled. To avoid this problem, in the DC algorithm each manager keeps a record of which clusters are currently available for polling. When a manager detects that it is in danger of violating its node's or LANs intrusiveness constraints (Equations 2 and 3), it notifies all other managers that it is removing itself from the load sharing pool and those managers update their state accordingly.

Thus, DC uses state information to guarantee non-intrusiveness. Additional state information can be exploited to reduce the number of managers that need to be polled to satisfy a remote request. We divide the state of each cluster into two categories: frequently and infrequently changing state. A state is said to change frequently when the frequency of state transitions is greater than or equal to the average arrival rate of remote tasks. Otherwise, that state is said to change infrequently. A cluster manager stores the relevant infrequently changing state of other clusters. In order to reduce the amount of stored state, caching schemes can be employed to maintain only frequently accessed state information.

The rationale behind the DC algorithm's state maintenance strategy is as follows. Certain policies like calendar based receiving policies induce infrequent state changes and the current state of a node governed by such policies can be correctly predicted from stored information. On the other hand, there can be frequent state changes due to policies that are based on the load of a resource (e.g., least busy processor scheduling). Caching this type of state is useless since the fundamental scalability and non-intrusiveness issues. In this section, we consider the scalability and intrusiveness of two widely used approaches to load sharing, centralized and fully distributed. Further discussions of scalability may also be found in [14, 15].

## 4.1 Centralized approach

In the centralized approach as described in [14], there is a dedicated server that maintains global knowledge of every node's available resources and current receiving policy. Each node in the system sends state update messages to the dedicated central server. When a task is submitted for remote execution, the node where the task is submitted forwards the remote task request to the central server, which uses its state information to schedule the task on an appropriate available node.

We will use a simple example to demonstrate the centralized approach's inability to scale. Suppose that the arrival of remote task requests and frequency of local state changes can be modeled with Poisson distributions. Assume further that each node has the same average remote task arrival rate, $\lambda_p$, and the same average frequency of local state changes $\delta_p$. Denote the average time the server takes to service a message by $\bar{y}_s$ and let $N$ be the number of nodes in the system. Then, the average server response time $T$ for a node can be calculated with the following formula [6]:

$$T = \frac{\bar{y}_s}{1 - N(\lambda_p + \delta_p)\bar{y}_s}. \qquad (4)$$

Assuming message transmission time is negligible, server response time $T$ is equal to request response time $\bar{x}_p$, and we can calculate the size to which the system scales for a given workload by inserting this equation into the scalability formula (Equation 1). For example, if $\bar{y}_s = 0.01$ seconds, $\lambda_p = 0.0167$ tasks/second, and $\delta_p = 0.1$ state changes/second, the system will not scale beyond $N = 856$ nodes.[1]

Some work has been directed at ameliorating this scalability problem by reducing the number of update messages ($N\delta_p$) that the server processes. The basic idea is to periodically determine the $K$ "least busy" nodes in the system and to have only these "idle" nodes send update messages to the central server [14]. Simulation results and timings of an actual implementation show the approach scales to thousands of nodes. However, the algorithm assumes that it is possible to determine a set of idle nodes. In a homogeneous shared pool of processors, this idle set is obvious, but in an owned environment the set is not well-defined. For example, if the owner of Workstation Seven is only willing to accept remote tasks from members of Department X, then we cannot assign a simple idleness measure to that workstation. More generally, if we want to support flexible local policies, we cannot use the restricted updators scheme outlined above to enhance the scalability of the centralized approach.

Now let us consider the intrusiveness constraints. In the centralized scheme, satisfying non-intrusiveness for all nodes other than the server is straightforward. The server, however, sees update messages in proportion to the number of active nodes in the system. Hence, in order for the server $s$ to satisfy the intrusiveness bound stated in Equation 2, it is normally necessary to choose a large $t_s$. Since most owners do not want to dedicate a large fraction of a node's cycles to load sharing overhead, this constraint normally implies a dedicated server node. To substantiate this argument, consider the example workload above. Suppose the owner of the node is willing to allow the server to run their as long as only one tenth of computation cycles of his node are used for the load sharing algorithm, so $t_s = 0.1$. The average number of messages that the server receives for load sharing is

$$\mu_s = N(\lambda_p + \delta_p). \qquad (5)$$

Inserting Equation 5 in Equation 2, we can calculate the number of nodes $N$ that this server is willing to support. For the values selected above, the server cannot support more than $N = 85$ nodes.

A similar argument applies to the network overhead constraint. The central server sees all the load sharing message traffic in the system which introduces traffic on the particular LAN where the server resides. Thus, it is usually necessary to place the server on its own LAN to satisfy the bound stated in Equation 3.

