State-change-driven. In this class of policy, nodes disseminate state information whenever their state changes by a certain degree [24]. 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 [32].

11.5 STABILITY

We now describe two views of stability.

11.5.1 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.

Alternatively, an algorithm can be stable but may still cause a system to perform worse than when it is not using the algorithm. Hence, a more restrictive criterion for evaluating algorithms is desirable, and we use the *effectiveness* of an algorithm as the evaluating criterion. A load distributing algorithm is said to be effective under a given set of conditions if it improves the performance relative to that of a system not using load distributing. Note that while an effective algorithm cannot be unstable, a stable algorithm can be ineffective.

11.5.2 The Algorithmic Perspective

If an algorithm can perform fruitless actions indefinitely with finite probability, the algorithm is said to be unstable [3]. For example, consider *processor thrashing*. The transfer of a task to a receiver may increase the receiver's queue length to the point of overload, necessitating the transfer of that task to yet another node. This process may repeat indefinitely [3]. In this case, a task is moved from one node to another in search of a lightly loaded node without ever receiving service. Discussions on various types of algorithmic instability are beyond the scope of this book and can be found in [6].

11.6 LOAD DISTRIBUTING ALGORITHMS

We now describe some load distributing algorithms that have appeared in the literature and discuss their performance.

11.6.1 Sender-Initiated Algorithms

In sender-initiated algorithms, load distributing activity is initiated by an overloaded node (sender) that attempts to send a task to an underloaded node (receiver). This section covers three simple yet effective sender-initiated algorithms studied by Eager, Lazowska, and Zohorjan [11].

Transfer policy. All three algorithms use the same transfer policy, a threshold policy based on 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 remote task if accepting the task will not cause the node's queue length to exceed T.

Selection policy. These sender-initiated algorithms consider only newly arrived tasks for transfer.

Location policy. These algorithms differ only in their location policy:

Random. Random is a simple dynamic location policy that 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 decision making. A problem with this approach is that useless task transfers can occur when a task is transferred to a node that is already heavily loaded (i.e., its queue length is above the threshold). An issue raised with this policy concerns the question of how a node should treat a transferred task. If it is treated as a new arrival, the transferred task can again be transferred to another node if the local queue length is above the threshold. Eager et al. [11] have shown that 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 and not executing them. A simple solution to this problem is to limit the number of times a task can be transferred. A sender-initiated algorithm using the random location policy provides substantial performance improvement over no load sharing at all [11].

Threshold. The problem of useless task transfers under random location policy can be avoided by polling a node (selected at random) to determine whether it is a receiver (see Fig. 11.3). If so, 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 polled. The number of polls is limited by a parameter called *PollLimit* to keep the overhead low. Note that while nodes are randomly selected, a sender node will not poll any node more than once during one searching session of PollLimit polls. If no suitable receiver node is found within the PollLimit polls, then the node at which the task originated must execute the task. By avoiding useless task transfers, the threshold policy provides substantial performance improvement over the random location policy [11].

Shortest. The two previous approaches make no effort to choose the best receiver for a task. Under the *shortest* location policy, a number of nodes (= PollLimit) are

