CHAPTER 7

Resource Management

7.1 INTRODUCTION

Distributed systems are characterized by resource multiplicity and system transparency. Every distributed system consists of a number of resources interconnected by a network. Besides providing communication facilities, the network facilitates resource sharing by migrating a local process and executing it at a remote node of the network. A process may be migrated because the local node does not have the required resources or the local node has to be shut down. A process may also be executed remotely if the expected turnaround time will be better. From a user's point of view, the set of available resources in a distributed system acts like a single virtual system. Hence, when a user submits a process for execution, it becomes the responsibility of the resource manager of the distributed operating system to control the assignment of resources to processes and to route the processes to suitable nodes of the system according to these assignments. A resource can be logical, such as a shared file, or physical, such as a CPU. For our purpose, we will consider a resource to be a processor of the system and assume that each processor forms a node of the distributed system. Thus, in this chapter, we will be interchangeably using the terms *node* and *processor* to mean the same thing.

A resource manager schedules the processes in a distributed system to make use of the system resources in such a manner that resource usage, response time, network congestion, and scheduling overhead are optimized. A variety of widely differing techniques and methodologies for scheduling processes of a distributed system have been proposed. These techniques can be broadly classified into three types:

1. *Task assignment approach*, in which each process submitted by a user for processing is viewed as a collection of related tasks and these tasks are scheduled to suitable nodes so as to improve performance
2. *Load-balancing approach*, in which all the processes submitted by the users are distributed among the nodes of the system so as to equalize the workload among the nodes
3. *Load-sharing approach*, which simply attempts to conserve the ability of the system to perform work by assuring that no node is idle while processes wait for being processed

Of the three approaches, the task assignment approach has limited applicability in practical situations because it works on the assumption that the characteristics of all the processes to be scheduled are known in advance. Furthermore, the scheduling algorithms that fall in this category do not normally take care of the dynamically changing state of the system. Therefore, this approach will be covered very briefly just to give an idea of how it works. Before presenting a description of each of these techniques, the desirable features of a good global scheduling algorithm are presented.

7.2 DESIRABLE FEATURES OF A GOOD GLOBAL SCHEDULING ALGORITHM

7.2.1 No A Priori Knowledge about the Processes

A good process scheduling algorithm should operate with absolutely no a priori knowledge about the processes to be executed. Scheduling algorithms that operate based on the information about the characteristics and resource requirements of the processes normally pose an extra burden upon the users who must specify this information while submitting their processes for execution.

7.2.2 Dynamic in Nature

It is intended that a good process-scheduling algorithm should be able to take care of the dynamically changing load (or status) of the various nodes of the system. That is, process assignment decisions should be based on the current load of the system and not on some fixed static policy. For this, sometimes it is also recommended that the scheduling algorithm should possess the flexibility to migrate a process more than once because the initial decision of placing a process on a particular node may have to be changed after

some time to adapt to the new system load. This feature may also require that the system support preemptive process migration facility in which a process can be migrated from one node to another during the course of its execution.

7.2.3 Quick Decision-Making Capability

A good process-scheduling algorithm must make quick decisions about the assignment of processes to processors. This is an extremely important aspect of the algorithms and makes many potential solutions unsuitable. For example, an algorithm that models the system by a mathematical program and solves it on-line is unsuitable because it does not meet this requirement. Heuristic methods requiring less computational effort while providing near-optimal results are therefore normally preferable to exhaustive (optimal) solution methods.

7.2.4 Balanced System Performance and Scheduling Overhead

Several global scheduling algorithms collect global state information and use this information in making process assignment decisions. A common intuition is that greater amounts of information describing global system state allow more intelligent process assignment decisions to be made that have a positive affect on the system as a whole. In a distributed environment, however, information regarding the state of the system is typically gathered at a higher cost than in a centralized system. The general observation is that, as overhead is increased in an attempt to obtain more information regarding the global state of the system, the usefulness of that information is decreased due to both the aging of the information being gathered and the low scheduling frequency as a result of the cost of gathering and processing that information. Hence algorithms that provide near-optimal system performance with a minimum of global state information gathering overhead are desirable.

7.2.5 Stability

A scheduling algorithm is said to be unstable if it can enter a state in which all the nodes of the system are spending all of their time migrating processes without accomplishing any useful work in an attempt to properly schedule the processes for better performance. This form of fruitless migration of processes is known as *processor thrashing*. Processor thrashing can occur in situations where each node of the system has the power of scheduling its own processes and scheduling decisions either are made independently of decisions made by other processors or are based on relatively old data due to transmission delay between nodes. For example, it may happen that nodes n_1 and n_2 both observe that node n_3 is idle and then both offload a portion of their work to node n_3 without being aware of the offloading decision made by the other. Now if node n_3 becomes overloaded due to the processes received from both nodes n_1 and n_2, then it may again start transferring its processes to other nodes. This entire cycle may be repeated again and again, resulting in an unstable state. This is certainly not desirable for a good scheduling algorithm.

Processor thrashing can also occur if processes in transit to a lightly loaded node are not taken into account. In this case, several processes may be migrated to the same node, possibly producing instabilities. For example, suppose at a particular instance of time node n_1 is very busy and node n_2 is the least busy node. Also suppose that node n_1 is activated every 2 seconds and on an average it takes 20 seconds for a process to reach node n_2 from node n_1. Then node n_1 could conceivably send at least 10 processes to node n_2 before the first one was received. This could result in an unstable situation. A simple method to overcome this problem is to keep track of which node has been sent processes recently and use this information in an effort to mitigate the problem and to minimize process movement. More sophisticated techniques to deal with stability are possible, but they require the retention of more past data and hence are expensive.

7.2.6 Scalability

A scheduling algorithm should be capable of handling small as well as large networks. An algorithm that makes scheduling decisions by first inquiring the workload from all the nodes and then selecting the most lightly loaded node as the candidate for receiving the process(es) has poor scalability factor. Such an algorithm may work fine for small networks but gets crippled when applied to large networks. This is because the inquirer receives a very large number of replies almost simultaneously, and the time required to process the reply messages for making a host selection is normally too long. Moreover, the N^2 (N is the total number of nodes in the system) nature of the algorithm creates heavy network traffic and quickly consumes network bandwidth. A simple approach to make an algorithm scalable is to probe only m of N nodes for selecting a host. The value of m can be dynamically adjusted depending upon the value of N.

7.2.7 Fault Tolerance

A good scheduling algorithm should not be disabled by the crash of one or more nodes of the system. At any instance of time, it should continue functioning for nodes that are up at that time. Moreover, if the nodes are partitioned into two or more groups due to link failures, the algorithm should be capable of functioning properly for the nodes within a group. Algorithms that have decentralized decision-making capability and consider only available nodes in their decision-making approach have better fault tolerance capability.

7.2.8 Fairness of Service

While the average quality of service provided is clearly an important performance index, how fairly service is allocated is also a common concern. For example, two users simultaneously initiating equivalent processes expect to receive about the same quality of service. Several researchers think that global scheduling policies that blindly attempt to balance the average load on all the nodes of the system are not good from the point of view of fairness of service. This is because in any load-balancing scheme, heavily loaded nodes will obtain all the benefits while lightly loaded nodes will suffer poorer response time than

that in a stand-alone configuration. What is desirable is a fair strategy that will improve response time to the former without unduly affecting the latter. For this, the concept of load balancing has to be replaced by the concept of load sharing, that is, a node will share some of its resources as long as its users are not significantly affected.

7.3 TASK ASSIGNMENT APPROACH

7.3.1 The Basic Idea

In this approach, a process is considered to be composed of multiple tasks and the goal is to find an optimal assignment policy for the tasks of an individual process. Typical assumptions found in task assignment work are as follows:

- A process has already been split into pieces called tasks. This split occurs along natural boundaries, so that each task will have integrity in itself and data transfers among the tasks will be minimized.
- The amount of computation required by each task and the speed of each processor are known.
- The cost of processing each task on every node of the system is known. This cost is usually derived based on the information about the speed of each processor and the amount of computation required by each task.
- The interprocess communication (IPC) costs between every pair of tasks is known. The IPC cost is considered zero (negligible) for tasks assigned to the same node. They are usually estimated by an analysis of the static program of a process. For example, during the execution of the process, if two tasks communicate n times and if the average time for each intertask communication is t, the intertask communication cost for the two tasks is $n \times t$.
- Other constraints, such as resource requirements of the tasks and the available resources at each node, precedence relationships among the tasks, and so on, are also known.
- Reassignment of the tasks is generally not possible.