## 4.2 Distributed approach

In the distributed approach, a remote task request is initiated at a node, and that node polls other nodes in the system to locate an available machine. Note that we describe a sender-initiated approach. For brevity, we do not consider a receiver-initiated approach. In the case of a homogeneous workload and policies, distributed polling eliminates the central server's bottleneck. However, if the workload and policies are not homogeneous, it is difficult to control the overhead that the distributed scheme induces on nodes and LANs, and intrusiveness can be a problem.

For example, a location policy may require a search first through all the nodes within the local LAN and

[1] These parameter values are used for the homogeneous workload simulations; justification may be found in Section 2.3.

centralized approaches (see Section 4.1). However, as is discussed further in Section 4.1, in an owner-based system, the potentially complex owner-defined sharing policies imply that the state of a node cannot be reduced to a single inexpensively cached parameter like system load.

Alternatively, there are probabilistic methods for load balancing that are based on partial state information [1], where each node maintains information about other nodes which is correct (up-to-date) with a given probability. These methods rely on the premise that the information about the state of a node need not be exact to achieve effective load balancing. Although such methods work well under the assumption that there is a pool of processors shared by all users uniformly, they are not appropriate for load sharing in the ownership paradigm because inexact state information may jeopardize ownership rights (e.g., an owner removes his node from the pool absolutely, not probabilisticly). In the paper we propose a clustering method in which a node in each cluster maintains the state of all nodes in the cluster, as well as limited information about other clusters. We refer to this approach as the *distributed clustering* (DC) *load sharing algorithm.*

The thesis of this paper is that the nonintrusiveness of load sharing and its ability to scale to a given system size depends on the workload and policies, and both centralized and fully distributed load sharing algorithms fail to scale to typical system sizes and/or be non-intrusive, under some set of policies and workloads, whereas the DC algorithm scales to systems with thousands nodes and remains non-intrusive.

### 2.3 Workload

In the distributed environment described above, each node generates two types of tasks—local and remote. Remote tasks are created by the sending policies of the node (which may have been induced by the receiving policies as described earlier). In practice, it is possible that the load sharing algorithm cannot find a suitable node on which to run a remote task. Ultimately, a load sharing algorithm should control this problem by regulating the number of remote tasks generated in the system to avoid overwhelming the available resources. Because our goal in this study is to compare different load sharing algorithms in terms of scalability and intrusiveness, we do not elaborate on a system's ability to regulate the creation of remote tasks. Instead, in our simulations, we choose workloads that preserve system stability in the sense that the the mean arrival rate of tasks does not exceed the mean duration of tasks. In this environment, we simply assume that unscheduled remote tasks are resubmitted again.

## 3 Evaluation model

In order to evaluate different load sharing algorithms, we develop measures for scalability and non-intrusiveness.

For a particular node $p$ in the system, let $\lambda_p$ denote the average rate at which the node generates remote task requests and $\bar{x}_p$ denote the average time the load sharing algorithm requires to service the node's request. In other words, $\bar{x}_p$ is the average time that the requester must wait for the load sharing algorithm to determine whether or not there is an available processor. We refer to $\bar{x}_p$ as the average response time. It is desirable to keep $\bar{x}_p small.$ Further, for scalability, we require that each node $p$ in the system must satisfy the relation:

$$\lambda_p \bar{x}_p < 1. \tag{1}$$

Suppose that $N$ is the number of nodes. We say that a load sharing algorithm *scales to a system of size $N$,* if Equation 1 is satisfied for each node in the system.

In an owner-based system, owners normally restrict the degree to which the load sharing algorithm is permitted to intrude on their resources. Let $\mu_p$ denote the average number of messages that a node receives due to the load sharing algorithm. Let $\bar{y}_p$ denote the average time the node takes to service a message. Then, the intrusiveness restriction is expressed as

$$\mu_p \bar{y}_p < t_p \tag{2}$$

where the value $t_p$, $0 \leq t_p \leq 1$ is a bound set by the owner of the node. Similarly, for a particular local area network $n$, the average number of messages that the load sharing algorithm generates in that network, $\gamma_n$, must satisfy a local bound

$$\gamma_n < b_n \tag{3}$$

set by the owner(s) of the network, where $b_n$ is less than or equal to the network bandwidth. We say that a load sharing algorithm is *non-intrusive,* if Equations 2 and 3 are satisfied for each node in the system.

## 4 Previous work

Much of the previous work on load sharing has focused on a particular policy for achieving balanced distribution of work (e.g., [5, 8, 10]) rather than on

In general, a local sending policy is defined in terms of task resource needs and local resource utilization. Resource based sending policies include trying to migrate a program that requires more physical memory than is locally available or attempting to distribute the load once a given CPU utilization threshold is exceeded. In addition, sending policies can sometimes be induced by receiving policies. For example, a receiving policy that states that no foreign tasks can be executed locally between 10am and 5pm will induce the migration of any foreign tasks that are executing at 10am.