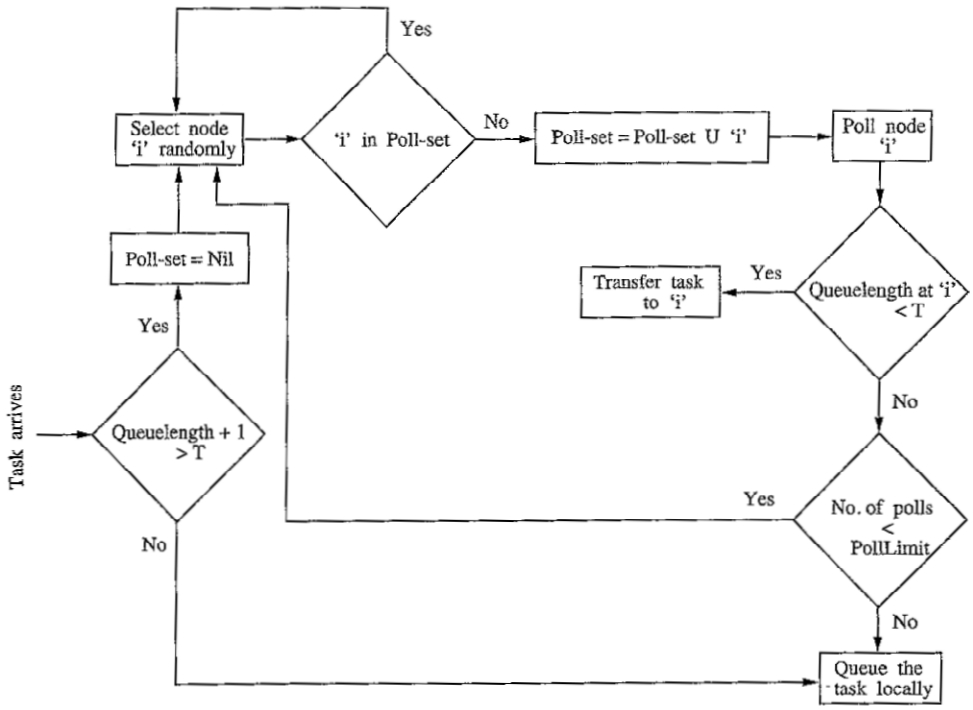

FIGURE 11.3
Sender-initiated load sharing with threshold location policy.

selected at random and are polled to determine their queue length [11]. The node with the shortest queue length is selected as the destination for task transfer unless its queue length $\geq T$. The destination node will execute the task regardless of its queue length at the time of arrival of the transferred task. The performance improvement obtained by using the shortest location policy over the threshold policy was found to be marginal [11], indicating that using more detailed state information does not necessarily result in significant improvement in system performance.

Information policy. When either the shortest or the threshold location policy is used, polling activity commences when the transfer policy identifies a node as the sender of a task. Hence, the information policy can be considered to be of the demand-driven type.

Stability. These three approaches for location policy used in sender-initiated algorithms cause system instability at high system loads, where no node is likely to be lightly loaded, and hence the probability that a sender will succeed in finding a receiver node is very low. However, the polling activity in sender-initiated algorithms increases as the rate at which work arrives in the system increases, eventually reaching a point where the cost of load sharing is greater than the benefit. At this point, most of the available CPU cycles are wasted in unsuccessful polls and in responding to these polls. When the load due to work arriving and due to the load sharing activity exceeds the system's

serving capacity, instability occurs. Thus, the actions of sender-initiated algorithms are not effective at high system loads and cause system instability by failing to adapt to the system state.

11.6.2 Receiver-Initiated Algorithms

In receiver-initiated algorithms, the load distributing activity is initiated from an underloaded node (receiver) that is trying to obtain a task from an overloaded node (sender). In this section, we describe the policies of an algorithm [31] that is a variant of the algorithm proposed in [10] (see Fig. 11.4).

Transfer policy. Transfer policy is a threshold policy where the decision is based on CPU queue length. The transfer policy is triggered when a task departs. If the local queue length falls below the threshold T, the node is identified as a receiver for obtaining a task from a 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. This algorithm can make use of any of the approaches discussed under the selection policy in Sec. 11.4.2.

Location policy. In this policy, a node selected at random is polled to determine if transferring a task from it would place its queue length below the threshold level. If

