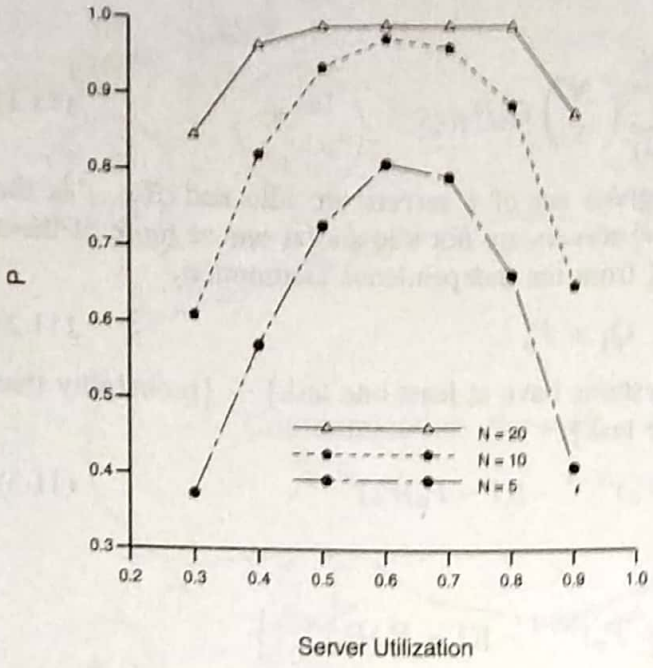

FIGURE 11.2
P as a function of ρ and N (adapted from [24]).

11.3 ISSUES IN LOAD DISTRIBUTING

We now discuss several central issues in load distributing that will help the reader understand its intricacies. Note here that the terms computer, machine, host, workstation, and node are used interchangeably, depending upon the context.

11.3.1 Load

Zhou [41] showed that resource queue lengths and particularly the CPU queue length are good indicators of load because they correlate well with the task response time. Moreover, measuring the CPU queue length is fairly simple and carries little overhead. If a task transfer involves significant delays, however, simply using the current CPU queue length as a load indicator can result in a node accepting tasks while other tasks it accepted earlier are still in transit. As a result, when all the tasks that the node has accepted have arrived, the node can become overloaded and require further task transfers to reduce its load. This undesirable situation can be prevented by artificially incrementing the CPU queue length at a node whenever the node accepts a remote task. To avoid anomalies when task transfers fail, a timeout (set at the time of acceptance) can be employed. After the timeout, if the task has not yet arrived, the CPU queue length is decremented.

While the CPU queue length has been extensively used in previous studies as a load indicator, it has been reported that little correlation exists between CPU queue length and processor utilization [35], particularly in an interactive environment. Hence, the designers of V-System used CPU utilization as an indicator of the load at a site. This approach requires a background process that monitors CPU utilization continuously and imposes more overhead, compared to simply finding the queue length at a node (see Sec. 11.10.1).

11.3.2 Classification of Load Distributing Algorithms

The basic function of a load distributing algorithm is to transfer load (tasks) from heavily loaded computers to idle or lightly loaded computers. Load distributing algorithms can be broadly characterized as *static*, *dynamic*, or *adaptive*. Dynamic load distributing algorithms [3, 10, 11, 18, 20, 24, 31, 34, 40] 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. In static load distributing algorithms, decisions are hard-wired in the algorithm using a priori knowledge of the system. Dynamic load distributing algorithms have the potential to outperform static load distributing algorithms because they are able to exploit short term fluctuations in the system state to improve performance. However, dynamic load distributing algorithms entail overhead in the collection, storage, and analysis of system state information. Adaptive load distributing algorithms [20, 31] are a special class of dynamic load distributing algorithms in that they adapt their activities by dynamically changing the parameters of the algorithm to suit the changing system state. For example, a dynamic algorithm may continue to collect the system state irrespective of the system load. An adaptive algorithm, on the other hand, may discontinue the collection of the system state if the overall system load is high to avoid imposing additional overhead on the system. At such loads, all nodes are likely to be busy and attempts to find receivers are unlikely to be successful.

11.3.3 Load Balancing vs. Load Sharing

Load distributing algorithms can further be classified as *load balancing* or *load sharing* algorithms, based on their load distributing principle. Both types of algorithms strive to reduce the likelihood of an *unshared state* (a state in which one computer lies idle while at the same time tasks contend for service at another computer [21]) by transferring tasks to lightly loaded nodes. Load balancing algorithms [7, 20, 24], however, go a step further by attempting to equalize loads at all computers. Because a load balancing algorithm transfers tasks at a higher rate than a load sharing algorithm, the higher overhead incurred by the load balancing algorithm may outweigh this potential performance improvement.