With these assumptions, the task assignment algorithms seek to assign the tasks of a process to the nodes of the distributed system in such a manner so as to achieve goals such as the following:

- Minimization of IPC costs
- Quick turnaround time for the complete process
- A high degree of parallelism
- Efficient utilization of system resources in general

These goals often conflict with each other. For example, while minimizing IPC tends to assign all the tasks of a process to a single node, efficient utilization of system resources

tries to distribute the tasks evenly among the nodes. Similarly, while quick turnaround time and a high degree of parallelism encourage parallel execution of the tasks, the precedence relationship among the tasks limits their parallel execution. Also notice that in case of m tasks and q nodes, there are m^q possible assignments of tasks to nodes. In practice, however, the actual number of possible assignments of tasks to nodes may be less than m^q due to the restriction that certain tasks cannot be assigned to certain nodes due to their specific resource requirements; the required resource(s) may not be available on all the nodes of the system.

To illustrate with an example, let us consider the assignment problem of Figure 7.1. This is the same problem discussed by Stone [1977]. It involves only two task assignment parameters—the task execution cost and the intertask communication cost. This system is made up of six tasks $\{t_1, t_2, t_3, t_4, t_5, t_6\}$ and two nodes $\{n_1, n_2\}$. The intertask communication costs (c_{ij}) and the execution costs (x_{ab}) of the tasks are given in tabular form in Figure 7.1(a) and (b), respectively. An infinite cost for a particular task against a particular node in Figure 7.1(b) indicates that the task cannot be executed on that node due to the task's requirement of specific resources that are not available on that node. Thus, task t_2 cannot be executed on node n_2 and task t_6 cannot be executed on node n_1. In this model of a distributed computing system, there is no parallelism or multitasking of task execution within a program. Thus the total cost of process execution consists of the total execution cost of the tasks on their assigned nodes plus the intertask communication costs between tasks assigned to different nodes.

Figure 7.1(c) shows a serial assignment of the tasks to the two nodes in which the first three tasks are assigned to node n_1 and the remaining three are assigned to node n_2. Observe that this assignment is aimed at minimizing the total execution cost. But if both the execution costs and the communication costs are taken into account, the total cost for this assignment comes out to be 58. Figure 7.1(d) shows an optimal assignment of the tasks to the two nodes that minimizes total execution and communication costs. In this case, although the execution cost is more than that of the previous assignment, the total assignment cost is only 38.

7.3.2 Finding an Optimal Assignment

The problem of finding an assignment of tasks to nodes that minimizes total execution and communication costs was elegantly analyzed using a network flow model and network flow algorithms by Stone [1977, 1978] and a number of other researchers [Lo 1988, Wu and Liu 1980, Bokhari 1979]. In this approach, an optimal assignment is found by creating a static assignment graph, as shown in Figure 7.2. In this graph, nodes n_1 and n_2 represent the two nodes (processors) of the distributed system and nodes t_1 through t_6 represent the tasks of the process. The weights of the edges joining pairs of task nodes represent intertask communication costs. The weight on the edge joining a task node to node n_1 represents the execution cost of that task on node n_2 and vice versa.

A *cutset* in this graph is defined to be a set of edges such that when these edges are removed, the nodes of the graph are partitioned into two disjoint subsets such that the nodes in one subset are reachable from n_1 and the nodes in the other are reachable from n_2. Each task node is reachable from either n_1 or n_2. No proper subset of a cutset is also

Intertask communications cost						
	t_1	t_2	t_3	t_4	t_5	t_6
t_1	0	6	4	0	0	12
t_2	6	0	8	12	3	0
t_3	4	8	0	0	11	0
t_4	0	12	0	0	5	0
t_5	0	3	11	5	0	0
t_6	12	0	0	0	0	0

(*a*)

Execution costs		
Tasks	Nodes	
	n_1	n_2
t_1	5	10
t_2	2	∞
t_3	4	4
t_4	6	3
t_5	5	2
t_6	∞	4

(*b*)

Serial assignment	
Task	Node
t_1	n_1
t_2	n_1
t_3	n_1
t_4	n_2
t_5	n_2
t_6	n_2

(*c*)

Optimal assignment	
Task	Node
t_1	n_1
t_2	n_1
t_3	n_1
t_4	n_1
t_5	n_1
t_6	n_2

(*d*)

Serial assignment execution cost (x) $= x_{11} + x_{21} + x_{31} + x_{42} + x_{52} + x_{62}$
$= 5 + 2 + 4 + 3 + 2 + 4 = 20$

Serial assignment communication cost (c) $= c_{14} + c_{15} + c_{16} + c_{24} + c_{25} + c_{26} + c_{34} + c_{35} + c_{36}$
$= 0 + 0 + 12 + 12 + 3 + 0 + 0 + 11 + 0 = 38$

Serial assignment total cost $= x + c = 20 + 38 = 58$

Optimal assignment execution cost (x) $= x_{11} + x_{21} + x_{31} + x_{41} + x_{51} + x_{62}$
$= 5 + 2 + 4 + 6 + 5 + 4 = 26$

Optimal assignment communication cost (c) $= c_{16} + c_{26} + c_{36} + c_{46} + c_{56}$
$= 12 + 0 + 0 + 0 + 0 = 12$

Optimal assignment total cost $= x + c = 26 + 12 = 38$

Fig. 7.1 A task assignment problem example. (*a*) intertask communication costs; (*b*) execution costs of the tasks on the two nodes; (*c*) serial assignment; (*d*) optimal assignment.

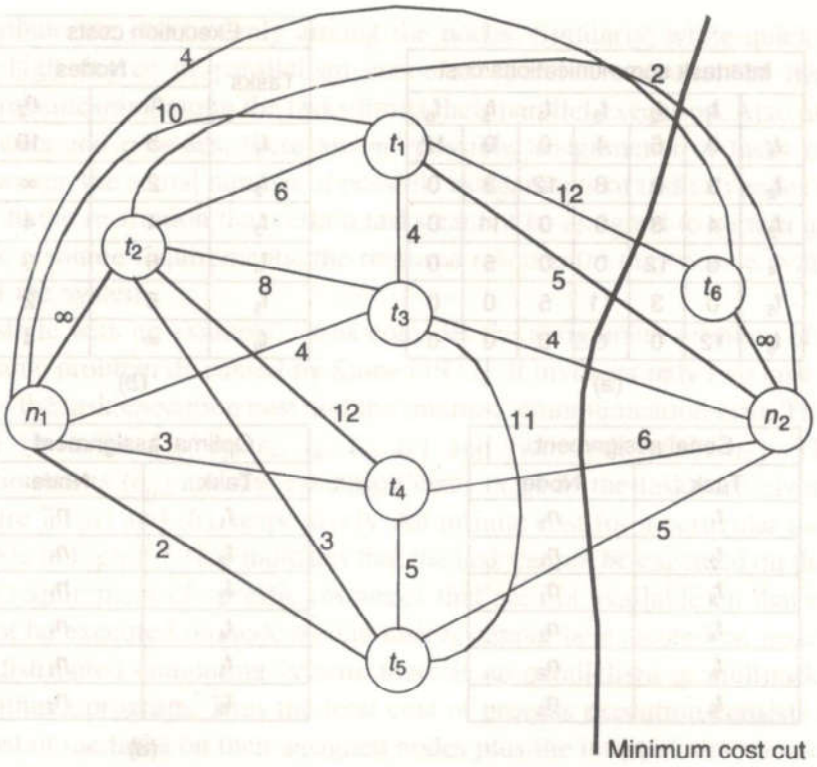

Fig. 7.2 Assignment graph for the assignment problem of Figure 7.1 with minimum cost cut.

a cutset, that is, a cutset is a minimal set. Each cutset corresponds in a one-to-one manner to a task assignment.

The weight of a cutset is the sum of the weights of the edges in the cutset. It represents the cost of the corresponding task assignment since the weight of a cutset sums up the execution and communication costs for that assignment. An optimal assignment may be obtained by finding a minimum-weight cutset. This may be done by the use of network flow algorithms, which are among the class of algorithms with relatively low computational complexity. The bold line in Figure 7.2 indicates a minimum-weight cutset that corresponds to the optimal assignment of Figure 7.1(*d*). Note that if task t_1 is assigned to node n_2, then the edge to node n_2 is cut, but this edge carries the cost of executing task t_1 on node n_1. Similarly, other edges cut between task t_1 and other nodes of the graph represent actual communication costs incurred by this assignment for communication between task t_1 and the other corresponding nodes.