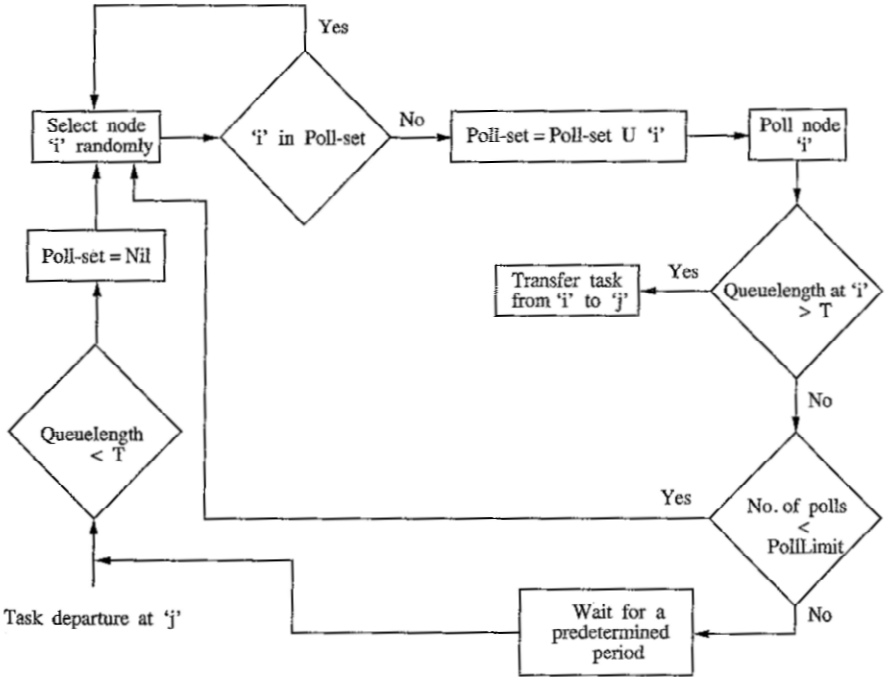

FIGURE 11.4
Receiver-initiated load sharing.

not, the polled node transfers a task. Otherwise, another node is selected at random and the above procedure is repeated until a node that can transfer a task (i.e., a sender) is found, or a static PollLimit number of tries have failed to find a sender. If all polls fail to find a sender, the node waits until another task completes or until a predetermined period is over before initiating the search for a sender, provided the node is still a receiver. Note that if the search does not start after a predetermined period, the extra processing power available at a receiver is completely lost to the system until another task completes, which may not occur soon.

Information policy. The information policy is demand-driven because the polling activity starts only after a node becomes a receiver.

Stability. Receiver-initiated algorithms do not cause system instability for the following reason. At high system loads there is a high probability that a receiver will find a suitable sender to share the load within a few polls. This results in the effective usage of polls from receivers and very little wastage of CPU cycles at high system loads. At low system loads, there are few senders but more receiver-initiated polls. These polls do not cause system instability as spare CPU cycles are available at low system loads.

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, it is unlikely that these polls will be received at senders when new tasks that have arrived at them have not yet begun executing. As a result, a drawback of receiver-initiated algorithms is that most transfers are preemptive and therefore expensive. Conversely, sender-initiated algorithms are able to make greater use of nonpreemptive transfers because they can initiate load distributing activity as soon as a new task arrives.

11.6.3 Symmetrically Initiated Algorithms

Under symmetrically initiated algorithms [21], both senders and receivers search for receivers and senders, respectively, 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 in finding underloaded nodes. At high system loads, the receiver-initiated component is more successful in finding overloaded nodes. However, these algorithms are not immune from the disadvantages of both sender- and receiver-initiated algorithms. As in sender-initiated algorithms, polling at high system loads may result in system instability, and as in receiver-initiated algorithms, a preemptive task transfer facility is necessary.

A simple symmetrically initiated algorithm can be constructed by using both the transfer and location policies described in Secs. 11.6.1 and 11.6.2. Another symmetrically initiated algorithm, called the above-average algorithm [20], is described next.

THE ABOVE-AVERAGE ALGORITHM. The above-average algorithm, proposed by Krueger and Finkel [20], tries to maintain the load at each node within an *acceptable*

range of the system average. Striving to maintain the load at a node at the exact system average can cause processor thrashing [3], as the transfer of a task may result in a node becoming either a sender (load above average) or a receiver (load below average). A description of this algorithm follows.

Transfer policy. The transfer policy is a threshold policy that uses two adaptive thresholds. These thresholds are equidistant from the node's estimate of the average load across all nodes. For example, if a node's estimate of the average load is 2, then the lower threshold = 1 and the upper threshold = 3. A node whose load is less than the lower threshold is considered a receiver, while a node whose load is greater than the upper threshold is considered a sender. Nodes that have loads between these thresholds lie within the acceptable range, so they are neither senders nor receivers.