### 2.1.2 Selection policy

Selection policy specifies which tasks will actually be migrated when a node becomes a sender. When receiving policies prioritize users, the lowest priority user's tasks will be considered first for migration. Within a priority class, selection policy may consider time of task submission, migration cost, expected execution time, and expected communication overhead.

### 2.1.3 Location policy

A location policy specifies classes of nodes where the remote tasks of an owner should be executed. The user can specify that his task must execute on a node with a certain average load, with special cases including least busy remote node and first available remote node. For a given location policy, the remote node will be chosen subject to the user's priorities on other nodes (which are defined as part of node receiving policies). For example, owner A may have an agreement with owner B to execute A's tasks on B's resources, or a company's organizational hierarchy may be used to decide how resources are to be shared.

## 2.2 Mechanisms

We identify two distinct but interrelated types of mechanism—location and information mechanism. Information mechanism determines the amount of state information to be maintained, and the method for maintaining this state. Location mechanism polls one or more nodes to find an appropriate node for the execution of a task. The number of nodes to be polled depends on the amount of state information. Note that the definition of a state depends on the policies. For example, if a location policy tries to reduce average response time by assigning a remote task to the "least busy" processor in the system, then each local task arrival and departure causes a state change. Alternatively, if the location policy simply searches for an available processor and the receiving policy is to accept tasks on weekends only, then state changes occur only at the beginning and end of the weekend.

The amount of state that is maintained can range from coherent global state to node-local state only. Two common methods for maintaining global state are the centralized and distributed approaches. In the centralized approach, one node maintains the state of the system. This approach requires only one message to update the global state when there is a local state change. Also, since global state is maintained at a single node and the state is coherent, a simple location mechanism can be constructed using a two message request/acknowledge protocol.

In the distributed approach, each local state change is broadcast to every node in the system. If the system architecture does not support broadcast, then $N-1$ messages are necessary to update the global state per local state change, where $N$ is the number of nodes in the system. Moreover, the distributed approach complicates the location mechanism, since it must be guaranteed that when a node $A$ decides that node $B$ in its current state is appropriate for execution of a task, no other node should execute a task on node $B$ before $A$'s task is spawned; otherwise, node $B$'s state may change making it inappropriate for execution of $A$'s task. This is a standard distributed mutual exclusion problem that requires, in the worst case, $2(N-1)$ messages be exchanged before node $A$ can execute its task on node $B$ [9]. Since the distributed approach is more expensive (in terms of the number of messages) than the centralized approach, we consider only the latter in this paper, referring to it as the *centralized load sharing algorithm.*

In the case when each node maintains only its own state, the number of nodes to be polled and the order in which they are polled depends on the location mechanism. Examples include various sender-initiated, receiver-initiated and symmetrically initiated schemes that poll some number of neighboring nodes [13]. In this paper, we refer to the load sharing scheme in which no global state is maintained as the *fully distributed load sharing algorithm.*

There are also information mechanisms which maintain partial state. For example, some algorithms cache information about the current load on other nodes. This system load information is exploited to reduce polling in distributed approaches (e.g., the stable sender-initiated and symmetrically initiated adaptive algorithms in [13]) and to reduce the set of updaters in workload and policies. To substantiate our claims we run simulations with realistic workload and policies where the centralized approach and fully distributed approach fail to scale while respecting owners intrusiveness bounds, whereas the DC approach performs acceptably.

The remainder of this paper is organized as follows. In Section 2, we describe the system model, while in Section 3 we develop measures for scalability and non-intrusiveness. Section 4 surveys previous work on both centralized and distributed techniques for load sharing. Section 5 introduces our load sharing algorithm, distributed clustering. In Section 6, we compare the various load sharing techniques via simulation to show the superior scalability and non-intrusiveness of our algorithm. We conclude in Section 7 with a discussion of the applicability and extensibility of the distributed clustering algorithm.

## 2 System model

A distributed system is a collection of heterogeneous resources that are owned by individuals or groups. Each resource owner specifies policies that define the conditions under which the resource can be shared. We consider two types of resources:

- Computing resources, such as PCs, workstations, multiprocessors, and mainframes.
- Communication resources, such as local area networks (LANs).

We refer to each computing resource as a *node.* We assume that nodes are connected through LANs and the LANs are connected via bridges, gateways and wide area networks.

In this environment, a task submitted by a user at his node is either executed locally at that node, or remotely at another node. We call a task that executes on a node that does not belong to the owner of the task a "foreign" task when we refer to the task in the context of the node that executes the task; we refer to such a task as a "remote" task when we refer to it in the context of the node at which it is submitted. The decision where the task will execute is determined by policies as described in Section 2.1.