Task transfers are not instantaneous because of communication delays and delays that occur during the collection of task state. Delays in transferring a task increase the duration of an unshared state as an idle computer must wait for the arrival of the transferred task. To avoid lengthy unshared states, *anticipatory* task transfers from overloaded computers to computers that are likely to become idle shortly can be used. Anticipatory transfers increase the task transfer rate of a load sharing algorithm, making it less distinguishable from load balancing algorithms. In this sense, load balancing can be considered a special case of load sharing, performing a particular level of anticipatory task transfers.

11.3.4 Preemptive vs. Nonpreemptive Transfers

Preemptive task transfers involve the transfer of a task that is partially executed. This transfer is an expensive operation as the collection of a task's state (which can be quite

large and complex) can be 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, etc. Nonpreemptive task transfers, on the other hand, involve the transfer of tasks that have not begun execution and hence do not require the transfer of the task's state. In both types of transfers, information about the environment in which the task will execute must be transferred to the receiving node. This information can include the user's current working directory, the privileges inherited by the task, etc. Nonpreemptive task transfers are also referred to as *task placements*.

11.4 COMPONENTS OF A LOAD DISTRIBUTING ALGORITHM

Typically, a load distributing algorithm has four components: (1) a *transfer* policy that determines whether a node is in a suitable state to participate in a task transfer, (2) a *selection* policy that determines *which* task should be transferred, (3) a *location* policy that determines to which node a task selected for transfer should be sent, and (4) an *information policy* which is responsible for triggering the collection of system state information. A transfer policy typically requires information on the local node's state to make decisions. A location policy, on the other hand, is likely to require information on the states of remote nodes to make decisions.

11.4.1 Transfer Policy

A large number of the transfer policies that have been proposed are *threshold* policies [10, 11, 24, 31]. Thresholds are expressed in units of load. When a new task originates at a node, and the load at that node exceeds a threshold T, the transfer policy decides that the node is a *sender*. If the load at a node falls below T, the transfer policy decides that the node can be a *receiver* for a remote task.

An alternative transfer policy initiates task transfers whenever an imbalance in load among nodes is detected because of the actions of the information policy.

11.4.2 Selection Policy

A selection policy selects a task for transfer, once the transfer policy decides that the node is a sender. Should the selection policy fail to find a suitable task to transfer, the node is no longer considered a sender until the transfer policy decides that the node is a sender again.

The simplest approach is to select newly originated tasks that have caused the node to become a sender by increasing the load at the node beyond the threshold [11]. Such tasks are relatively cheap to transfer, as the transfer is nonpreemptive.

A basic criterion that a task selected for transfer should satisfy is that the overhead incurred in the transfer of the task should be compensated for by the reduction in the response time realized by the task. In general, long-lived tasks satisfy this criterion [4]. Also, a task can be selected for remote execution if the estimated average execution time for that type of task is greater than some execution time threshold [36].

Bryant and Finkel [3] propose another approach based on the reduction in response time that can be obtained for a task by transferring it elsewhere. In this method, a task is selected for transfer only if its response time will be improved upon transfer. (See [3] for details on how to estimate response time.)

There are other factors to consider in the selection of a task. First, the overhead incurred by the transfer should be minimal. For example, a task of small size carries less overhead. Second, the number of location-dependent system calls made by the selected task should be minimal. Location-dependent calls must be executed at the node where the task originated because they use resources such as windows, or the mouse, that only exist at the node [8, 19].

11.4.3 Location Policy

The responsibility of a location policy is to find suitable nodes (senders or receivers) to share load. A widely used method for finding a suitable node is through *polling*. In polling, a node polls another node to find out whether it is a suitable node for load sharing [3, 10, 11, 24, 31]. Nodes can be polled either serially or in parallel (e.g., multicast). A node can be selected for polling either randomly [3, 10, 11], based on the information collected during the previous polls [24, 31], or on a nearest-neighbor basis. An alternative to polling is to broadcast a query to find out if any node is available for load sharing.

11.4.4 Information Policy

The information policy is responsible for deciding when information about the states of other nodes in the system should be collected, where it should be collected from, and what information should be collected. Most information policies are one of the following three types:

Demand-driven. In this class of policy, a node collects the state of other nodes only when it becomes either a sender or a receiver (decided by the transfer and selection policies at the node), making it a suitable candidate to initiate load sharing. Note that a demand-driven information policy is inherently a dynamic policy, as its actions depend on the system state. Demand-driven policies can be *sender-initiated*, *receiver-initiated*, or *symmetrically initiated*. In sender-initiated policies, senders look for receivers to transfer their load. In receiver-initiated policies, receivers solicit load from senders. A symmetrically initiated policy is a combination of both, where load sharing actions are triggered by the demand for extra processing power or extra work.

Periodic. In this class of policy, nodes exchange load information periodically [14, 40]. Based on the information collected, the transfer policy at a node may decide to transfer jobs. Periodic information policies do not adapt their activity to the system state. For example, the benefits due to load distributing are minimal at high system loads because most of the nodes in the system are busy. Nevertheless, overheads due to periodic information collection continue to increase the system load and thus worsen the situation.