Location policy. The location policy has the following two components:

Sender-initiated component

- A sender (a node that has a load greater than the acceptable range) broadcasts a *TooHigh* message, sets a *TooHigh* timeout alarm, and listens for an *Accept* message until the timeout expires.
- A receiver (a node that has a load less than the acceptable range) that receives a *TooHigh* message cancels its *TooLow* timeout, sends an *Accept* message to the source of the *TooHigh* message, increases its load value (taking into account the task to be received), and sets an *AwaitingTask* timeout. Increasing its load value prevents a receiver from over-committing itself to accepting remote tasks. If the *AwaitingTask* timeout expires without the arrival of a transferred task, the load value at the receiver is decreased.
- On receiving an *Accept* message, if the node is still a sender, it chooses the best task to transfer and transfers it to the node that responded.
- On expiration of the *TooHigh* timeout, if no *Accept* message has been received, the sender infers that its estimate of the average system load is too low (since no node has a load much lower). To correct this problem, the sender broadcasts a *ChangeAverage* message to increase the average load estimate at the other nodes.

Receiver-initiated component

- A node, on becoming a receiver, broadcasts a *TooLow* message, sets a *TooLow* timeout alarm, and starts listening for a *TooHigh* message.
- If a *TooHigh* message is received, the receiver performs the same actions that it does under sender-initiated negotiation (see above).
- If the *TooLow* timeout expires before receiving any *TooHigh* messages, the receiver broadcasts a *ChangeAverage* message to decrease the average load estimate at the other nodes.

Selection policy. This algorithm can make use of any of the approaches discussed under the selection policy in Sec.11.4.2.

Information policy. The information policy is demand-driven. A highlight of this algorithm is that the average system load is determined individually at each node, imposing little overhead and without the exchange of many messages. Another key point to note is that the acceptable range determines the responsiveness of the algorithm. When the communication network is heavily/lightly loaded (indicated by long/short message transmission delays, respectively), the acceptable range can be increased/decreased by each node individually so that the load balancing actions adapt to the state of the communication network as well.

11.6.4 Adaptive Algorithms

A STABLE SYMMETRICALLY INITIATED ALGORITHM. The main cause of system instability due to load sharing by the previous algorithms is the indiscriminate polling by the sender's negotiation component. The stable symmetrically initiated algorithm [31] utilizes the information gathered during polling (instead of discarding it as was done by the previous algorithms) to classify the nodes in the system as either *Sender/overloaded, Receiver/underloaded,* or *OK* (i.e., nodes having manageable load). The *knowledge* concerning the state of nodes is maintained by a data structure at each node, comprised of a senders list, a receivers list, and an OK list. These lists are maintained using an efficient scheme in which list manipulative actions, such as moving a node from one list to another, or finding the list to which a node belongs, impose a small and constant overhead irrespective of the number of nodes in the system. (See [31] for more details on the list maintenance scheme.)

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

Transfer policy. The transfer policy is a threshold policy where decisions are based on CPU queue length. The transfer policy is triggered when a new task originates or when a task departs. The transfer policy makes use of two threshold values to classify the nodes: a lower threshold (LT) and an upper threshold (UT). A node is said to be a sender if its queue length $>$ UT, a receiver if its queue length $<$ LT, and OK if LT $\leq$ node's queue length $\leq$ UT.

Location policy. The location policy has the following two components:

Sender-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 a receiver, sender, or OK node based on its current status. 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 put at the head of the OK list or at the head of senders list based on its reply. Then the sender polls the node at the head of the receivers list.

The polling process stops if a suitable receiver is found for the newly arrived task, if the number of polls reaches a PollLimit (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 can later migrate as a result of receiver-initiated load sharing.

Receiver-initiated Component. The goal of the receiver-initiated component is to obtain tasks from a sender node. The nodes polled are selected in the following order: head to tail in the senders list (the most up-to-date information is used first); then 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); then 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 its receivers list, and informs the receiver whether it (the polled node) is a receiver or OK. On receipt of the reply, the receiver node removes the polled node ID from whatever list it is presently in and puts it at the head of the appropriate list based on its reply.

The polling process stops if a sender is found, if the receiver is no longer a receiver, or if the number of polls reaches a static PollLimit.