The owner of a node specifies a bound on the overhead that the load sharing algorithm can impose on his node. The bound $t_p$ for node $p$ is expressed as a fraction of the CPU cycles of node $p$. Similarly, the owner or owners of the LAN must specify a bound on the overhead that the load sharing algorithm can impose on the LAN. The bound $b_n$ for LAN $n$ is expressed in terms of communication bandwidth.

In the load sharing literature, a load sharing algorithm is typically characterized by four policies: transfer policy, selection policy, location policy, and information policy [2, 13]. Our framework, however, distinguishes the mechanisms and policies that govern a load sharing algorithm. This distinction leads us to modify the "standard" definitions of location and information policy. We give our policy and mechanism definitions below.

### 2.1 Policies

The policies in a load sharing system can be classified into three groups: transfer, selection and location policies. Transfer policy specifies the conditions under which a node is eligible to send or receive a task, selection policy specifies which tasks are eligible to migrate, and location policy specifies the nodes on which a task can be executed. Note, however, that it is up to the owners of the nodes and LANs to specify the policies that govern their particular resources. Below, we describe a reasonable set of transfer, selection, and location policies.

#### 2.1.1 Transfer policies

The transfer policy can be subdivided into *receiving* and *sending policies.* The receiving policy specifies when a node is eligible to receive a foreign task, whereas the sending policy specifies when a node should send a task to another node.

The receiving policy is expressed in terms of administrative and resource-based constraints. Administrative constraints specify who can run jobs on a node, when they can run jobs on a node, and what priority they have to run jobs on the node. These constraints typically capture "political" or "social" reasons for prioritizing or preventing the execution of foreign tasks. For example, a user may want to prohibit tasks from users in group X from executing on his workstation, or a user may simply want to reserve the machine for exclusive use from 9-5 on weekdays. Resource-based constraints specify what fraction of machine resources (real memory, CPU time, disk space, etc.) must be preserved for exclusive use by the local owner. These constraints are approximated by assuming that an owner's recent resource requirements reflect future resource requirements (e.g., if the machine has had a load average of 0.1 for 10 minutes, the user only requires 10% of the CPU) as it is done in most current systems [11].

# Scalable and Non-Intrusive Load Sharing in Owner-Based Distributed Systems

Banu Özden[*] Aaron J. Goldberg Avi Silberschatz [†]

600 Mountain Ave.
AT&T Bell Laboratories
Murray Hill, NJ 07974

**Abstract**

*Previously proposed load sharing algorithms do not support flexible sharing policies in a non-intrusive fashion and do not scale to systems consisting of several thousand workstations, and, therefore, are not amenable for owner-based distributed systems. This paper introduces a new algorithm that supports a rich set of policies while scaling to adequate system sizes with bounded intrusiveness.*

## 1 Introduction

Load sharing in early distributed systems work assumes a "pool of processors" model where available computational power is shared uniformly by all users [4, 14]. Since this model does not preserve owners' rights to their resources, it provides limited incentive to the users to share their resources. Ultimately, the owners should be able to specify *policies* that govern the conditions under which their resources can be shared. We refer to a system in which the load sharing algorithm adheres to these policies as an *owner-based* distributed system. In order to gain wide acceptance, we believe that an owner-based system must satisfy the following three criteria:

1. The load sharing algorithm should not fix the policies; rather, it should provide mechanisms that support a rich set of owner-defined policies. Resource owners can then specify sharing policies that they consider appropriate.

2. The load sharing algorithm must be *non-intrusive.* We call a load sharing algorithm non-intrusive if the overhead that it induces on each resource is less than the bound on overhead defined by the resource owner. By respecting owners' intrusiveness bounds, we gain owner acceptance.

3. The load sharing algorithm must *scale* to the size of typical systems. We say that a load sharing algorithm scales to a given system size, if the mean service rate of the load sharing requests at each node exceeds the average arrival rate of load sharing requests there. We consider systems that are composed of several thousand computers, since the total computational resources in medium and large organizations often exceeds a thousand workstations.

Though recent distributed systems do guarantee the availability of a workstation to its owner [3, 7, 11, 12], they fail to to meet the three criteria above, and therefore do not provide owners adequate control of their resources.

In this paper, we examine the capacity of the existing distributed and centralized load sharing algorithms to scale to typical system sizes and to remain non-intrusive when they support a rich set of policies. We conclude that the centralized load sharing algorithms fail to scale to thousands of workstations, and that the distributed load sharing algorithms yield poor performance and fail to remain non-intrusive under heterogeneous workload. We devise an alternative hybrid, parameterized algorithm, called *distributed clustering* (DC). The DC algorithm utilizes a set of managers, each of which is responsible for locating resources for a cluster of nodes. The basic parameter in the DC algorithm is the number of nodes assigned to a manager, which is determined as a function of

[*] A PhD candidate in the Department of Computer and Electrical Engineering at the University of Texas at Austin and summer intern at AT&T.

[†] On leave from the Department of Computer Sciences, at the University of Texas at Austin.