In a two-processor system, an optimal assignment can be found in polynomial time by utilizing Max Flow/Min Cut algorithms [Shen and Tsai 1985]. However, for an arbitrary number of processors, the problem is known to be Nonpolynomial (NP) hard. An NP hard problem is computationally intractable because it cannot be solved in polynomial time. Thus, for more general cases, several researchers have turned to heuristic algorithms that are computationally efficient but may yield suboptimal assignments. Readers interested in some of these heuristic algorithms may refer to [Arora and Rana 1980, Efe 1982, Lo 1988].

It may be noted that in the model described above, the tasks of a process were assigned to the various nodes of the system. This model may be generalized to the general task assignment problem in which several processes are to be assigned. In this case, each process is treated to be a task of the process force and the interprocess communication costs are assumed to be known.

Several extensions to the basic task assignment model described above have been proposed in the literature. In addition to the task assignment cost and the intertask communication cost parameters of the basic task assignment model, the extended models take into account other parameters such as memory size requirements of the task and memory size constraint of the processors, precedence relationship among the tasks, and so on. However, we will not discuss this topic any further because of the limited applicability of the task assignment approach in practical situations. Readers interested in some of these extended models may refer to [Lo 1988, Chu and Lan 1987, Rao et al. 1979].

7.4 LOAD-BALANCING APPROACH

The scheduling algorithms using this approach are known as *load-balancing algorithms* or *load-leveling algorithms*. These algorithms are based on the intuition that, for better resource utilization, it is desirable for the load in a distributed system to be balanced evenly. Thus, a load-balancing algorithm tries to balance the total system load by transparently transferring the workload from heavily loaded nodes to lightly loaded nodes in an attempt to ensure good overall performance relative to some specific metric of system performance. When considering performance from the user point of view, the metric involved is often the response time of the processes. However, when performance is considered from the resource point of view, the metric involved is the total system throughput. In contrast to response time, throughput is concerned with seeing that all users are treated fairly and that all are making progress. Notice that the resource view of maximizing resource utilization is compatible with the desire to maximize system throughput. Thus the basic goal of almost all the load-balancing algorithms is to maximize the total system throughput.

7.4.1 A Taxonomy of Load-Balancing Algorithms

The taxonomy presented here is a hierarchy of the features of load-balancing algorithms. The structure of the taxonomy is shown in Figure 7.3. To describe a specific load-balancing algorithm, a taxonomy user traces paths through the hierarchy. A description of this taxonomy is given below.

Static versus Dynamic

At the highest level, we may distinguish between *static* and *dynamic* load-balancing algorithms. Static algorithms use only information about the average behavior of the system, ignoring the current state of the system. On the other hand, dynamic algorithms react to the system state that changes dynamically.

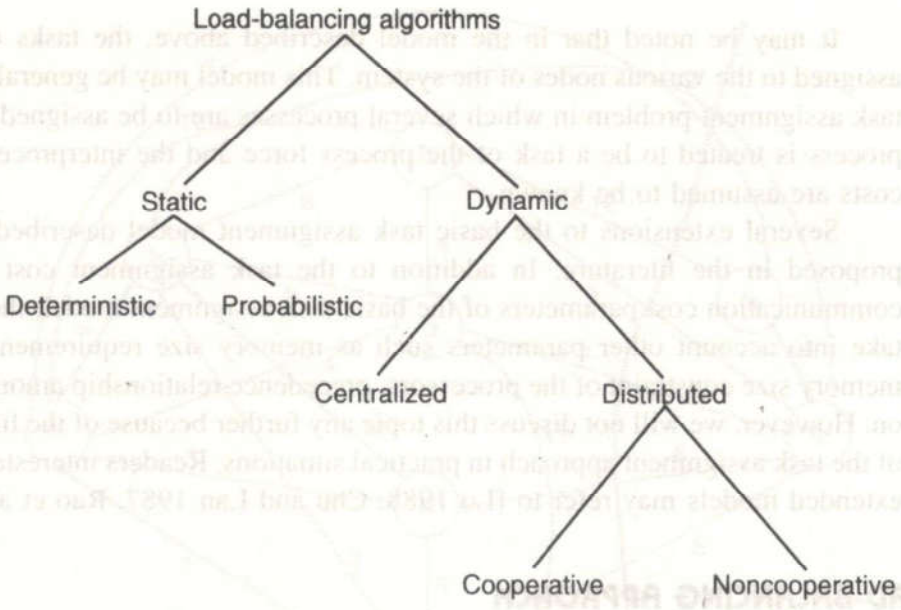

Fig. 7.3 A taxonomy of load-balancing algorithms.

Static load-balancing algorithms are simpler because there is no need to maintain and process system state information. However, the potential of static algorithms is limited by the fact that they do not react to the current system state. The attraction of dynamic algorithms is that they do respond to system state and so are better able to avoid those states with unnecessarily poor performance. Owing to this reason, dynamic policies have significantly greater performance benefits than static policies. However, since dynamic algorithms must collect and react to system state information, they are necessarily more complex than static algorithms.

Deterministic versus Probabilistic

Static load-balancing algorithms may be either *deterministic* or *probabilistic*. Deterministic algorithms use the information about the properties of the nodes and the characteristics of the processes to be scheduled to deterministically allocate processes to nodes. Notice that the task assignment algorithms basically belong to the category of deterministic static load-balancing algorithms.

A probabilistic load-balancing algorithm uses information regarding static attributes of the system such as number of nodes, the processing capability of each node, the network topology, and so on, to formulate simple process placement rules. For example, suppose a system has two processors p_1 and p_2 and four terminals t_1, t_2, t_3, and t_4. Then a simple process placement rule can be, assign all processes originating at terminals t_1 and t_2 to processor p_1 and processes originating at terminals t_3 and t_4 to processor p_2. Obviously, such static load-balancing algorithms have limited potential to avoid those states that have unnecessarily poor performance. For example, suppose at the time a particular process originates at terminal t_1 processor p_1 is very heavily loaded and processor p_2 is idle. Certainly, processor p_2 is a better choice for the process in such a situation.

In general, the deterministic approach is difficult to optimize and costs more to implement. The probabilistic approach is easier to implement but often suffers from having poor performance.

Centralized versus Distributed

Dynamic scheduling algorithms may be *centralized* or *distributed*. In a centralized dynamic scheduling algorithm, the responsibility of scheduling physically resides on a single node. On the other hand, in a distributed dynamic scheduling algorithm, the work involved in making process assignment decisions is physically distributed among the various nodes of the system. In the centralized approach, the system state information is collected at a single node at which all scheduling decisions are made. This node is called the *centralized server node*. All requests for process scheduling are handled by the centralized server, which decides about the placement of a new process using the state information stored in it. The centralized approach can efficiently make process assignment decisions because the centralized server knows both the load at each node and the number of processes needing service. In the basic method, the other nodes periodically send status update messages to the central server node. These messages are used to keep the system state information up to date at the centralized server node. One might consider having the centralized server query the other nodes for state information. This would reduce message traffic if state information was used to answer several process assignment requests, but since nodes can change their load any time due to local activities, this would introduce problems of stale state information.

A problem associated with the centralized mechanism is that of reliability. If the centralized server fails, all scheduling in the system would cease. A typical approach to overcome this problem would be to replicate the server on $k+1$ nodes if it is to survive k faults (node failures). In this approach, the overhead involved in keeping consistent all the $k+1$ replicas of the server may be considerably high. A typical solution to reduce this overhead is to forgo strong consistency and use a cheaper mechanism to update the multiple copies of the server. This solution is based on the idea that strict consistency is not necessary in this case for the system to function properly.

Theimer and Lantz [1989] proposed that if occasional delays in service of several seconds are acceptable, one can use the simpler approach of *reinstantiation* to improve the reliability of the centralized scheduling mechanism. In this approach, rather than maintaining $k+1$ server replicas, a single server is maintained and there are k entities monitoring the server to detect its failure. When failure is detected, a new instance of the server is brought up, which reconstructs its state information by sending a multicast message requesting immediate state update. The time during which the scheduling service is unavailable will be the sum of the time to detect failure of the previous server, the time to load the server program on the new server, the time to resolve the possibility of multiple concurrent instantiations among the k entities, and the time to reconstruct the global state information on the new server [Theimer and Lantz 1989].