Selection policy. The sender-initiated component considers only newly arrived tasks for transfer. The receiver-initiated component can make use of any of the approaches discussed under the selection policy in Sec. 11.4.2.

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

Discussion. At high system loads, the probability of a node 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 them as senders, which is unlikely at high system loads, the receivers lists remain empty. As a result, future sender-initiated polls at high system loads (which are most likely to fail) are prevented. (Note that a sender polls only nodes found in its receivers list.) 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 polling generally fails. These failures do not adversely affect performance because 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 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 while preserving system stability.

A STABLE SENDER-INITIATED ALGORITHM. This algorithm [31] has two desirable properties. First, it does not cause instability. Second, load sharing is due to nonpreemptive transfers (which are cheaper) only. This algorithm uses the sender-initiated load sharing component of the stable symmetrically initiated algorithm as is, but has a modified receiver-initiated component to attract the future nonpreemptive task transfers from sender nodes. The stable sender-initiated policy is very similar to the stable symmetrically initiated approach, so only the differences will be pointed out.

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 *statevector*. The statevector is used by each node to keep track of which list (senders, receivers, or OK) it belongs to at all the other nodes in the system. Moreover, the sender-initiated load sharing is augmented with the following step: when a sender polls a selected node, the sender's statevector is updated to reflect that the sender now belongs to the senders list at the selected node. Likewise, the polled node updates its statevector 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 all the nodes that are misinformed about its current state. The misinformed nodes are those nodes whose receivers lists do not contain the receiver's ID. This information is available in the statevector at the receiver. The statevector at the receiver is then updated to reflect that it now belongs to the receivers list at all those nodes that were informed of its current state. By this technique, this algorithm avoids receivers sending broadcast messages to inform other nodes that they are receivers. Remember that broadcast messages impose message handling overhead at all nodes in the system. This overhead can be high if nodes frequently change their state.

Note that there are no preemptive transfers of partly executed tasks here. The sender-initiated load sharing component will perform any load sharing, if possible on the arrival of a new task. The stability of this approach is due to the same reasons given for the stability of the stable symmetrically initiated algorithm.

11.7 PERFORMANCE COMPARISON

This section discusses the general performance *trends* of some of the example algorithms described in the previous section. Figure 11.5 through Fig. 11.7 plot the average response time of tasks vs. the offered system load for several load sharing algorithms discussed in Sec. 11.6 [32]. The average service demand for tasks is assumed to be one time unit, and the task interarrival times and service demands are independently exponentially distributed. The system load is assumed to be homogeneous; that is, all nodes have the same long-term task arrival rate. The system is assumed to contain 40 identical nodes. The notations used in the figures correspond to the algorithms as follows:

M/M/1	A distributed system that performs no load distributing.
RECV	Receiver-initiated algorithm.
RAND	Sender-initiated algorithm with random location policy.
SEND	Sender-initiated algorithm with threshold location policy.
ADSEND	Stable sender-initiated algorithm.
SYM	Symmetrically initiated algorithm (SEND and RECV combined).
ADSYM	Stable symmetrically initiated algorithm.
M/M/K	A distributed system that performs ideal load distributing without incurring any overhead.

A fixed threshold of T = lower threshold = upper threshold = 1 was used for these comparisons. However, the value of T should adapt to the system load and the task transfer cost because a node is identified as a sender or a receiver by comparing its queue length with T [11]. At low system loads, many nodes are likely to be idle—a low value of T will result in nodes with small queue lengths being identified as senders who can benefit by transferring load. At high system loads, most nodes are likely to be busy—a high value of T will result in the identification of only those nodes with significant queue lengths as senders, who can benefit the most by transferring load. While a scheduling algorithm may adapt to the system load by making use of an adaptive T, the adaptive stable algorithms of Sec.11.6.4 adapt to the system load by varying the PollLimit with the help of the lists. Also, low thresholds are desirable for low transfer costs as smaller differences in node queue lengths can be exploited; high transfer costs demand higher thresholds.

