Load distributing in locally distributed systems

ARTICLE *in* COMPUTER · JANUARY 1993

Impact Factor: 1.44 · DOI: 10.1109/2.179115 · Source: DBLP

3 AUTHORS, INCLUDING:

Manish Singhal
Bhabha Atomic Research Centre

113 PUBLICATIONS **3,296** CITATIONS

SEE PROFILE

Load Distributing for Locally Distributed Systems

Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal
Ohio State University

Load-distributing algorithms can improve a distributed system's performance by judiciously redistributing the workload among its nodes. This article describes load-distributing algorithms and compares their performance.

The availability of low-cost microprocessors and advances in communication technologies has spurred considerable interest in locally distributed systems. The primary advantages of these systems are high performance, availability, and extensibility at low cost. To realize these benefits, however, system designers must overcome the problem of allocating the considerable processing capacity available in a locally distributed system so that it is used to its fullest advantage.

This article focuses on the problem of judiciously and transparently redistributing the load of the system among its nodes so that overall performance is maximized. We discuss several key issues in load distributing for general-purpose systems, including the motivations and design trade-offs for load-distributing algorithms. In addition, we describe several load-distributing algorithms and compare their performance. We also survey load-distributing policies used in existing systems and draw conclusions about which algorithm might help in realizing the most benefits of load distributing.

Issues in load distributing

We first discuss several load-distributing issues central to understanding its intricacies. In this article, we use the terms computer, processor, machine, workstation, and node interchangeably.

Motivation. A *locally distributed system* consists of a collection of autonomous computers connected by a local area communication network. Users submit tasks at their host computers for processing. As Figure 1 shows, the random arrival of tasks in such an environment can cause some computers to be heavily loaded while other computers are idle or only lightly loaded. Load distributing improves performance by transferring tasks from heavily loaded computers, where service is poor, to lightly loaded computers, where the tasks can take advantage of

0018-9162/92/1200-0033$03.00 © 1992 IEEE

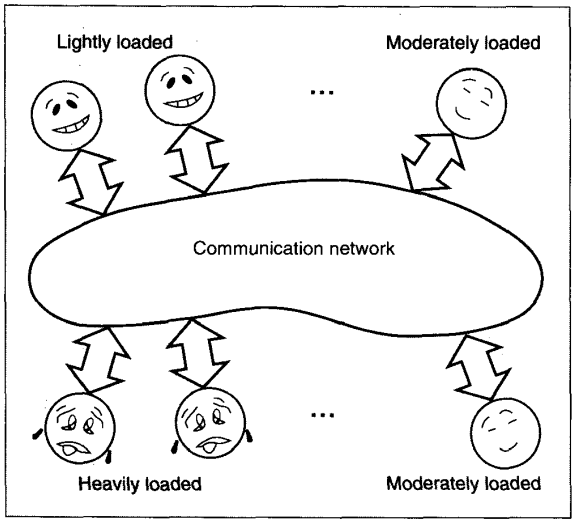


Figure 1. Distributed system without load distributing.

computing capacity that would otherwise go unused.

If workloads at some computers are typically heavier than at others, or if some processors execute tasks more slowly than others, the situation shown in Figure 1 is likely to occur often. The usefulness of load distributing is not so obvious in systems in which all processors are equally powerful and have equally heavy workloads over the long term. However, Livny and Melman[1] have shown that even in such a homogeneous distributed system, at least one computer is likely to be idle while other computers are heavily loaded because of statistical fluctuations in the arrival of tasks to computers and task-service-time requirements. Therefore, even in a homogeneous distributed system, system performance can potentially be improved by appropriate transfers of workload from heavily loaded computers (senders) to idle or lightly loaded computers (receivers).

What do we mean by performance? A widely used performance metric is the *average response time* of tasks. The response time of a task is the time elapsed between its initiation and its completion. Minimizing the average response time is often the goal of load distributing.

Dynamic, static, and adaptive algorithms. Load-distributing algorithms can be broadly characterized as dynamic, static, or adaptive. *Dynamic* load-distributing algorithms use system-state information (the loads at nodes), at least in part, to make load-distributing decisions, while *static* algorithms make no use of such information.

Decisions are hardwired in static load-distributing algorithms using a priori knowledge of the system. For example, under a simple "cyclic splitting" algorithm, each node assigns the ith task it initiates to node $i \bmod N$, where N is the number of nodes in the system. Alternatively, a probabilistic algorithm assigns a task to node i with probability p_i, where the probabilities are determined statically according to factors such as the average task-initiation rate and execution rate for each node. Each of these algorithms can potentially make poor assignment decisions. Because they do not consider node states when making such decisions, they can transfer a task initiated at an otherwise idle node to a node having a serious backlog of tasks.

Dynamic algorithms have the potential to outperform static algorithms by using system-state information to improve the quality of their decisions. For example, a simple dynamic algorithm might be identical to a static algorithm, except that it would not transfer an arriving task if the node where it arrived was idle. A more sophisticated dynamic algorithm might also take the state of the receiving node into account, possibly transferring a task to a node only if the receiving node was idle. A still more sophisticated dynamic algorithm might transfer an executing task if it was sharing a node with another task and some other node became idle. Essentially, dynamic algorithms improve performance by exploiting short-term fluctuations in the system state. Because they must collect, store, and analyze state information, dynamic algorithms incur more overhead than their static counterparts, but this overhead is often well spent. Most recent load-distributing research has concentrated on dynamic algorithms, and they will be our focus for the remainder of this article.

Adaptive load-distributing algorithms are a special class of dynamic algorithms. They adapt their activities by dynamically changing their parameters, or even their policies, to suit the changing system state. For example, if some load-distributing policy performs better than others under certain conditions, while another policy performs better under other conditions, a simple adaptive algorithm might choose between these policies based on observations of the system state. Even when the system is uniformly so heavily loaded that no performance advantage can be gained by transferring tasks, a nonadaptive dynamic algorithm might continue operating (and incurring overhead). To avoid overloading such a system, an adaptive algorithm might instead curtail its load-distributing activity when it observes this condition.

Load. A key issue in the design of dynamic load-distributing algorithms is identifying a suitable *load index*. A load index predicts the performance of a task if it is executed at some particular node. To be effective, load index readings taken when tasks initiate should correlate well with task-response times. Load indexes that have been studied and used include the length of the CPU queue, the average CPU queue length over some period, the amount of available memory, the context-switch rate, the system call rate, and CPU utilization. Researchers have consistently found significant differences in the effectiveness of such load indexes — and that simple load indexes are particularly effective. For example, Kunz[2] found that the choice of a load index has considerable effect on performance, and that the most effective of the indexes we have mentioned is the CPU queue length. Furthermore, Kunz found no performance improvement over this simple measure when combinations of these

load indexes were used. It is crucial that the mechanism used to measure load be efficient and impose minimal overhead.