In contrast to the centralized scheme, a distributed scheme does not limit the scheduling intelligence to one node. It avoids the bottleneck of collecting state information at a single node and allows the scheduler to react quickly to dynamic changes in the system

state. A distributed dynamic scheduling algorithm is composed of k physically distributed entities $e_1, e_2, \ldots, e_k$. Each entity is considered a local controller. Each local controller runs asynchronously and concurrently with the others, and each is responsible for making scheduling decisions for the processes of a predetermined set of nodes. Each entity e_i makes decisions based on a systemwide objective function, rather than on a local one. Each e_i makes decisions on an equal basis with the other entities; that is, there is no master entity, even for a short period of time. In a fully distributed algorithm, each node has its own entity and hence $k = N$ for a system having N nodes. In this case, each entity is responsible for making scheduling decisions for the processes of its own node, which includes both transfer of local processes and acceptance of remote processes.

Cooperative versus Noncooperative

Distributed dynamic scheduling algorithms may be categorized as *cooperative* and *noncooperative*. In noncooperative algorithms, individual entities act as autonomous entities and make scheduling decisions independently of the actions of other entities. On the other hand, in cooperative algorithms, the distributed entities cooperate with each other to make scheduling decisions. Hence, cooperative algorithms are more complex and involve larger overhead than noncooperative ones. However, the stability of a cooperative algorithm is better than that of a noncooperative algorithm.

7.4.2 Issues in Designing Load-Balancing Algorithms

Designing a good load-balancing algorithm is a difficult task because of the following issues:

- Load estimation policy, which determines how to estimate the workload of a particular node of the system
- Process transfer policy, which determines whether to execute a process locally or remotely
- State information exchange policy, which determines how to exchange the system load information among the nodes
- Location policy, which determines to which node a process selected for transfer should be sent
- Priority assignment policy, which determines the priority of execution of local and remote processes at a particular node
- Migration limiting policy, which determines the total number of times a process can migrate from one node to another

These issues are discussed below. For this discussion, we divide the processes within the system into two classes: local processes and remote processes. A *local process* is one that is processed at its originating node and a *remote process* is one that is processed at a node different from the one on which it originated. A new process, arriving from the external world at a node, becomes a local process if it is admitted to that node for

processing. Otherwise, it is transferred across the network and becomes a remote process at the destination node.

Load Estimation Policies

The main goal of load-balancing algorithms is to balance the workload on all the nodes of the system. However, before an algorithm can attempt to balance the workload, it is necessary to decide how to measure the workload of a particular node. Hence the first issue in any load-balancing algorithm is to decide on the method to be used to estimate the workload of a particular node. Estimation of the workload of a particular node is a difficult problem for which no completely satisfactory solution exists. A node's workload can be estimated based on some measurable parameters. These parameters could include time-dependent and node-dependent factors such as the following:

- Total number of processes on the node at the time of load estimation
- Resource demands of these processes
- Instruction mixes of these processes
- Architecture and speed of the node's processor

Since the measurement of load would occur quite often and the load would reflect the current state of the node, its calculation must be very efficient. This rules out an exhaustive use of all parameters even if their relative importance is known. Thus several load-balancing algorithms use the total number of processes present on the node as a measure of the node's workload. However, several designers believe that this is an unsuitable measure for such an estimate since the true load could vary widely depending on the remaining service times for those processes. Therefore another measure used for estimating a node's workload is the sum of the remaining service times of all the processes on that node. However, in this case another issue that must be resolved is how to estimate the remaining service time of the processes. Bryant and Finkel [1981] have proposed the use of one of the following methods for this purpose:

1. *Memoryless method.* This method assumes that all processes have the same expected remaining service time, independent of the time used so far. The use of this method for remaining service time estimation of a process basically reduces the load estimation method to that of total number of processes.
2. *Pastrepeats.* This method assumes that the remaining service time of a process is equal to the time used so far by it.
3. *Distribution method.* If the distribution of service times is known, the associated process's remaining service time is the expected remaining time conditioned by the time already used.

Neither the method of counting the total number of processes nor the method of taking the sum of the remaining service times of all processes is suitable for use as load estimation policies in modern distributed systems. This is because in modern distributed

systems, even on an idle node, several processes such as mail and news daemons, window managers, and so on, exist permanently. Moreover, many of these daemon processes wake up periodically, check to see if there is anything that they have to do, and if not, go back to sleep. Therefore, an acceptable method for use as the load estimation policy in these systems would be to measure the CPU utilization of the nodes [Tanenbaum 1995]. *Central processing unit (CPU) utilization* is defined as the number of CPU cycles actually executed per unit of real time. Obviously, the CPU utilization of a heavily loaded node will be greater than the CPU utilization of a lightly loaded node. The CPU utilization of a node can be measured by setting up a timer to periodically observe the CPU state (idle/busy).

Process Transfer Policies

The strategy of load-balancing algorithms is based on the idea of transferring some processes from the heavily loaded nodes to the lightly loaded nodes for processing. However, to facilitate this, it is necessary to devise a policy to decide whether a node is lightly or heavily loaded. Most of the load-balancing algorithms use the *threshold policy* to make this decision. The threshold value of a node is the limiting value of its workload and is used to decide whether a node is lightly or heavily loaded. Thus a new process at a node is accepted locally for processing if the workload of the node is below its threshold value at that time. Otherwise, an attempt is made to transfer the process to a lightly loaded node. The threshold value of a node may be determined by any of the following methods:

1. *Static policy.* In this method, each node has a predefined threshold value depending on its processing capability. This threshold value does not vary with the dynamic changes in workload at local or remote nodes. The main advantage of this method is that no exchange of state information among the nodes is required in deciding the threshold value.

2. *Dynamic policy.* In this method, the threshold value of a node (n_i) is calculated as a product of the average workload of all the nodes and a predefined constant (c_i). For each node n_i, the value of c_i depends on the processing capability of node n_i relative to the processing capability of all other nodes. In this method, the nodes exchange state information by using one of the state information exchange policies (described later). Although the dynamic policy gives a more realistic value of threshold for each node, the overhead involved in exchange of state information makes its applicability questionable.

Most load-balancing algorithms use a single threshold and thus only have overloaded and underloaded regions [Fig. 7.4(a)]. In this single-threshold policy, a node accepts new processes (either local or remote) if its load is below the threshold value and attempts to transfer local processes and rejects remote execution requests if its load is above the threshold value. Therefore the decision regarding both the transfer of local processes and the acceptance of remote processes is done based on a single-threshold value. The use of

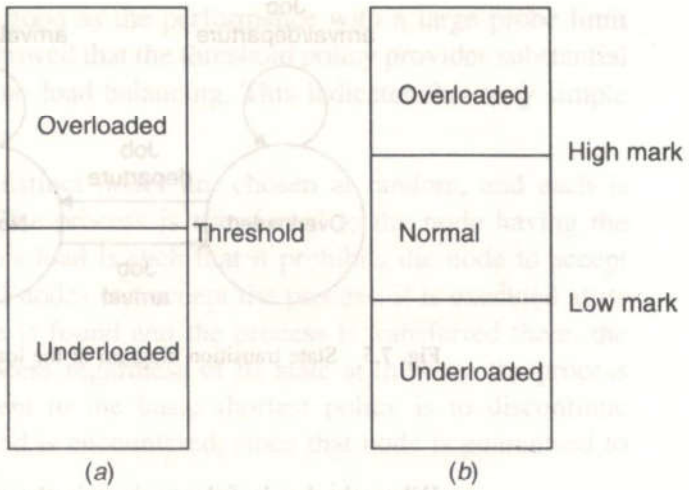

Fig. 7.4 The load regions of (*a*) single-threshold policy and (*b*) double-threshold policy.

a single-threshold value may lead to fruitless process transfers, making the scheduling algorithm unstable because a node's load may be below the threshold when it decides to accept a remote process, but its load may become larger than the threshold as soon as the remote process arrives. Therefore, immediately after receiving the remote process, the node will again try to transfer one or more of its processes to some other node. Moreover, Alonso and Cova [1988] observed:

- A node should only transfer one or more of its processes to another node if such a transfer greatly improves the performance of the rest of its local processes.
- A node should accept remote processes only if its load is such that the added workload of processing these incoming processes does not significantly affect the service to the local ones.

To reduce the instability of the single-threshold policy and to take care of these two notions, Alonso and Cova [1988] proposed a double-threshold policy called the *high-low policy*. As shown in Figure 7.4(*b*), the high-low policy uses two threshold values called *high mark* and *low mark*, which divide the space of possible load states of a node into the following three regions:

- Overloaded—above the high-mark and low-mark values
- Normal—above the low-mark value and below the high-mark value
- Underloaded—below both values

A node's load state switches dynamically from one region to another, as shown in Figure 7.5.

Now depending on the current load status of a node, the decision to transfer a local process or to accept a remote process is based on the following policies [Alonso and Cova 1988]:

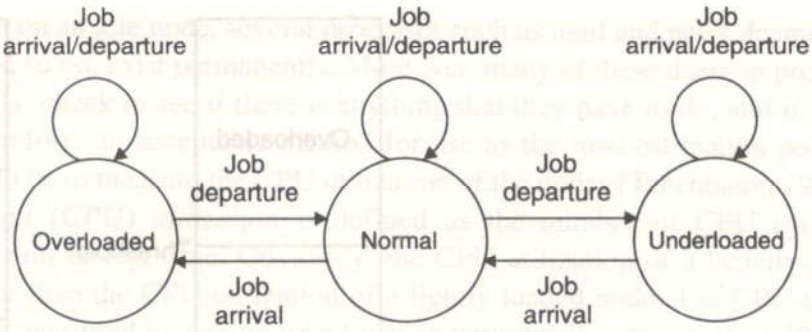

Fig. 7.5 State transition diagram of the load of a node in case of double-threshold policy.

- When the load of the node is in the overloaded region, new local processes are sent to be run remotely and requests to accept remote processes are rejected.
- When the load of the node is in the normal region, new local processes run locally and requests to accept remote processes are rejected.
- When the load of the node is in the underloaded region, new local processes run locally and requests to accept remote processes are accepted.

Notice that the high-low policy guarantees a predefined level of performance to the node owners. It accounts for the overhead that the load-balancing algorithm may incur in transferring and receiving a remote process. A process will not be transferred to another node unless it is worthwhile, and a remote process will not be accepted unless there is enough excess capacity to handle it [Alonso and Cova 1988].

Location Policies

Once a decision has been made through the transfer policy to transfer a process from a node, the next step is to select the destination node for that process's execution. This selection is made by the location policy of a scheduling algorithm. The main location policies proposed in the literature are described below.

Threshold. In this method, a destination node is selected at random and a check is made to determine whether the transfer of the process to that node would place it in a state that prohibits the node to accept remote processes. If not, the process is transferred to the selected node, which must execute the process regardless of its state when the process actually arrives. On the other hand, if the check indicates that the selected node is in a state that prohibits it to accept remote processes, another node is selected at random and probed in the same manner. This continues until either a suitable destination node is found or the number of probes exceeds a static probe limit L_p. In the latter case, the node on which the process originated must execute the process. Eager et al. [1986b] performed simulations by using the single static threshold transfer policy and this threshold location policy. Their simulation results showed that the performance of this policy is surprisingly insensitive to the choice of probe limit; the performance with a small (and economical)

probe limit (e.g., 3 or 5) is almost as good as the performance with a large probe limit (e.g., 20). The simulation results also showed that the threshold policy provides substantial performance improvement relative to no load balancing. This indicates that very simple schemes can yield significant benefits.

Shortest. In this method, L_p distinct nodes are chosen at random, and each is polled in turn to determine its load. The process is transferred to the node having the minimum load value, unless that node's load is such that it prohibits the node to accept remote processes. If none of the polled nodes can accept the process, it is executed at its originating node. If a destination node is found and the process is transferred there, the destination node must execute the process regardless of its state at the time the process actually arrives. A simple improvement to the basic shortest policy is to discontinue probing whenever a node with zero load is encountered, since that node is guaranteed to be an acceptable destination.

The shortest policy uses more state information, in a more complex manner, than does the threshold policy. But Eager et al. [1986b] found, through their simulation results, that the performance of the shortest policy is not significantly better than that of the simpler threshold policy. This suggests that state information beyond that used by the threshold policy or a more complex usage of state information is of little benefit.

Bidding. In this method, the system is turned into a distributed computational economy with buyers and sellers of services [Waldspurger et al. 1992, Malone et al. 1988]. Each node in the network is responsible for two roles with respect to the bidding process: manager and contractor. The *manager* represents a node having a process in need of a location to execute, and the *contractor* represents a node that is able to accept remote processes. Note that a single node takes on both these roles and no nodes are strictly managers or contractors alone. To select a node for its process, the manager broadcasts a request-for-bids message to all other nodes in the system. Upon receiving this message, the contractor nodes return bids to the manager node. The bids contain the quoted prices, which vary based on the processing capability, memory size, resource availability, and so on, of the contractor nodes. Of the bids received from the contractor nodes, the manager node chooses the best bid. The best bid for a manager's request may mean the cheapest, fastest, or best price-performance, depending on the application for which the request was made. Once the best bid is determined, the process is transferred from the manager node to the winning contractor node. But it is possible that a contractor node may simultaneously win many bids from many other manager nodes and thus become overloaded. To prevent this situation, when the best bid is selected, a message is sent to the owner of that bid. At that point the bidder may choose to accept or reject that process. A message is sent back to the concerned manager node informing it as to whether the process has been accepted or rejected. A contractor node may reject a winning bid because of changes in its state between the time the bid was made and the time it was notified that it won the bid. If the bid is rejected, the bidding procedure is started all over again.

Bidding algorithms are interesting because they provide full autonomy to the nodes to decide whether to participate in the global scheduling process. For example, a manager node has the power to decide whether to send a process to a contractor node, which responds with bids, and a contractor node has the power to decide whether it wants to

accept remote processes. A contractor node is never forced to accept remote processes if it does not choose to do so. On the other hand, the two main drawbacks of bidding algorithms are that they create a great deal of communication overhead and it is very difficult to decide a good pricing policy. Both factors call for a proper choice of the amount and type of information exchanged during bidding. For a bidding algorithm, the amount and type of information exchanged are generally decided in such a manner so as to balance the effectiveness and performance of the algorithm. A variety of possibilities exist concerning the type and amount of information exchanged in order to make decisions [Stankovic and Sidhu 1984, Smith 1980, Hwang et al. 1982, Stankovic 1984].

Pairing. The method of accomplishing load balancing employed by the policies described until now is to balance or reduce the variance between the loads experienced by all the nodes of the system. Contrary to this approach, the method of accomplishing load balancing by the pairing policy is to reduce the variance of loads only between pairs of nodes of the system. This location policy was proposed by Bryant and Finkel [1981]. In this method, two nodes that differ greatly in load are temporarily paired with each other, and the load-balancing operation is carried out between the nodes belonging to the same pair by migrating one or more processes from the more heavily loaded node to the other node. Several node pairs may exist simultaneously in the system. A node only tries to find a partner if it has at least two processes; otherwise migration from this node is never reasonable. However, every node is willing to respond favorably to a pairing request.

In the basic pairing method, each node asks some randomly chosen node if it will pair with it. While awaiting an answer, the querier rejects any queries from other nodes. If it receives a rejection, it randomly selects another node and tries to pair again. If it receives a query from its own intended partner, a pair is formed. After the formation of a pair, one or more processes are migrated from the more heavily loaded node of the two nodes to the other node in order to balance the loads on these two nodes. The processes to be migrated are selected by comparing their expected time to complete on their current node with the expected time to complete on its partner. Migration delay is included in this estimate. The process with the best ratio of service time on the partner node to service time on the current node is selected to be sent first. Decisions for migrating other processes are based on the assumption that the first process has been received by the partner node and the load of the partner node has been updated to reflect the presence of this process on it. The pair is broken as soon as the process migration is over. During the time that a pair is in force, both members of the pair reject other pairing queries. Some other variations of the pairing mechanism can be found in [Bryant and Finkel 1981].

State Information Exchange Policies

We have seen that the dynamic policies require frequent exchange of state information among the nodes of the system. In fact, a dynamic load-balancing algorithm faces a transmission dilemma because of the two opposing impacts the transmission of a message has on the overall performance of the system. On the one hand, the transmission improves the ability of the algorithm to balance the load. On the other hand, it raises the expected queuing time of messages because of the increase in the

utilization of the communication channel. Thus proper selection of the state information exchange policy is essential. The proposed load-balancing algorithms use one of the following policies for this purpose.

Periodic Broadcast. In this method each node broadcasts its state information after the elapse of every t units of time. Obviously this method is not good because it generates heavy network traffic and also because there is a possibility of fruitless messages (messages from those nodes whose state has not changed during the last t units of time) being broadcast. Furthermore, the scalability of this method is poor because the number of messages generated for state information exchanges will be too large for networks having many nodes.

Broadcast When State Changes. This method avoids the problem of fruitless message exchanges of the periodic broadcast method by ensuring that a node broadcasts its state information only when the state of the node changes. A node's state changes when a process arrives at that node or when a process departs from that node. A process may arrive at a node either from the external world or from some other node in the system. A process departs from a node when either its execution is over or it is transferred to some other node.

A further improvement in this method can be obtained by observing that it is not necessary to report every small change in the state of a node to all other nodes because a node can participate in the load-balancing process only when it is either underloaded or overloaded. Therefore in the refined method, a node broadcasts its state information only when its state switches from the normal load region to either the underloaded region or the overloaded region. Obviously this refined method works only with the two-threshold transfer policy.

On-Demand Exchange. A node needs to know about the state of other nodes only when it is either underloaded or overloaded. The method of on-demand exchange of state information is based on this observation. In this method a node broadcasts a *StateInformationRequest* message when its state switches from the normal load region to either the underloaded region or the overloaded region. On receiving this message, other nodes send their current state to the requesting node. Notice that this method also works only with the two-threshold transfer policy.

Further reduction in the number of messages is possible in this method by observing that the requesting node does not need the state information of all other nodes. Rather, the state information of only those nodes is useful for the requesting node, which can cooperate with it in the load-balancing process. That is, if the requesting node is underloaded, only overloaded nodes can cooperate with it in the load-balancing process and vice versa. Therefore, in the improved on-demand exchange policy, the status of the requesting node is included in the *StateInformationRequest* message. On receiving the *StateInformationRequest* message, only those nodes reply that can cooperate with the requesting node in the load-balancing process. Other nodes do not send any reply.

Exchange by Polling. All the methods described above use the method of broadcasting due to which their scalability is poor. The polling mechanism overcomes this

limitation by avoiding the use of broadcast protocol. This method is based on the idea that there is no need for a node to exchange its state information with all other nodes in the system. Rather, when a node needs the cooperation of some other node for load balancing, it can search for a suitable partner by randomly polling the other nodes one by one. Therefore state information is exchanged only between the polling node and the polled nodes. The polling process stops either when a suitable partner is found or a predefined poll limit is reached.

Priority Assignment Policies

When process migration is supported by a distributed operating system, it becomes necessary to devise a priority assignment rule for scheduling both local and remote processes at a particular node. One of the following priority assignment rules may be used for this purpose:

1. *Selfish*. Local processes are given higher priority than remote processes.
2. *Altruistic*. Remote processes are given higher priority than local processes.
3. *Intermediate*. The priority of processes depends on the number of local processes and the number of remote processes at the concerned node. If the number of local processes is greater than or equal to the number of remote processes, local processes are given higher priority than remote processes. Otherwise, remote processes are given higher priority than local processes.

Lee and Towsley [1986] studied the effect of these three priority assignment policies on the overall response time performance. The results of their study show the following:

- The selfish priority assignment rule yields the worst response time performance of the three policies. This is due to the extremely poor performance of remote processes under this policy. However, it yields the best response time performance for local processes. Consequently, this policy imposes a severe penalty for processes that arrive at a busy node and is beneficial for processes arriving at a lightly loaded node.
- The altruistic priority assignment rule achieves the best response time performance of the three policies. However, under this policy remote processes incur lower delays than local processes, which is somewhat unrealistic in the sense that local processes are the principal workload at each node while remote processes are secondary workload.
- The performance of the intermediate priority assignment rule falls between the other two policies. Interestingly enough, the overall response time performance of this policy is much closer to that of the altruistic policy. Under this policy, local processes are treated better than remote processes for a wide range of system utilizations.

Migration-Limiting Policies

Another important policy to be used by a distributed operating system that supports process migration is to decide about the total number of times a process should be allowed to migrate. One of the following two policies may be used for this purpose.

Uncontrolled. In this case, a remote process arriving at a node is treated just as a process originating at the node. Therefore, under this policy, a process may be migrated any number of times. This policy has the unfortunate property of causing instability.

Controlled. To overcome the instability problem of the uncontrolled policy, most systems treat remote processes different from local processes and use a *migration count* parameter to fix a limit on the number of times that a process may migrate. Several system designers feel that process migration is an expensive operation and hence a process should not be allowed to migrate too frequently. Hence this group of designers normally favors an irrevocable migration policy. That is, the upper limit of the value of migration count is fixed to 1, and hence a process cannot be migrated more than once under this policy. However, some system designers feel that multiple process migrations, especially for long processes, may be very useful for adapting to the dynamically changing states of the nodes. Thus this group of designers sets the upper limit of the value of migration count to some value $k > 1$. The value of k may be decided either statically or dynamically. Its value may also be different for processes having different characteristics. For example, a long process (a process whose execution time is large) may be allowed to migrate more times as compared to a short process.

7.5 LOAD-SHARING APPROACH

Several researchers believe that load balancing, with its implication of attempting to equalize workload on all the nodes of the system, is not an appropriate objective. This is because the overhead involved in gathering state information to achieve this objective is normally very large, especially in distributed systems having a large number of nodes. Moreover, load balancing in the strictest sense is not achievable because the number of processes in a node is always fluctuating and the temporal unbalance among the nodes exists at every moment, even if the static (average) load is perfectly balanced [Livny and Melman 1982]. In fact, for the proper utilization of the resources of a distributed system, it is not required to balance the load on all the nodes. Rather, it is necessary and sufficient to prevent the nodes from being idle while some other nodes have more than two processes. Therefore this rectification is often called *dynamic load sharing* instead of *dynamic load balancing*.

7.5.1 Issues in Designing Load-Sharing Algorithms

Similar to the load-balancing algorithms, the design of a load-sharing algorithm also requires that proper decisions be made regarding load estimation policy, process transfer policy, state information exchange policy, location policy, priority assignment policy, and

migration limiting policy. However, as compared to load balancing, it is simpler to decide about most of these policies in the case of load sharing. This is because, unlike load-balancing algorithms, load-sharing algorithms do not attempt to balance the average workload on all the nodes of the system. Rather, they only attempt to ensure that no node is idle when a node is heavily loaded. The priority assignment policies and the migration limiting policies for load-sharing algorithms are the same as that for the load-balancing algorithms. Hence their description will not be repeated again. Other policies for load sharing are described below.

Load Estimation Policies

Since load-sharing algorithms simply attempt to ensure that no node is idle while processes wait for service at some other node, it is sufficient to know whether a node is busy or idle. Thus load-sharing algorithms normally employ the simplest load estimation policy of counting the total number of processes on a node.

Once again the simple load estimation policy of counting the total number of processes on a node is not suitable for use in modern distributed systems because of the permanent existence of several processes on an idle node. Therefore, measuring CPU utilization should be used as a method of load estimation in these systems.

Process Transfer Policies

Since load-sharing algorithms are normally interested only in the busy or idle states of a node, most of them employ the *all-or-nothing strategy*. This strategy uses the single-threshold policy with the threshold value of all the nodes fixed at 1. That is, a node becomes a candidate for accepting a remote process only when it has no process, and a node becomes a candidate for transferring a process as soon as it has more than one process. Krueger and Livny [1987] pointed out that the *all-or-nothing* strategy is not good in the sense that a node that becomes idle is unable to immediately acquire new processes to execute even though processes wait for service at other nodes, resulting in a loss of available processing power in the system. They suggested that, to avoid this loss, anticipatory transfers to nodes that are not idle but are expected to soon become idle are necessary. Thus to take care of this loss, some load-sharing algorithms use a threshold value of 2 instead of 1.

Notice here that if the measure of CPU utilization is used as the load estimation policy, the high-low policy (as discussed for load balancing) should be used as the process transfer policy.

Location Policies

In load-sharing algorithms, the location policy decides the sender node or the receiver node of a process that is to be moved within the system for load sharing. Depending on the type of node that takes the initiative to globally search for a suitable node for the process, the location policies are of the following types:

- *Sender-initiated policy*, in which the sender node of the process decides where to send the process
- *Receiver-initiated policy*, in which the receiver node of the process decides from where to get the process

Each of these policies are described below along with their relative merits and demerits.

Sender-Initiated Location Policy. In the sender-initiated location policy, heavily loaded nodes search for lightly loaded nodes to which work may be transferred. That is, in this method, when a node's load becomes more than the threshold value, it either broadcasts a message or randomly probes the other nodes one by one to find a lightly loaded node that can accept one or more of its processes. A node is a viable candidate for receiving a process from the sender node only if the transfer of the process to that node would not increase the receiver node's load above its threshold value. In the broadcast method, the presence or absence of a suitable receiver node is known as soon as the sender node receives reply messages from the other nodes. On the other hand, in the random probing method, the probing continues until either a suitable receiver node is found or the number of probes reaches a static probe limit, as L_p. If a suitable receiver node is not found, the node on which the process originated must execute that process. The method of probing with a fixed limit has better scalability than the broadcast method because it ensures that the cost of executing the load-sharing policy will not be prohibitive even in large networks. Eager et al. [1986a] found through their analysis that the performance of this policy is insensitive to the choice of probe limit: the performance with a small probe limit (e.g., 3 or 5) is nearly as good as the performance with a large probe limit (e.g., 20).

Receiver-Initiated Location Policy. In the receiver-initiated location policy, lightly loaded nodes search for heavily loaded nodes from which work may be transferred. That is, in this method, when a node's load falls below the threshold value, it either broadcasts a message indicating its willingness to receive processes for executing or randomly probes the other nodes one by one to find a heavily loaded node that can send one or more of its processes. A node is a viable candidate for sending one of its processes only if the transfer of the process from that node would not reduce its load below the threshold value. In the broadcast method, a suitable node is found as soon as the receiver node receives reply messages from the other nodes. On the other hand, in the random probing method, the probing continues until either a node is found from which a process can be obtained or the number of probes reaches a static probe limit, L_p. In the latter case, the node waits for a fixed timeout period before attempting again to initiate a transfer.

It may be noted here that, in sender-initiated algorithms, scheduling decisions are usually made at process arrival epochs (or a subset thereof), whereas in receiver-initiated algorithms, scheduling decisions are usually made at process departure epochs (or a subset thereof). Owing to this, receiver-initiated policies typically require the preemptive transfer of processes while sender-initiated policies can work even with those systems that do not support preemptive process migration facility. A *preemptive process migration* facility allows the transfer of an executing process from one node to another. On the other hand,

in systems supporting only *non-preemptive process migration* facility, a process can only be transferred prior to beginning its execution. Preemptive process migration is costlier than non-preemptive process migration since the process state, which must accompany the process to its new node, becomes much more complex after execution begins. Receiver-initiated process transfers are mostly preemptive since it is unlikely that a receiver node would open negotiation with a potential sender at the moment that a new process arrived at the sender node. Process transfers that are sender initiated, however, may be either non-preemptive or preemptive, depending on which process the sender chooses to transfer.

Eager et al. [1986a] used simple analytical models for comparing the performance of the two policies relative to each other and to that of a system in which there is no load sharing. Their results, which are valid over a wide range of system parameters, show the following [Eager et al. 1986a]:

- Both sender-initiated and receiver-initiated policies offer substantial performance advantages over the situation in which no load sharing is attempted.
- Sender-initiated policies are preferable to receiver-initiated policies at light to moderate system loads. This is because in a lightly loaded system the receiver-initiated policy generates a large number of probe messages because all the free nodes desperately hunt for work.
- Receiver-initiated policies are preferable at high system loads, but only if the costs of process transfer under the two strategies are comparable. This is because the receiver-initiated policy does not put extra load on the system at critical times (when the system is heavily loaded), but the sender-initiated policy generates a large number of probe messages precisely when the system is heavily loaded and can least tolerate extra load generated by probe messages.
- If the cost of process transfer under receiver-initiated policies is significantly greater than under the sender-initiated policies due to the preemptive transfer of processes, sender-initiated policies provide uniformly better performance.

State Information Exchange Policies

Since load-sharing algorithms do not aim at equalizing the average load on all the nodes, it is not necessary for the nodes to periodically exchange the state information with each other. Rather, a node needs to know the state of other nodes only when it is either underloaded or overloaded. Therefore, in load-sharing algorithms, a node normally exchanges state information with other nodes only when its state changes. The two commonly used policies for this purpose are described below.

Broadcast When State Changes. In this method, a node broadcasts a *StateInformationRequest* message when it becomes either underloaded or overloaded. Obviously, in the sender-initiated policy, a node broadcasts this message only when it becomes overloaded, and in the receiver-initiated policy, this message is broadcast by a node when it becomes underloaded. In receiver-initiated policies that use a fixed threshold value of 1, this method of state information exchange is called *broadcast-when-idle* policy.

Poll When State Changes. Since a mechanism that uses broadcast protocol is unsuitable for large networks, the polling mechanism is normally used in such systems. In this method, when a node's state changes, it does not exchange state information with all other nodes but randomly polls the other nodes one by one and exchanges state information with the polled nodes. The state exchange process stops either when a suitable node for sharing load with the probing node has been found or the number of probes has reached the probe limit. Obviously, in sender-initiated policy, polling is done by a node when it becomes overloaded, and in receiver-initiated policy, polling is done by a node when it becomes underloaded. In receiver-initiated policies that use a fixed threshold value of 1, this method of state information exchange is called *poll-when-idle* policy.

7.6 SUMMARY

A resource manager of a distributed operating system schedules the processes in a distributed system to any one or more or a pool of free resources that can optimize a combination of resource usage, response time, network congestion, and scheduling overhead. The process scheduling decisions are based on such factors as the resource requirements of the processes, the availability of the various resources on different nodes of the system, and the static and/or dynamic state information of the various nodes of the system. A good global scheduling algorithm should possess features such as having no a priori knowledge about the processes, being dynamic in nature, and having quick decision-making capability, balanced system performance and scheduling efficiency, stability, scalability, fault tolerance, and fairness of service. The three different approaches used for the design of global scheduling algorithms are the task assignment approach, the load-balancing approach, and the load-sharing approach.

In the task assignment approach, the process assignment decisions are basically based on a priori knowledge of the characteristics of both the processes to be executed and the system. The basic task assignment model deals with the assignment of tasks to the various nodes of the system in such a manner so as to minimize interprocess communication costs and improve the turnaround time for the complete task force. Some extensions to the basic task assignment model also consider factors such as interference costs, precedence relationships, or memory size constraints. The task assignment approach has limited applicability because it works on the assumption that the characteristics of all the processes to be scheduled are known in advance and also because task assignment algorithms are generally not dynamic in nature.

In the load-balancing approach, the process assignment decisions attempt to equalize the average workload on all the nodes of the system. The basic problem associated with the dynamic load-balancing algorithms is to decide about the amount of global state information to be used. Although only limited success has been achieved in this direction, the results of the efforts made have shown that attempting to gather a large amount of information to describe the current global state of the system more accurately is not always beneficial. Hence, the degree of global knowledge in an algorithm must be, in

some way, normalized to the complexity of the performance objective of the algorithm. This relationship and normalization are the subject of current research.

Finally, in the load-sharing approach, the process assignment decisions attempt to keep all the nodes busy if there are sufficient processes in the system for all the nodes. This is achieved by ensuring that no node is idle while processes wait to be processed at other nodes. Since load-sharing algorithms do not attempt to balance the load on all the nodes, they are simpler and easier to implement as compared to load-balancing algorithms. However, for the various global scheduling algorithms, there is no absolute answer to the question,"Is algorithm *A* better than *B*?" Therefore, getting a better understanding of the processes involved in global scheduling has to be the aim of study of this type of algorithms.

EXERCISES

7.1. For global scheduling algorithms, why are heuristic methods providing near-optimal results normally preferable to optimal solution methods?

7.2. What is "processor thrashing"? Give examples of two global scheduling algorithms that may lead to processor thrashing. Suggest necessary measures to be taken to handle this problem.

7.3. Suppose you have to design a scheduling algorithm based on the task assignment approach for scheduling the tasks of processes to be processed in a distributed system. What types of cost information would you like to have for the tasks of a process? Suggest methods that may be used to make a rough estimate of these costs.

7.4. A system consists of three processors p_1, p_2, and p_3, and a process having four tasks t_1, t_2, t_3, and t_4 is to be executed on this system. Suppose E_{ij} is the cost of executing task t_i on processor p_j and C_{ij} is the communication cost between tasks t_i and t_j when the two tasks are assigned to different processors. Let $E_{11}=31$, $E_{12}=4$, $E_{13}=14$, $E_{21}=1$, $E_{22}=5$, $E_{23}=6$, $E_{31}=2$, $E_{32}=4$, $E_{33}=24$, $E_{41}=3$, $E_{42}=28$, $E_{43}=10$, $C_{12}=35$, $C_{13}=3$, $C_{14}=8$, $C_{23}=6$, $C_{24}=4$, and $C_{34}=23$. Find an optimal assignment of the tasks to the processors and calculate the cost of optimal assignment. Now compare this cost with the assignment cost of the case in which t_1 and t_2 are assigned to p_1, t_3 is assigned to p_2, and t_4 is assigned to p_3.

7.5. A system consists of two processors p_1 and p_2, and a process having six tasks t_1, t_2, t_3, t_4, t_5, and t_6 is to be executed on this system. Suppose E_{ij} is the cost of executing task t_i on processor p_j. C_{ij} is the communication cost between tasks t_i and t_j when the two tasks are assigned to different processors, and I_{ij} is the interference cost between tasks t_i and t_j when the two tasks are assigned to the same processor. *Interference costs* reflect the degree of incompatibility between two tasks and are incurred when the two tasks are assigned to the same node [Lo 1988]. Let $E_{11}=20$, $E_{12}=50$, $E_{21}=25$, $E_{22}=10$, $E_{31}=5$, $E_{32}=20$, $E_{41}=10$, $E_{42}=20$, $E_{51}=10$, $E_{52}=20$, $E_{61}=50$, $E_{62}=10$, $C_{12}=15$, $C_{23}=50$, $C_{34}=15$, $C_{45}=50$, and $C_{56}=15$. The values of all other C_{ij} are zero. Furthermore, the interference cost between any pair of tasks is 10 (i.e., $I_{ij}=10$ if i is not equal to j). Find the assignment costs for the following cases:

(a) Task t_1 is assigned to p_1 and all other tasks are assigned to p_2, and interference cost is not taken into account.

(b) Task sets $\{t_1, t_2, t_3\}$ and $\{t_4, t_5, t_6\}$ are assigned to p_1 and p_2, respectively, and interference cost is not taken into account.

(c) Task t_1 is assigned to p_1 and all other tasks are assigned to p_2, and interference cost is taken into account.

(d) Task sets $\{t_1, t_2, t_3\}$ and $\{t_4, t_5, t_6\}$ are assigned to p_1 and p_2, respectively, and interference cost is taken into account.

What conclusion can be drawn from the assignment costs of these four cases?

7.6. A computer system has three processors p_1, p_2, and p_3. It is to be used to process the same type of processes, all of which have six tasks t_1, t_2, t_3, t_4, t_5, and t_6. Assume that the arrival rate of processes is a deterministic process. Also assume that every task's execution time is exponentially distributed with the mean execution time of tasks t_1, t_2, t_3, t_4, t_5, and t_6 being 1, 10, 1, 10, 1, and 10, respectively. There is a strong precedence relationship among the tasks of a process, and task t_j of a process can be executed only when the execution of task t_i ($i < j$) of the same process has been completed. Which one of the following assignments will yield the best response times and which one will yield the worst response times for the processes executed on this system:

(a) Task pairs (t_1, t_2), (t_3, t_4), and (t_5, t_6) are assigned to processors p_1, p_2, and p_3, respectively.

(b) Task pairs (t_1, t_6), (t_2, t_3), and (t_4, t_5) are assigned to processors p_1, p_2, and p_3, respectively.

(c) Task pairs (t_1, t_4), (t_2, t_5), and (t_3, t_6) are assigned to processors p_1, p_2, and p_3, respectively.

Give reasons for your answer.

7.7. A system has two processors p_1 and p_2 with p_1 having limited memory capacity and p_2 having unlimited memory. A process having multiple tasks is to be executed on this system. The execution costs for each task on each processor, the intertask communication costs, the memory requirement of each task, and the total memory capacity of processor p_1 is given. Suppose a_1 is the assignment that minimizes the total execution and communication costs without the memory size constraint, and a_2 is the assignment that minimizes the total execution and communication costs with the memory size constraint. Prove that the tasks assigned to p_1 for assignment a_2 is a subset of the tasks assigned to p_1 for assignment a_1.

7.8. Comment on the practical applicability of the task assignment approach as a scheduling scheme for distributed systems.

7.9. Comment on the practical applicability of the load-balancing approach as a scheduling scheme for the following types of distributed systems:

(a) A LAN-based distributed system

(b) A WAN-based distributed system

(c) A distributed system based on the processor-pool model

(d) A distributed system based on the workstation-server model

7.10. A distributed operating system designer is of the opinion that state information in a distributed system is typically gathered at a high cost. Therefore, for a distributed system based on the processor-pool model, he or she decides to use a load-balancing policy that uses the following simple process placement rule: *Execute all processes originating from terminal* i *on processor* j (j << i). *The value of* j *is defined for all values of* i, *and for several values of* i, *the value of* j *may be the same.*

In your opinion, is the designer's choice of the scheduling algorithm appropriate for this system? Give reasons for your answer. What drawbacks, if any, does this scheduling algorithm have? If you feel that this algorithm is not appropriate for this system, suggest a suitable global scheduling algorithm for this system.

7.11. Suppose you have to design a centralized load-balancing algorithm for global scheduling of processes in a distributed system. What issues must you handle? Suggest a suitable method for handling each of the issues mentioned by you.

7.12. Suppose you have to design a load-balancing algorithm for a distributed system. Which of the selfish, altruistic, and intermediate priority assignment policies will you use in your algorithm if the distributed system is based on the following:
(a) Workstation-server model
(b) Processor-pool model
Give reasons for your answer.

7.13. Suppose you have decided to use the high-low policy as the process transfer policy of a load-balancing algorithm for a distributed system. Suggest a suitable method that you will use in your implementation for choosing the high-mark and low-mark values. Do these threshold values have to be the same for all processors in the system? Give reasons for your answer.

7.14. Load balancing in the strictest sense is not achievable in distributed systems. Discuss.

7.15. What are the main differences between the load-balancing and load-sharing approaches for process scheduling in distributed systems? Which of the various policies to be used in the implementation of the two approaches are different and which of them are same?

7.16. Suppose you have to design a load-sharing algorithm for a distributed system. Will you prefer to use a sender-initiated or a receiver-initiated location policy in your algorithm? Give reasons for your answer.

7.17. Suggest some policies that may be used for load estimation in load-balancing algorithms. Discuss the relative advantages and disadvantages of the policies suggested by you. Which one of the policies suggested by you can also be used for load-sharing algorithms? If none, suggest a suitable load estimation policy for a load-sharing algorithm.

7.18. A system has two processors p_1 and p_2. Suppose at a particular instance of time, p_1 has one process with remaining service time of 200 seconds and p_2 has 100 processes each with a remaining service time of 1 second. Now suppose a new process enters the system. Calculate the response time of the new process if:
(a) The new process is a 1-second process and it is allocated to p_1 for execution.
(b) The new process is a 1-second process and it is allocated to p_2 for execution.
(c) The new process is a 200-second process and it is allocated to p_1 for execution.
(d) The new process is a 200-second process and it is allocated to p_2 for execution
Assume that there are no other new arrivals in the system. From the obtained results, what can you conclude about load estimation policies to be used in load-balancing algorithms?

7.19. A distributed system does not support preemptive process migration facility. You are to design a load-sharing algorithm for scheduling of processes in this system. Will you use a sender-initiated or a receiver-initiated location policy for your algorithm? Give reasons for your answer.

7.20. A distributed system uses the all-or-nothing strategy as the process transfer policy for its load-sharing algorithm. Explain why the processing capabilities of the processors of this system may not be properly utilized. Suggest a suitable method to overcome this problem.

7.21. What research issues do you think need further attention in the area of global scheduling of processes in distributed systems?

BIBLIOGRAPHY

[Ahrens and Hansen 1995] Ahrens, J. P., and Hansen, C. D., "Cost-Effective Data-Parallel Load Balancing," In: *Proceedings of the 24th Annual International Conference on Parallel Processing*, CRC Press, Boca Raton, FL (August 1995).