For these comparisons, a small, fixed PollLimit = 5 was assumed. We can see why such a small limit is sufficient by noting that if P is the probability that a particular node is below threshold, then (because the nodes are assumed to be independent) the probability that a node below threshold is first encountered on the ith poll is $P(1-P)^{i-1}$ [11]. 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. Comparing M/M/1 with the sender-initiated algorithm that uses the random location policy (RAND) in Fig. 11.5, 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 (SEND) and receiver-initiated (RECV) load sharing schemes. M/M/K gives the optimistic lower bound on the performance that can be obtained through load distributing, since it assumes no load distributing overhead.

11.7.1 Receiver-initiated vs. Sender-initiated Load Sharing

It can be observed from Fig. 11.5 that the sender-initiated algorithm (SEND) performs marginally better than the receiver-initiated algorithm (RECV) at light to moderate

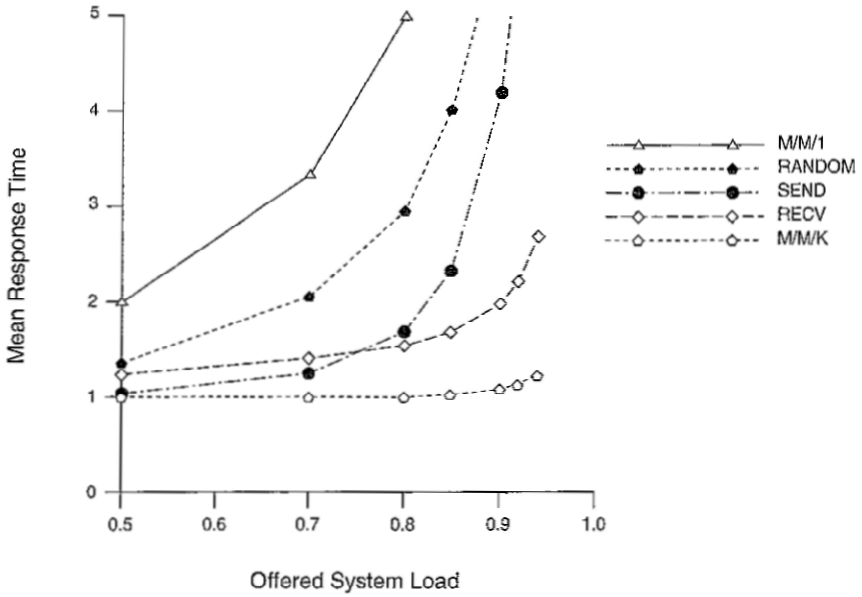

FIGURE 11.5
Average response time vs. system load (adapted from [32]).

system loads, while the receiver-initiated algorithm performs substantially better at high system loads . Receiver-initiated load sharing is less effective at low system loads because load sharing is not initiated when one of the few nodes becomes a sender, and thus load sharing often occurs late.

Regarding the robustness of these policies, the receiver-initiated policy has an edge over the sender-initiated policies. The receiver-initiated policy performs acceptably with a single value of the threshold over the entire system load spectrum, whereas the sender-initiated policy requires an adaptive location policy to perform acceptably at high loads. It can be seen from Fig. 11.5 that at high system 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.

11.7.2 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 of these components at moderate system loads. Hence, its performance is better or matches that of the sender-initiated policy at all levels of system load, and is better than that of receiver-initiated policy at low to moderate system loads [32] (Fig. 11.6). Nevertheless, this policy also causes system instability at high system loads because of the ineffective polling activity of its sender-initiated component at such loads.

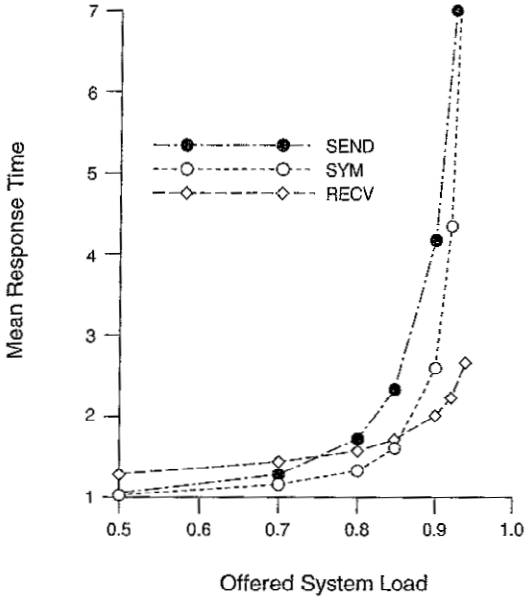

FIGURE 11.6
Average response time vs. system load (adapted from [32]).