Preemptive versus nonpreemptive transfers. *Preemptive* task transfers involve transferring a partially executed task. This operation is generally expensive, since collecting a task's state (which can be quite large or complex) is often difficult. Typically, a task state consists of a virtual memory image, a process control block, unread I/O buffers and messages, file pointers, timers that have been set, and so on. *Nonpreemptive* task transfers, on the other hand, involve only tasks that have not begun execution and hence do not require transferring the task's state. In both types of transfers, information about the environment in which the task will execute must be transferred to the remote node. This information may include the user's current working directory and the privileges inherited by the task. Nonpreemptive task transfers are also called *task placements*. Artsy and Finkel[3] and Douglis and Ousterhout[4] contain detailed discussions of issues in preemptive task transfer.

Centralization. Dynamic load-distributing algorithms differ in their degree of centralization. Algorithms can be centralized, hierarchical, fully decentralized, or some combination of these. Algorithms with some centralized components are potentially less reliable than fully decentralized algorithms, since the failure of a central component may cause the entire system to fail. A solution to this problem is to maintain redundant components, which can become active when the previously active component fails. A second weakness of centralized algorithms is not so easily remedied: A central component is potentially a bottleneck, limiting load distribution. While hierarchical algorithms can alleviate both problems, the complete solution lies in fully decentralized algorithms.

Components of a load-distributing algorithm. Typically, a dynamic load-distributing algorithm has four components: a *transfer policy*, a *selection policy*, a *location policy*, and an *information policy*.

Transfer policy. A transfer policy determines whether a node is in a suitable state to participate in a task transfer, either as a sender or a receiver. Many proposed transfer policies are *threshold* policies.[1,5-7] Thresholds are expressed in units of load. When a new task originates at a node, the transfer policy decides that the node is a *sender* if the load at that node exceeds a threshold T_1. On the other hand, if the load at a node falls below T_2, the transfer policy decides that the node can be a *receiver* for a remote task. Depending on the algorithm, T_1 and T_2 may or may not have the same value.

Alternatives to threshold transfer policies include *relative* transfer policies. Relative policies consider the load of a node in relation to loads at other system nodes. For example, a relative policy might consider a node to be a suitable receiver if its load is lower than that of some other node by at least some fixed δ. Alternatively, a node might be considered a receiver if its load is among the lowest in the system.

Selection policy. Once the transfer policy decides that a node is a sender, a selection policy selects a task for transfer. Should the selection policy fail to find a suitable task to transfer, the node is no longer considered a sender.

The simplest approach is to select one of the newly originated tasks that caused the node to become a sender. Such a task is relatively cheap to transfer, since the transfer is nonpreemptive.

A selection policy considers several factors in selecting a task:

(1) The overhead incurred by the transfer should be minimal. For example, a small task carries less overhead.

Load sharing versus load balancing

Dynamic load-distributing algorithms can be further classified as being load-sharing or load-balancing algorithms. The goal of a *load-sharing algorithm* is to maximize the rate at which a distributed system performs work when work is available. To do so, load-sharing algorithms strive to avoid *unshared states*[1]: states in which some computer lies idle while tasks contend for service at some other computer.

If task transfers were instantaneous, unshared states could be avoided by transferring tasks only to idle computers. Because of the time required to collect and package a task's state and because of communication delays, transfers are not instantaneous. The lengthy unshared states that would otherwise result from these delays can be partially avoided through *anticipatory transfers*,[1] which are transfers from overloaded to lightly loaded computers, under the assumption that lightly loaded computers are likely to become idle soon. The potential performance gain of these anticipatory transfers must be weighed against the additional overhead they incur.

Load-balancing algorithms also strive to avoid unshared states, but go a step beyond load sharing by attempting to equalize the loads at all computers. Krueger and Livny[2] have shown that load balancing can potentially reduce the mean and standard deviation of task response times, relative to load-sharing algorithms. Because load balancing requires a higher transfer rate than load sharing, however, the higher overhead incurred may outweigh this potential performance improvement.

References

1. M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proc. ACM Computer Network Performance Symp.*, 1982, pp. 47-55. Proceedings printed as a special issue of *ACM Performance Evaluation Rev.*, Vol. 11, No. 1, 1982, pp. 47-55.
2. P. Krueger and M. Livny, "The Diverse Objectives of Distributed Scheduling Policies," *Proc. Seventh Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 801 (microfiche only), 1987, pp. 242-249.

(2) The selected task should be long lived so that it is worthwhile to incur the transfer overhead.
(3) The number of *location-dependent* system calls made by the selected task should be minimal. Location-dependent calls are system calls that must be executed on the node where the task originated, because they use resources such as windows, the clock, or the mouse that are only at that node.[4,8]

Location policy. The location policy's responsibility is to find a suitable "transfer partner" (sender or receiver) for a node, once the transfer policy has decided that the node is a sender or receiver.

A widely used decentralized policy finds a suitable node through *polling:* A node polls another node to find out whether it is suitable for load sharing. Nodes can be polled either serially or in parallel (for example, multicast). A node can be selected for polling on a random basis,[5,6] on the basis of the information collected during the previous polls,[1,7] or on a nearest neighbor basis. An alternative to polling is to broadcast a query seeking any node available for load sharing.

In a centralized policy, a node contacts one specified node called a *coordinator* to locate a suitable node for load sharing. The coordinator collects information about the system (which is the responsibility of the information policy), and the transfer policy uses this information at the coordinator to select receivers.

Information policy. The information policy decides when information about the states of other nodes in the system is to be collected, from where it is to be collected, and what information is collected. There are three types of information policies:

(1) *Demand-driven policies.* Under these decentralized policies, a node collects the state of other nodes only when it becomes either a sender or a receiver, making it a suitable candidate to initiate load sharing. A demand-driven information policy is inherently a dynamic policy, as its actions depend on the system state. Demand-driven policies may be sender, receiver, or symmetrically initiated. In *sender-initiated* policies, senders look for receivers to which they can transfer their load. In *receiver-initiated* policies, receivers solicit loads from senders. A *symmetrically initiated* policy is a combination of both: Load-sharing actions are triggered by the demand for extra processing power or extra work.

Using detailed state information does not always significantly aid system performance.

(2) *Periodic policies.* These policies, which may be either centralized or decentralized, collect information periodically. Depending on the information collected, the transfer policy may decide to transfer tasks. Periodic information policies generally do not adapt their rate of activity to the system state. For example, the benefits resulting from load distributing are minimal at high system loads because most nodes in the system are busy. Nevertheless, overheads due to periodic information collection continue to increase the system load and thus worsen the situation.

(3) *State-change-driven policies.* Under state-change-driven policies, nodes disseminate information about their states whenever their states change by a certain degree. A state-change-driven policy differs from a demand-driven policy in that it disseminates information about the state of a node, rather than collecting information about other nodes. Under centralized state-change-driven policies, nodes send state information to a centralized collection point. Under decentralized state-change-driven policies, nodes send information to peers.

Stability: We first informally describe two views of stability: the queuing theoretic perspective and the algorithmic perspective. According to the *queuing theoretic perspective*, when the long-term arrival rate of work to a system is greater than the rate at which the system can perform work, the CPU queues grow without bound. Such a system is termed unstable. For example, consider a load-distributing algorithm performing excessive message exchanges to collect state information. The sum of the load due to the external work arriving and the load due to the overhead imposed by the algorithm can become higher than the service capacity of the system, causing system instability.

On the other hand, an algorithm can be stable but still cause a system to perform worse than the same system without the algorithm. Hence, we need a more restrictive criterion for evaluating algorithms — the *effectiveness* of an algorithm. A load-distributing algorithm is effective under a given set of conditions if it improves performance relative to a system not using load distributing. An effective algorithm cannot be unstable, but a stable algorithm can be ineffective.

According to the *algorithmic perspective*, if an algorithm can perform fruitless actions indefinitely with nonzero probability, the algorithm is unstable. For example, consider *processor thrashing:* The transfer of a task to a receiver may increase the receiver's queue length to the point of overloading it, necessitating the transfer of that task to yet another node. This process may repeat indefinitely. In this case, a task moves from one node to another in search of a lightly loaded node without ever receiving any service. Casavant and Kuhl[9] discuss algorithmic instability in detail.

Example algorithms

During the past decade, many load-distributing algorithms have been proposed. In the following sections, we describe several representative algorithms that have appeared in the literature. They illustrate how the components of load-distributing algorithms fit together and show how the choice of components affects system stability. We discuss the performance of these algorithms in the "Performance comparison" section.

Sender-initiated algorithms

Under sender-initiated algorithms, load-distributing activity is initiated by an overloaded node (sender) trying to send a task to an underloaded node (receiver). Eager, Lazowska, and Zahorjan[6] studied three simple, yet effective, fully distributed sender-initiated algorithms.

Transfer policy. Each of the algorithms uses the same transfer policy, a threshold policy based on the CPU queue length. A node is identified as a sender if a new task originating at the node makes the queue length exceed a threshold T. A node identifies itself as a suitable receiver for a task transfer if accepting the task will not cause the node's queue length to exceed T.

Selection policy. All three algorithms have the same selection policy, considering only newly arrived tasks for transfer.

Location policy. The algorithms differ only in their location policies, which we review in the following subsections.

Random. One algorithm has a simple dynamic location policy called random, which uses no remote state information. A task is simply transferred to a node selected at random, with no information exchange between the nodes to aid in making the decision. Useless task transfers can occur when a task is transferred to a node that is already heavily loaded (its queue length exceeds T).

An issue is how a node should treat a transferred task. If a transferred task is treated as a new arrival, then it can again be transferred to another node, providing the local queue length exceeds T. If such is the case, then irrespective of the average load of the system, the system will eventually enter a state in which the nodes are spending all their time transferring tasks, with no time spent executing them. A simple solution is to limit the number of times a task can be transferred. Despite its simplicity, this random location policy provides substantial performance improvements over systems not using load distributing.[6] The "Performance comparison" section contains examples of this ability to improve performance.

Threshold. A location policy can avoid useless task transfers by polling a node (selected at random) to determine whether transferring a task would make its queue length exceed T (see Figure 2). If not, the task is transferred to the selected node, which must execute the task regardless of its state when the task actually arrives. Otherwise, another node is selected at random and is polled. To keep the overhead low, the number of polls is limited by a parameter called the *poll limit*. If no suitable receiver node is found within the poll limit polls, then the node at which the task originated must execute the task. By avoiding useless task transfers, the threshold policy provides a substantial performance improvement over the random location policy.[6] Again, we will examine this improvement in the "Performance comparison" section.

Shortest. The two previous approaches make no effort to choose the best destination node for a task. Under the shortest location policy, a number of nodes (poll limit) are selected at random and polled to determine their queue length. The node with the shortest queue is selected as the destination for task transfer, unless its queue length is greater than or equal to T. The destination node will execute the task regardless of its queue length when the transferred task arrives. The performance improvement obtained by using the shortest location policy over the threshold policy was found to be marginal, indicating that using more detailed state information does not necessarily improve system performance significantly.[6]

Information policy. When either the shortest or the threshold location policy is used, polling starts when the transfer policy identifies a node as the sender of a task. Hence, the information policy is demand driven.

Stability. Sender-initiated algorithms using any of the three location policies cause system instability at high system loads. At such loads, no node is likely to be lightly loaded, so a sender is unlikely to find a suitable destination node. However, the polling activity in sender-initiated algorithms increases as the task arrival rate increases, eventually reaching a point where the cost of load sharing is greater than its benefit. At a more extreme point, the workload that cannot be offloaded from a node, together with the overhead incurred by polling, exceeds the node's CPU capacity and instability results. Thus, the actions of sender-initiated algorithms are not effective at high system loads and cause system instability, because the algorithms fail to adapt to the system state.

Receiver-initiated algorithms

In receiver-initiated algorithms, load-distributing activity is initiated from an underloaded node (receiver), which tries to get a task from an overloaded node (sender). In this section, we describe an algorithm studied by Livny and Melman,[1] and Eager, Lazowska, and Zahorjan[5] (see Figure 3).

Transfer policy. The algorithm's threshold transfer policy bases its decision on the CPU queue length. The policy is triggered when a task departs. If the local queue length falls below the threshold T, then the node is identified as a receiver for obtaining a task from a

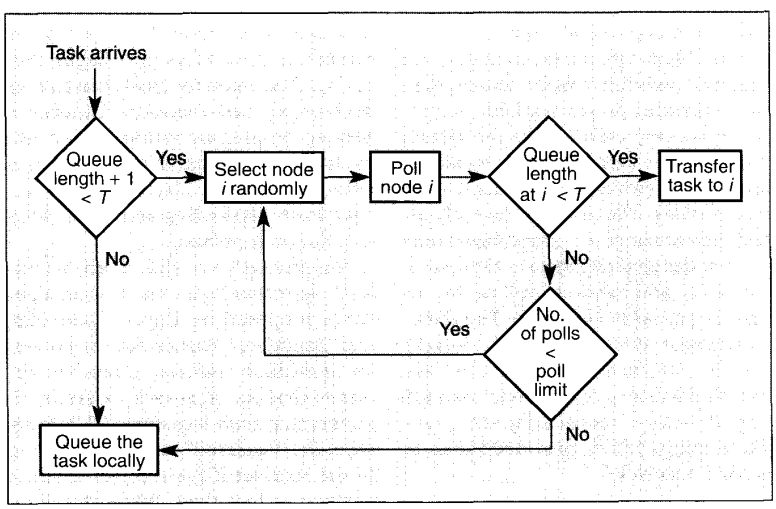


Figure 2. Sender-initiated load sharing using a threshold location policy.

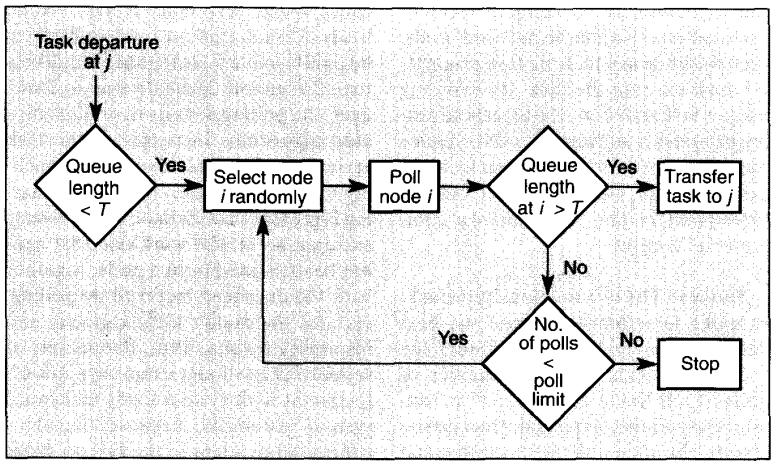


Figure 3. Receiver-initiated load sharing.

node (sender) to be determined by the location policy. A node is identified to be a sender if its queue length exceeds the threshold T.

Selection policy. The algorithm considers all tasks for load distributing, and can use any of the approaches discussed under "Selection policy" in the "Issues in load distributing" section.

Location policy. The location policy selects a node at random and polls it to determine whether transferring a task would place its queue length below the threshold level. If not, then the polled node transfers a task. Otherwise, another node is selected at random, and the procedure is repeated until either a node that can transfer a task (a sender) is found or a static poll limit number of tries has failed to find a sender.

A problem with the location policy is that if all polls fail to find a sender, then the processing power available at a receiver is completely lost by the system until another task originates locally at the receiver (which may not happen for a long time). The problem severely affects performance in systems where only a few nodes generate most of the system workload and random polling by receivers can easily miss them. The remedy is simple: If all the polls fail to find a sender, then the node waits until another task departs or for a predetermined period before reinitiating the load-distributing activity, provided the node is still a receiver.[7]

Information policy. The information policy is demand driven, since polling starts only after a node becomes a receiver.

Stability. Receiver-initiated algorithms do not cause system instability because, at high system loads, a receiver is likely to find a suitable sender within a few polls. Consequently, polls are increasingly effective with increasing system load, and little waste of CPU capacity results.

A drawback. Under the most widely used CPU scheduling disciplines (such as round-robin and its variants), a newly arrived task is quickly provided a quantum of service. In receiver-initiated algorithms, the polling starts when a node becomes a receiver. However, these polls seldom arrive at senders just after new tasks have arrived at the senders but before these tasks have begun executing. Consequently, most transfers are preemptive and therefore expensive. Sender-initiated algorithms, on the other hand, make greater use of nonpreemptive transfers, since they can initiate load-distributing activity as soon as a new task arrives.

An alternative to this receiver-initiated algorithm is the reservation algorithm proposed by Eager, Lazowska, and Zahorjan.[5] Rather than negotiate an immediate transfer, a receiver requests that the next task to arrive be nonpreemptively transferred. Upon arrival, the "reserved" task is transferred to the receiver if the receiver is still a receiver at that time. While this algorithm does not require preemptive task transfers, it was found to perform significantly worse than the sender-initiated algorithms.

Symmetrically initiated algorithms

Under symmetrically initiated algorithms,[10] both senders and receivers initiate load-distributing activities for task transfers. These algorithms have the advantages of both sender- and receiver-initiated algorithms. At low system loads, the sender-initiated component is more successful at finding underloaded nodes. At high system loads, the receiver-initiated component is more successful at finding overloaded nodes. However, these algorithms may also have the disadvantages of both sender- and receiver-initiated algorithms. As with sender-initiated algorithms, polling at high system loads may result in system instability. As with receiver-initiated algorithms, a preemptive task transfer facility is necessary.

A simple symmetrically initiated algorithm can be constructed by combining the transfer and location policies described for sender-initiated and receiver-initiated algorithms.

Adaptive algorithms

A stable symmetrically initiated adaptive algorithm. The *main cause* of system instability due to load sharing in the previously reviewed algorithms is indiscriminate polling by the sender's negotiation component. The stable symmetrically initiated algorithm[7] uses the information gathered during polling (instead of discarding it, as the previous algorithms do) to classify the nodes in the system as *sender/overloaded, receiver/underloaded,* or *OK* (nodes having manageable load). The knowledge about the state of nodes is maintained at each node by a data structure composed of a senders list, a receivers list, and an OK list. These lists are maintained using an efficient scheme: List-manipulative actions, such as moving a node from one list to another or determining to which list a node belongs, impose a small and constant overhead, irrespective of the number of nodes in the system. Consequently, this algorithm scales well to large distributed systems.

Initially, each node assumes that every other node is a receiver. This state is represented at each node by a receivers list containing all nodes (except the node itself), and an empty senders list and OK list.

Transfer policy. The threshold transfer policy makes decisions based on the CPU queue length. The transfer policy is triggered when a new task originates or when a task departs. The policy uses two threshold values — a lower threshold and an upper threshold — to classify the nodes. A node is a sender if its queue length is greater than its upper threshold, a receiver if its queue length is less than its lower threshold, and OK otherwise.

Location policy. The location policy has two components: the *sender-initiated component* and the *receiver-initiated component.* The sender-initiated component is triggered at a node when it becomes a sender. The sender polls the node at the head of the receivers list to determine whether it is still a receiver. The polled node removes the sender node ID from the list it is presently in, puts it at the head of its senders list, and informs the sender whether it is currently a receiver, sender, or OK. On receipt of this reply, the sender transfers the new task if the polled node has indicated that it is a receiver. Otherwise, the polled node's ID is removed from the receivers list and is put at the head of the OK list or the senders list based on its reply.

Polling stops if a suitable receiver is found for the newly arrived task, if the number of polls reaches a poll limit (a parameter of the algorithm), or if the receivers list at the sender node becomes empty. If polling fails to find a receiver, the task is processed locally, though it may later be preemptively transferred as a result of receiver-initiated load sharing.

The goal of the receiver-initiated component is to obtain tasks from a sender node. The nodes polled are selected in the following order:

(1) *Head to tail in the senders list.* The most up-to-date information is used first.
(2) *Tail to head in the OK list.* The most out-of-date information is used first in the hope that the node has become a sender.
(3) *Tail to head in the receivers list.* Again, the most out-of-date information is used first.

The receiver-initiated component is triggered at a node when the node becomes a receiver. The receiver polls the selected node to determine whether it is a sender. On receipt of the message, the polled node, if it is a sender, transfers a task to the polling node and informs it of its state after the task transfer. If the polled node is not a sender, it removes the receiver node ID from the list it is presently in, puts it at the head of the receivers list, and informs the receiver whether the polled node is a receiver or OK. On receipt of this reply, the receiver node removes the polled node ID from whatever list it is presently in and puts it at the head of its receivers list or OK list, based on its reply.

Polling stops if a sender is found, if the receiver is no longer a receiver, or if the number of polls reaches a static poll limit.

Selection policy. The sender-initiated component considers only newly arrived tasks for transfer. The receiver-initiated component can use any of the approaches discussed under "Selection policy" in the "Issues in load distributing" section.

Information policy. The information policy is demand driven, as polling starts when a node becomes either a sender or a receiver.

Discussion. At high system loads, the probability of a node's being underloaded is negligible, resulting in unsuccessful polls by the sender-initiated component. Unsuccessful polls result in the removal of polled node IDs from receivers lists. Unless receiver-initiated polls to these nodes fail to find senders, which is unlikely at high system loads, the receivers lists remain empty. This scheme prevents future sender-initiated polls at high system loads (which are most likely to fail). Hence, the sender-initiated component is deactivated at high system loads, leaving only receiver-initiated load sharing (which is effective at such loads).

At low system loads, receiver-initiated polls are frequent and generally fail. These failures do not adversely affect performance, since extra processing capacity is available at low system loads. In addition, these polls have the positive effect of updating the receivers lists. With the receivers lists accurately reflecting the system's state, future sender-initiated load sharing will generally succeed within a few polls. Thus, by using sender-initiated load sharing at low system loads, receiver-initiated load sharing at high loads, and symmetrically initiated load sharing at moderate loads, the stable symmetrically initiated algorithm achieves improved performance over a wide range of system loads and preserves system stability.

A stable sender-initiated adaptive algorithm. This algorithm[7] uses the sender-initiated load-sharing component of the previous approach but has a modified receiver-initiated component to attract future nonpreemptive task transfers from sender nodes. An important feature is that the algorithm performs load sharing only with nonpreemptive transfers, which are cheaper than preemptive transfers. The stable sender-initiated algorithm is very similar to the stable symmetrically initiated algorithm. In the following, we point out only the differences.

In the stable sender-initiated algorithm, the data structure (at each node) of the stable symmetrically initiated algorithm is augmented by an array called the *state vector.* Each node uses the state vector to keep track of which list (senders, receivers, or OK) it belongs to at all the other nodes in the system. For example, $statevector_i[nodeid]$ says to which list node i belongs at the node indicated by *nodeid.* As in the stable symmetrically initiated algorithm, the overhead for maintaining this data structure is small and constant, irrespective of the number of nodes in the system.

The sender-initiated load sharing is augmented with the following step: When a sender polls a selected node, the sender's state vector is updated to show that the sender now belongs to the senders list at the selected node. Likewise, the polled node updates its state vector based on the reply it sent to the sender node to reflect which list it will belong to at the sender.

The receiver-initiated component is replaced by the following protocol: When a node becomes a receiver, it informs only those nodes that are misinformed about its current state. The misinformed nodes are those nodes whose receivers lists do not contain the receiv-

er's ID. This information is available in the state vector at the receiver. The state vector at the receiver is then updated to reflect that it now belongs to the receivers list at all those nodes that were misinformed about its current state.

There are no preemptive transfers of partly executed tasks here. The sender-initiated load-sharing component will do any task transfers, if possible, on the arrival of a new task. The reasons for this algorithm's stability are the same as for the stable symmetrically initiated algorithm.

Example systems

Here we review several working load-distributing algorithms.

V-system. The V-system[1] uses a state-change-driven information policy. Each node broadcasts (or publishes) its state whenever its state changes significantly. State information consists of expected CPU and memory utilization and particulars about the machine itself, such as its processor type and whether it has a floating-point coprocessor. The broadcast state information is cached by all the nodes. If the distributed system is large, each machine can cache information about only the best N nodes (for example, only those nodes having unused or underused CPU and memory).

The V-system's selection policy selects only newly arrived tasks for transfer. Its relative transfer policy defines a node as a receiver if it is one of the M most lightly loaded nodes in the system, and as a sender if it is not. The decentralized location policy locates receivers as follows: When a task arrives at a machine, it consults the local cache and constructs the set containing the M most lightly loaded machines that can satisfy the task's requirements. If the local machine is one of the M machines, then the task is scheduled locally. Otherwise, a machine is chosen randomly from the set and is polled to verify the correctness of the cached data. This random selection reduces the chance that multiple machines will select the same remote machine for task execution. If the cached data matches the machine's state (within a degree of accuracy), the polled machine is selected for executing the task. Otherwise, the entry for the polled machine is updated and the selection procedure is repeated. In practice, the cache entries are quite accurate, and more than three polls are rarely required.[1]

The V-system's load index is the CPU utilization at a node. To measure CPU utilization, a background process that periodically increments a counter is run at the lowest priority possible. The counter is then polled to see what proportion of the CPU has been idle.

Sprite. The Sprite system[2] is targeted toward a workstation environment. Sprite uses a centralized state-change-driven information policy. Each workstation, on becoming a receiver, notifies a central coordinator process. The location policy is also centralized: To locate a receiver, a workstation contacts the central coordinator process.

Sprite's selection policy is primarily manual. Tasks must be chosen by users for remote execution, and the workstation on which these tasks reside is identified as a sender. Since the Sprite system is targeted for an environment in which workstations are individually owned, it must guarantee the availability of the workstation's resources to the workstation owner. To do so, it evicts foreign tasks from a workstation whenever the owner wishes to use the workstation. During eviction, the selection policy is automatic, and Sprite selects only foreign tasks for eviction. The evicted tasks are returned to their home workstations.

In keeping with its selection policy, the transfer policy used in Sprite is not completely automated:

(1) A workstation is automatically identified as a sender only when foreign tasks executing at that workstation must be evicted. For normal transfers, a node is identified as a sender manually and implicitly when the transfer is requested.

(2) Workstations are identified as receivers only for transfers of tasks chosen by the users. A threshold-based policy decides that a workstation is a receiver when the workstation has had no keyboard or mouse input for at least 30 seconds and the number of active tasks is less than the number of processors at the workstation.

The Sprite system designers used semiautomated selection and transfer policies because they felt that the benefits of completely automated policies would not outweigh the implementation difficulties.

To promote fair allocation of computing resources, Sprite can evict a foreign process from a workstation to allow the workstation to be allocated to another foreign process under the following conditions: If the central coordinator cannot find an idle workstation for a remote execution request and it finds that a user has been allocated more than his fair share of workstations, then one of the heavy user's processes is evicted from a workstation. The freed workstation is then allocated to the process that had received less than its fair share. The evicted process may be automatically transferred elsewhere if idle workstations become available.

For a parallelized version of Unix "make," Sprite's designers have observed a speedup factor of five for a system containing 12 workstations.

Condor. Condor[3] is concerned with scheduling long-running CPU-intensive tasks (background tasks) only. Condor is designed for a workstation environment in which the total availability of a workstation's resources is guaranteed to the user logged in at the workstation console (the owner).

Condor's selection and transfer policies are similar to Sprite's in that most transfers are manually initiated by users. Unlike Sprite, however, Condor is centralized, with a workstation designated as the controller. To transfer a task,

Performance comparison

In this section, we discuss the general performance trends of some of the example algorithms described in the pre-

vious sections. In addition, we compare their performance with that of a system that performs no load distributing (a system composed of n independent M/M/1 systems) and that of an ideal system that performs perfect load distributing (no unshared states) without incurring any overhead in doing so (an M/M/K system). The results we present are from simulations of a distributed system containing 40 nodes, interconnected by a 10-megabit-per-second token ring communication network.

For the simulation we made the following assumptions: Task interarrival times and service demands are independently exponentially distributed, and the average task CPU service demand is one time unit. The size of a polling message is 16 bytes, and the CPU overhead to either send or receive a polling

a user links it with a special system-call library and places it in a local queue of background tasks. The controller's duty is to find idle workstations for these tasks. To accomplish this, Condor uses a periodic information policy. The controller polls each workstation at two-minute intervals to find idle workstations and workstations with background tasks waiting. A workstation is considered idle only when the owner has not been active for at least 12.5 minutes. The controller queues information about background tasks. If it finds an idle workstation, it transfers a background task to that workstation.

If a foreign background task is being served at a workstation, a local scheduler at that workstation checks for local activity from the owner every 30 seconds. If the owner has been active since the previous check, the local scheduler preempts the foreign task and saves its state. If the workstation owner remains active for five minutes or more, the foreign task is preemptively transferred back to the workstation at which it originated. The task may be transferred later to an idle workstation if one is located by the controller.

Condor's scheduling scheme provides fair access to computing resources for both heavy and light users. Fair allocation is managed by the "up-down" algorithm, under which the controller maintains an index for each workstation. Initially the indexes are set to zero. They are updated periodically in the following manner: Whenever a task submitted by a workstation is assigned to an idle workstation, the index of the submitting workstation is increased. If, on the other hand, the task is not assigned to an idle workstation, the index is decreased. The controller periodically checks to see if any new foreign task is waiting for an idle workstation. If a task is waiting, but no idle workstation is available and some foreign task from the lowest priority (highest index value) workstation is running, then that foreign task is preempted and the freed workstation is assigned to the new foreign task. The preempted foreign task is transferred back to the workstation at which it originated.

Stealth. The Stealth Distributed Scheduler[4] differs from V-system, Sprite, and Condor in the degree of cooperation that occurs between load distributing and local resource allocation at individual nodes. Like Condor and Sprite, Stealth is targeted for workstation environments in which the availability of a workstation's resources must be guaranteed to its owner. While Condor and Sprite rely on preemptive transfers to guarantee availability, Stealth accomplishes this task through preemptive allocation of local CPU, memory, and file-system resources.

A number of researchers and practitioners have noted that even when workstations are being used by their owners, they are often only lightly utilized, leaving large portions of their processing capacities available. The designers of Stealth[4] observed that over a network of workstations, this unused capacity represents a considerable portion of the total unused capacity in the system — often well over half.

To exploit this capacity, Stealth allows foreign tasks to execute at workstations even while those workstations are used by their owners. Owners are insulated from these foreign tasks through prioritized local resource allocation. Stealth includes a prioritized CPU scheduler, a unique prioritized virtual memory system, and a prioritized file-system cache. Through these means, owners' tasks get the resources they need, while foreign tasks get only the leftover resources (which are generally substantial). In effect, Stealth replaces an expensive global operation (preemptive transfer) with a cheap local operation (prioritized allocation). By doing so, Stealth can simultaneously increase the accessibility of unused computing capacity (by exploiting underused workstations, as well as idle workstations) and reduce the overhead of load distributing.

Under Stealth, task selection is fully automated. It takes into account the availability of CPU and memory resources, as well as past successes and failures with transferring similar tasks under similar resource-availability conditions. The remainder of Stealth's load-distributing policy is identical to the stable sender-initiated adaptive policy discussed in the "Example algorithms" section of the main text. Because it does not need preemptive transfers to assure the availability of workstation resources to their owners, Stealth can use relatively cheap nonpreemptive transfers almost exclusively. Preemptive transfers are necessary only to prevent starvation of foreign tasks.

References

1. M. Stumm, "The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster," *Proc. Second Conf. Computer Workstations*, IEEE CS Press, Los Alamitos, Calif., Order No. 810, 1988, pp. 12-22.
2. F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software — Practice and Experience*, Vol. 21, No. 8, Aug. 1991, pp. 757-785.
3. M.J. Litzkow, M. Livny, and M.W. Mutka, "Condor — A Hunter of Idle Workstations," *Proc. Eighth Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 865, 1988, pp. 104-111.
4. P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *Proc. 11th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2144, 1991, pp. 336-343.

message is 0.003 time units. A nonpreemptive task transfer incurs a CPU overhead of 0.02 time units, and a preemptive transfer incurs a 0.1 time unit overhead. Transfer overhead is divided evenly between the sending and the receiving nodes. The amount of information that must be communicated for a nonpreemptive transfer is 8 Kbytes, while a preemptive transfer requires 200 Kbytes.

While the specific performance values we present are sensitive to these assumptions, the performance trends we observe are far less sensitive and endure across a wide range of distributed systems. Errors in the results we present are less than 5 percent at the 90 percent confidence level.

Figure 4 plots the performance of the following:

- M/M/1, a distributed system that performs no load distributing;
- a sender-initiated algorithm with a random location policy, assuming that a task can be transferred at most once;
- a sender-initiated algorithm with a threshold location policy;
- a symmetrically initiated algorithm (sender- and receiver-initiated algorithms combined);
- a stable sender-initiated algorithm;
- a receiver-initiated algorithm;
- a stable symmetrically initiated algorithm; and
- M/M/K, a distributed system that performs ideal load distributing without incurring overhead for load distributing.

A fixed threshold of T = upper threshold = lower threshold = 1 was used for each algorithm.

For these comparisons, we assumed a small fixed poll limit (5). A small limit is sufficient: If P is the probability that a particular node is below threshold, then the probability that a node below threshold is first encountered on the ith poll is $P(1 - P)^{i-1}$.[6] (This result assumes that nodes are independent, a valid assumption if the poll limit is small relative to the number of nodes in the system.) For large P, this expression decreases rapidly with increasing i. The probability of succeeding on the first few polls is high. For small P, the quantity decreases more slowly. However, since most nodes are above threshold, the improvement in systemwide response time that will result from locating a node below threshold is small. Quitting the search after the first few polls does not carry a substantial penalty.

Main result. The ability of load distributing to improve performance is intuitively obvious when work arrives at some nodes at a greater rate than at others, or when some nodes have faster processors than others. Performance advantages are not so obvious when all nodes are equally powerful and have equal workloads over the long term. Figure 4a plots the average task response time versus offered system load for such a homogeneous system under each load-distributing algorithm. Comparing M/M/1 with the sender-initiated algorithm (random location policy), we see that even this simple load-distributing scheme provides a substantial performance improvement over a system that does not use load distributing. Considerable further improvement in performance can be gained through simple sender-initiated (threshold location policy) and receiver-initiated load-

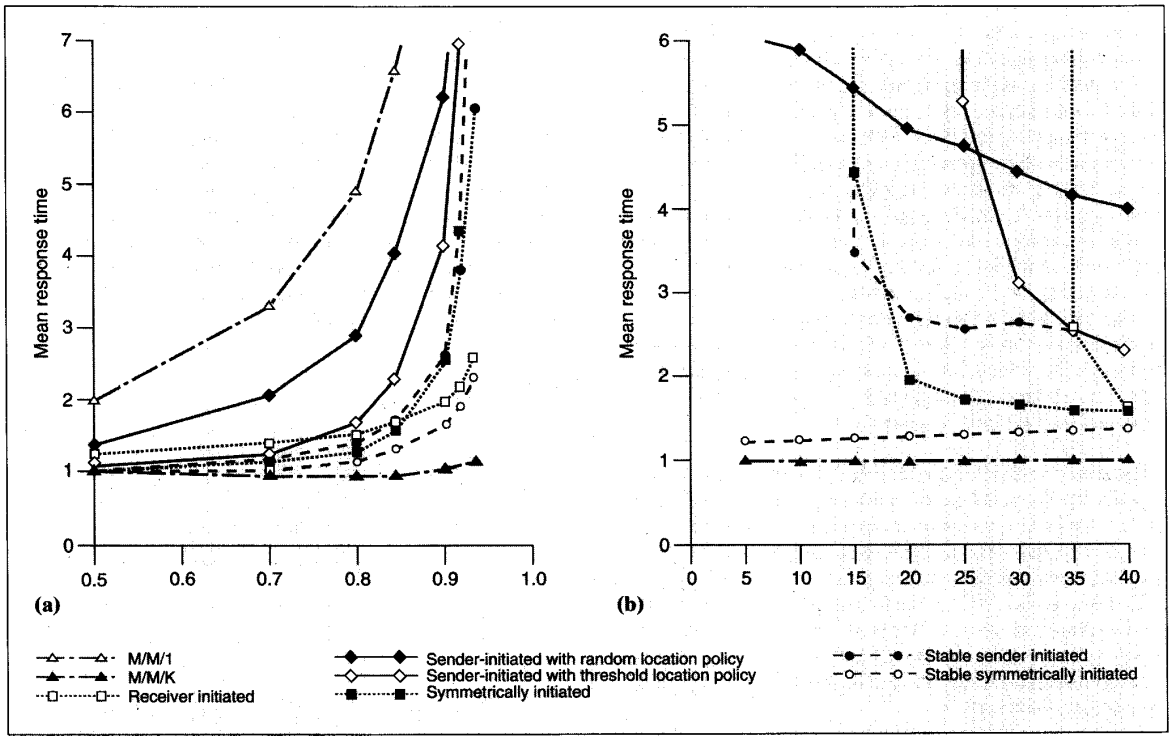


Figure 4. Average response time versus system load (a) and number of load-generating machines (b).

sharing schemes. The performance of the best algorithm — the stable symmetrically initiated algorithm — approaches that of M/M/K, though this optimistic lower bound can never be reached, since it assumes no load-distribution overhead.

Receiver- versus sender-initiated load sharing. Figure 4a shows that the sender-initiated algorithm with a threshold location policy performs marginally better than the receiver-initiated algorithm at light to moderate system loads, while the receiver-initiated algorithm performs substantially better at high system loads (even though the preemptive transfers it uses are much more expensive than the nonpreemptive transfers used by the sender-initiated algorithm). Receiver-initiated load sharing is less effective at low system loads because load sharing is not initiated at the time that one of the few nodes becomes a sender, and thus load sharing often occurs late.

In robustness, the receiver-initiated policy has an edge over the sender-initiated policy. The receiver-initiated policy performs acceptably over the entire system load spectrum, whereas the sender-initiated policy causes system instability at high loads. At such loads, the receiver-initiated policy maintains system stability because its polls generally find busy nodes, while polls due to the sender-initiated policy are generally ineffective and waste resources in efforts to find underloaded nodes.

Symmetrically initiated load sharing. This policy takes advantage of its sender-initiated load-sharing component at low system loads, its receiver-initiated component at high system loads, and both at moderate system loads. Hence, its performance is better than or matches that of the sender-initiated algorithm with threshold location policy at all levels of system load, and is better than that of the receiver-initiated policy at low to moderate system loads. Nevertheless, this policy also causes system instability at high system loads because of the ineffective polling by its sender-initiated component at such loads.

Stable load-sharing algorithms. The performance of the stable symmetrically initiated algorithm matches that of the best of the algorithms at low system loads and offers substantial improvements at high loads (greater than 0.85) over all the nonadaptive algorithms. This performance improvement results from its judicious use of the knowledge gained by polling. Furthermore, this algorithm does not cause system instability.

The stable sender-initiated algorithm yields as good or better performance than the sender-initiated algorithm with threshold location policy, with marked improvement at loads greater than 0.6, and yields better performance than the receiver-initiated policy for system loads less than 0.85. And it does not cause system instability. While it is not as good as the stable symmetrically initiated algorithm, it does not require expensive preemptive task transfers.

Heterogeneous workload. Heterogeneous workloads are common in distributed systems.[8] Figure 4b plots mean response time against the number of load-generating nodes in the system. All system workload is assumed to initiate at this subset of nodes, with none originating at the remaining nodes. A smaller subset of load-generating nodes indicates a higher degree of heterogeneity.

We assume a system load of 0.85. Without load distributing, the system becomes unstable even at low levels of heterogeneity under this load. While we do not plot these results, instability occurs for M/M/1 when the number of load-generating nodes is less than or equal to 33.

Among the load-distributing algorithms, Figure 4b shows that the receiver-initiated algorithm becomes unstable at a much lower degree of heterogeneity than any other algorithm. The instability occurs because random polling is unlikely to find a sender when only a few nodes are senders. The sender-initiated algorithm with a threshold location policy also becomes unstable at relatively low levels of heterogeneity. As fewer nodes receive all the system load, they must quickly transfer tasks. But the senders become overwhelmed as random polling results in many wasted polls.

The symmetrically initiated algorithm also becomes unstable, though at higher levels of heterogeneity, because of ineffective polling. It outperforms the receiver- and sender-initiated algorithms because it can transfer tasks at a higher rate than either. The stable sender-initiated algorithm remains stable for higher levels of heterogeneity than the sender-initiated algorithm with a threshold location policy because it is able to poll more effectively. Its eventual instability results from the absence of preemptive transfers, which prevents senders from transferring existing tasks even after they learn about receivers. Thus senders become overwhelmed.

The sender-initiated algorithm with a random location policy, the simplest algorithm of all, performs better than most algorithms at extreme levels of heterogeneity. By simply transferring tasks from the load-generating nodes to randomly selected nodes without any regard to their status, it essentially balances the load across all nodes in the system, thus avoiding instability.

Only the stable symmetrically initiated algorithm remains stable for all levels of heterogeneity. Interestingly, it performs better with increasing heterogeneity. As heterogeneity increases, senders rarely change their states and will generally be in the senders lists at the nonload-generating nodes. The nonload-generating nodes will alternate between the OK and receiver states and appear in the OK or receivers lists at the load-generating nodes. With the lists accurately representing the system state, nodes are often successful in finding partners.

Over the past decade, the mode of computing has shifted from mainframes to networks of computers, which are often engineering workstations. Such a network promises higher performance, better reliability, and improved extensibility over mainframe systems. The total computing capacity of such a network can be enormous. However, to realize the performance potential, a good load-distributing scheme is essential.

We have seen that even the simplest load-distributing algorithms have a great deal of potential to improve performance. Simply transferring tasks that arrive at busy nodes to randomly chosen nodes can improve performance considerably. Performance is improved still more if potential receiving nodes are first polled to determine whether they are suitable as receivers. Another significant performance benefit can be gained, with little additional complexity, by modifying such an algorithm to poll only the nodes most likely to be

suitable receivers. Each of these algorithms can be designed to use nonpreemptive transfers exclusively. As a result, all carry relatively low overhead in software development time, maintenance time, and execution overhead. Among these algorithms, an algorithm such as the stable sender-initiated algorithm plotted in Figure 4 provides good performance over a wide range of conditions for all but the most "extreme" systems.

By extreme systems we mean systems that may experience periods during which a few nodes generate very heavy workloads. Under such conditions, a load-distributing algorithm that uses preemptive transfers may be necessary. We recommend an algorithm such as the stable symmetrically initiated algorithm, which can initiate transfers from either potential sending or potential receiving nodes, and targets its polls to nodes most likely to be suitable partners in a transfer. Such an algorithm provides larger performance improvements — over a considerably wider range of conditions — than any other algorithm discussed in this article. ■

Acknowledgments

We are deeply grateful to the referees, whose comments helped to improve the presentation. Portions of this material are based on work supported by the National Science Foundation under Grant No. CCR-8909072.

References

1. M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proc. ACM Computer Network Performance Symp.*, 1982, pp. 47-55. Proceedings printed as a special issue of *ACM Performance Evaluation Rev.*, Vol. 11, No. 1, 1982, pp. 47-55.

2. T. Kunz, "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," *IEEE Trans. Software Eng.*, Vol. 17, No. 7, July 1991, pp. 725-730.

3. Y. Artsy and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience," *Computer*, Vol. 22, No. 9, Sept. 1989, pp. 47-56.

4. F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software — Practice and Experience*, Vol. 21, No. 8, Aug. 1991, pp. 757-785.

5. D.L. Eager, E.D. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Performance Evaluation*, Vol. 6, No. 1, Mar. 1986, pp. 53-68.

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

7. N.G. Shivaratri and P. Krueger, "Two Adaptive Location Policies for Global Scheduling," *Proc. 10th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2048, 1990, pp. 502-509.

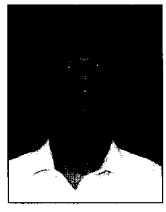
8. P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *Proc. 11th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2144, 1991, pp. 336-343.

9. T.L. Casavant and J.G. Kuhl, "Effects of Response and Stability on Scheduling in Distributed Computing Systems," *IEEE Trans. Software Eng.*, Vol. 14, No. 11, Nov. 1988, pp. 1,578-1,587.

10. P. Krueger and M. Livny, "The Diverse Objectives of Distributed Scheduling Policies," *Proc. Seventh Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 801 (microfiche only), 1987, pp. 242-249.

Niranjan G. Shivaratri is a PhD student in the Department of Computer and Information Science at Ohio State University. From 1983 to 1987, he worked as a systems programmer for the Unisys Corporation, concentrating on network and operating systems software development. His research interests include distributed systems and performance modeling. He and Mukesh Singhal coauthored *Advanced Concepts in Operating Systems*, to be published by McGraw-Hill.

Shivaratri received a BS degree in electrical engineering from Mysore University, India, in 1979, and an MS degree in computer science from Villanova University in 1983. He is a member of the IEEE Computer Society.



Phillip Krueger is an assistant professor of computer science at Ohio State University. He directs the Stealth Distributed Scheduler Project at Ohio State, focusing on the design and construction of a distributed scheduler targeted for workstation-based distributed systems. His research interests include operating systems, concurrent and distributed systems, real-time systems, simulation, and performance analysis.

Krueger received his BS degree in physics, and his MS and PhD degrees in computer science from the University of Wisconsin - Madison. He is a member of the IEEE Computer Society and the ACM.

Mukesh Singhal has been a member of the faculty of the Department of Computer and Information Science at Ohio State University since 1986 and is currently an associate professor. His research interests include distributed systems, distributed databases, and performance modeling. He and Niranjan Shivaratri coauthored *Advanced Concepts in Operating Systems*, to be published by McGraw-Hill. He also served as co-guest editor of the August 1991 special issue of *Computer* on distributed computing systems.

Singhal received a bachelor of engineering degree in electronics and communication engineering with high distinction from the University of Roorkee, India, in 1980, and a PhD in computer science from the University of Maryland in 1986.

Readers can contact the authors at the Dept. of Computer and Information Science, Ohio State University, Columbus, OH 43210. Their e-mail addresses are {niran philk singhal}@cis.ohio-state.edu.