11.7.3 Stable Load Sharing Algorithms

The performance of the stable symmetrically initiated algorithm (ADSYM) approaches that of M/M/K (Fig. 11.7), though this optimistic lower bound can never be reached, as it assumes no load distributing overhead. The performance of ADSYM matches that of the sender-initiated algorithm at low system loads and offers substantial improvements at high loads (> 0.85) over all the nonadaptive algorithms [31]. This performance improvement is the result of its judicious use of the knowledge gained by polling. Furthermore, this algorithm does not cause system instability.

The stable sender-initiated algorithm (ADSEND) yields a better performance than the unstable sender-initiated policy (SEND) for system loads > 0.6 and does not cause system instability. While ADSEND is not as effective as ADSYM, it does not require expensive preemptive task transfers.

11.7.4 Performance Under Heterogeneous Workloads

Heterogeneous workloads have been shown to be common for distributed systems [19]. Figure 11.8 plots mean response time against the number of nonload generating nodes at a constant offered system load of 0.85. These nodes originate none of the system workload, while the remaining nodes originate all of the system workload. From the figure, we observe that RECV becomes unstable at a much lower degree of heterogeneity than any other algorithm. The instability occurs because, in RECV, the load sharing does not start in accordance with the arrivals of tasks at a few (but highly overloaded) sender nodes, and random polling by RECV is likely to fail to find a sender when only a small subset of nodes are senders. SEND also becomes unstable with increasing heterogeneity. As fewer nodes receive all the system load, it is imperative that they quickly transfer

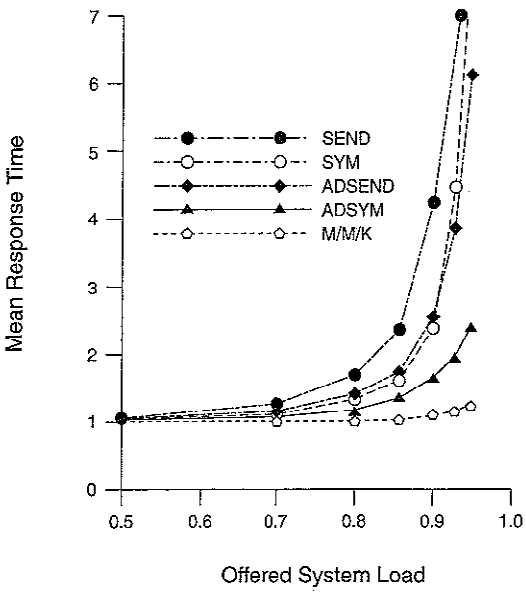

FIGURE 11.7
Average response time vs. system load (adapted from [32]).

tasks. But the senders become overwhelmed, as random polling is ineffective in reducing wasteful tries. SYM also becomes unstable at higher levels of heterogeneity because of ineffective polling. SYM outperforms RECV and SEND because it can transfer tasks at a higher rate than either RECV or SEND alone can. The sender-initiated algorithm with the random location policy (RAND) 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 ADSYM remains stable and performs better with increasing heterogeneity. As heterogeneity increases, senders rarely change their state and will generally be in senders list at nonload generating nodes. The nonload generating nodes will alternate between OK and receiver states and appear in OK or receivers lists at senders. When the lists accurately represent the system state, nodes are often successful at finding partners.

11.8 SELECTING A SUITABLE LOAD SHARING ALGORITHM

Based on the performance trends of load sharing algorithms, one may select a load sharing algorithm that is appropriate to the system under consideration as follows:

1. If the system under consideration never attains high loads, sender-initiated algorithms will give an improved average response time over no load sharing at all.
2. Stable scheduling algorithms are recommended for systems that can reach high loads. These algorithms perform better than nonadaptive algorithms for the following reasons: