CHAPTER 6

DISTRIBUTED MUTUAL EXCLUSION

6.1 INTRODUCTION

In the problem of mutual exclusion, concurrent access to a shared resource by several uncoordinated user requests is serialized to secure the integrity of the shared resource. It requires that the actions performed by a user on a shared resource must be *atomic*. That is, if several users concurrently access a shared resource then the actions performed by a user, as far as the other users are concerned, must be instantaneous and indivisible such that the net effect on the shared resource is the same as if user actions were executed serially, as opposed to in an interleaved manner.

The problem of mutual exclusion frequently arises in distributed systems whenever concurrent access to shared resources by several sites is involved. For correctness, it is necessary that the shared resource be accessed by a single site (or process) at a time. A typical example is directory management, where an update to a directory must be done atomically because if updates and reads to a directory proceed concurrently, reads may obtain inconsistent information. If an entry contains several fields, a read operation may read some fields before the update and some after the update. Mutual exclusion is a fundamental issue in the design of distributed systems and an efficient and robust technique for mutual exclusion is essential to the viable design of distributed systems.

Mutual exclusion in single-computer systems vs. distributed systems

The problem of mutual exclusion in a single-computer system, where shared memory exists, was studied in Chap. 2. In single-computer systems, the status of a shared resource and the status of users is readily available in the shared memory, and solutions to the mutual exclusion problem can be easily implemented using shared variables (e.g., semaphores). However, in distributed systems, both the shared resources and the users may be distributed and shared memory does not exist. Consequently, approaches based on shared variables are not applicable to distributed systems and approaches based on message passing must be used.

The problem of mutual exclusion becomes much more complex in distributed systems (as compared to single-computer systems) because of the lack of both shared memory and a common physical clock and because of unpredictable message delays. Owing to these factors, it is virtually impossible for a site in a distributed system to have current and complete knowledge of the state of the system.

6.2 THE CLASSIFICATION OF MUTUAL EXCLUSION ALGORITHMS

Over the last decade, the problem of mutual exclusion has received considerable attention and several algorithms to achieve mutual exclusion in distributed systems have been proposed. They tend to differ in their communication topology (e.g., tree, ring, and any arbitrary graph) and in the amount of information maintained by each site about other sites. These algorithms can be grouped into two classes. The algorithms in the first class are nontoken-based, e.g., [4, 9, 10, 16, 19]. These algorithms require two or more successive rounds of message exchanges among the sites. These algorithms are assertion based because a site can enter its critical section (CS) when an assertion defined on its local variables becomes true. Mutual exclusion is enforced because the assertion becomes true only at one site at any given time.

The algorithms in the second class are token-based, e.g., [11, 14, 20, 21, 22]. In these algorithms, a unique token (also known as the PRIVILEGE message) is shared among the sites. A site is allowed to enter its CS if it possesses the token and it continues to hold the token until the execution of the CS is over. These algorithms essentially differ in the way a site carries out the search for the token.

In this chapter, we describe several distributed mutual exclusion algorithms and compare their features and performance. We discuss relationship among various mutual exclusion algorithms and examine trade offs among them.

6.3 PRELIMINARIES

We now describe the underlying system model and requirements that mutual exclusion algorithms should meet. We also introduce terminology that is used in describing the performance of mutual exclusion algorithms.

SYSTEM MODEL. At any instant, a site may have several requests for CS. A site queues up these requests and serves them one at a time. A site can be in one of the following three states: *requesting CS*, *executing CS*, or neither requesting nor executing CS (i.e., *idle*). In the requesting CS state, the site is blocked and cannot make further requests for CS. In the idle state, the site is executing outside its CS. In the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS. Such a state is refered to as an *idle token* state.

6.3.1 Requirements of Mutual Exclusion Algorithms

The primary objective of a mutual exclusion algorithm is to maintain mutual exclusion; that is, to guarantee that only one request accesses the CS at a time. In addition, the following characteristics are considered important in a mutual exclusion algorithm:

Freedom from Deadlocks. Two or more sites should not endlessly wait for messages that will never arrive.

Freedom from Starvation. A site should not be forced to wait indefinitely to execute CS while other sites are repeatedly executing CS. That is, every requesting site should get an opportunity to execute CS in a finite time.

Fairness. Fairness dictates that requests must be executed in the order they are made (or the order in which they arrive in the system). Since a physical global clock does not exist, time is determined by logical clocks. Note that fairness implies freedom from starvation, but not vice-versa.

Fault Tolerance. A mutual exclusion algorithm is fault-tolerant if in the wake of a failure, it can reorganize itself so that it continues to function without any (prolonged) disruptions.

6.3.2 How to Measure the Performance

The performance of mutual exclusion algorithms is generally measured by the following four metrics: First, the *number of messages* necessary per CS invocation. Second, the *synchronization delay*, which is the time required after a site leaves the CS and before the next site enters the CS (see Fig. 6.1). Note that normally one or more sequential message exchanges are required after a site exits the CS and before the next site enters the CS. Third, the *response time*, which is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Fig. 6.2). Thus, response time does not include the time a request waits at a site before its request messages have been sent out. Fourth, the *system throughput*, which is the rate at which the system executes requests for the CS. If sd is the synchronization delay and E is the average critical section execution time, then the throughput is given by the following equation:

$$\text{system throughput} = 1/(sd + E)$$

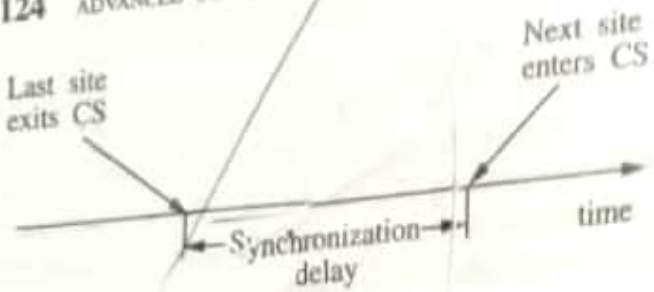

FIGURE 6.1
Synchronization delay.

LOW AND HIGH LOAD PERFORMANCE. Performance of a mutual exclusion algorithm depends upon the loading conditions of the system and is often studied under two special loading conditions, viz., *low load* and *high load*. Under low load conditions, there is seldom more than one request for mutual exclusion simultaneously in the system. Under high load conditions, there is always a pending request for mutual exclusion at a site. Thus, after having executed a request, a site immediately initiates activities to let the next site execute its CS. A site is seldom in an idle state under high load conditions. For many mutual exclusion algorithms, the performance metrics can be easily determined under low and high loads through simple reasoning.

BEST AND WORST CASE PERFORMANCE. Generally, mutual exclusion algorithms have best and worst cases for the performance metrics. In the best case, prevailing conditions are such that a performance metric attains the best possible value. For example, in most algorithms the best value of the response time is a round-trip message delay plus CS execution time, $2T + E$ (where T is the average message delay and E is the average critical section execution time).

Often for mutual exclusion algorithms, the best and worst cases coincide with low and high loads, respectively. For example, the best and worst values of the response time are achieved when the load is, respectively, low and high. The best and the worse message traffic is generated in Maekawa's algorithm [10] at low and high load conditions, respectively. When the value of a performance metric fluctuates statistically, we generally talk about the average value of that metric.

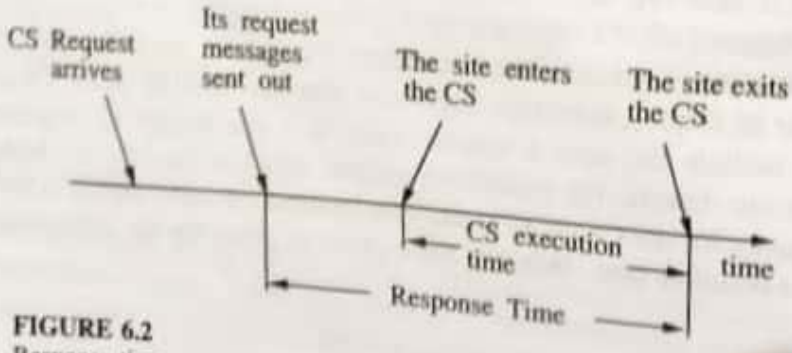

FIGURE 6.2
Response time.

6.4 A SIMPLE SOLUTION TO DISTRIBUTED MUTUAL EXCLUSION

In a simple solution to distributed mutual exclusion, a site, called the *control site*, is assigned the task of granting permission for the CS execution. To request the CS, a site sends a REQUEST message to the control site. The control site queues up the requests for the CS and grants them permission, one by one. This method to achieve mutual exclusion in distributed systems requires only three messages per CS execution.

This naive, centralized solution has several drawbacks. First, there is a single point of failure, the control site. Second, the control site is likely to be swamped with extra work. Also, the communication links near the control site are likely to be congested and become a bottleneck. Third, the synchronization delay of this algorithm is $2T$ because a site should first release permission to the control site and then the control site should grant permission to the next site to execute the CS. This has serious implications for the system throughput, which is equal to $1/(2T + E)$ in this algorithm. Note that if the synchronization delay is reduced to T, the system throughput is almost doubled to $1/(T + E)$. We later discuss several mutual exclusion algorithms that reduce the synchronization delay to T at the cost of higher message traffic.

6.5 NON-TOKEN-BASED ALGORITHMS

In non-token-based mutual exclusion algorithms, a site communicates with a set of other sites to arbitrate who should execute the CS next. For a site S_i, request set R_i contains ids of all those sites from which site S_i must acquire permission before entering the CS. Next, we discuss some non-token-based mutual exclusion algorithms which are good representatives of this class.

Non-token-based mutual exclusion algorithms use timestamps to order requests for the CS and to resolve conflicts between simultaneous requests for the CS. In all these algorithms, logical clocks are maintained and updated according to Lamport's scheme [9]. Each request for the CS gets a timestamp, and smaller timestamp requests have priority over larger timestamp requests.

6.6 LAMPORT'S ALGORITHM

Lamport was the first to give a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme [9]. In Lamport's algorithm, $\forall i : 1 \leq i \leq N :: R_i = \{S_1, S_2, ..., S_N\}$. Every site S_i keeps a queue, $request\_queue_i$, which contains mutual exclusion requests ordered by their timestamps. This algorithm requires messages to be delivered in the FIFO order between every pair of sites.

The Algorithm

Requesting the critical section.

1. When a site S_i wants to enter the CS, it sends a REQUEST(ts_i, i) message to all the sites in its request set R_i and places the request on $request\_queue_i$. ((ts_i, i) is the timestamp of the request.)

2. When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it returns a timestamped REPLY message to S_i and places site S_i's request on $request\_queue_j$.

Executing the critical section. Site S_i enters the CS when the two following conditions hold:

[**L1:**] S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

[**L2:**] S_i's request is at the top of $request\_queue_i$.

Releasing the critical section.

3. Site S_i, upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.
4. When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. The algorithm executes CS requests in the increasing order of timestamps.

Correctness

Theorem 6.1. Lamport's algorithm achieves mutual exclusion.

Proof: The proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently. For this to happen, conditions L1 and L2 must hold at both the sites concurrently. This implies that at some instant in time, say t, both S_i and S_j have their own requests at the top of their $request\_queues$ and condition L1 holds at them. Without a loss of generality, assume that S_i's request has a smaller timestamp than the request of S_j. Due to condition L1 and the FIFO property of the communication channels, it is clear that at instant t, the request of S_i must be present in $request\_queue_j$, when S_j was executing its CS. This implies that S_j's own request is at the top of its own $request\_queue$ when a smaller timestamp request,

2. When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it returns a timestamped REPLY message to S_i and places site S_i's request on $request\_queue_j$.

Executing the critical section. Site S_i enters the CS when the two following conditions hold:

[L1:] S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

[L2:] S_i's request is at the top of $request\_queue_i$.

Releasing the critical section.

3. Site S_i, upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.
4. When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. The algorithm executes CS requests in the increasing order of timestamps.

Correctness

Theorem 6.1. Lamport's algorithm achieves mutual exclusion.

Proof: The proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently. For this to happen, conditions L1 and L2 must hold at both the sites concurrently. This implies that at some instant in time, say t, both S_i and S_j have their own requests at the top of their *request_queues* and condition L1 holds at them. Without a loss of generality, assume that S_i's request has a smaller timestamp than the request of S_j. Due to condition L1 and the FIFO property of the communication channels, it is clear that at instant t, the request of S_i must be present in $request\_queue_j$, when S_j was executing its CS. This implies that S_j's own request is at the top of its own $request\_queue$ when a smaller timestamp request, S_i's request, is present in the $request\_queue_j$—a contradiction! Hence, Lamport's algorithm achieves mutual exclusion. □

Example 6.1. In Fig. 6.3 through Fig. 6.6, we illustrate the operation of Lamport's algorithm. In Fig. 6.3, sites S_1 and S_2 are making requests for the CS and send out REQUEST messages to other sites. The timestamps of the requests are (2, 1) and (1, 2), respectively. In Fig. 6.4, S_2 has received REPLY messages from all the other sites and its request is at the top of its $request\_queue$. Consequently, it enters the CS. In Fig. 6.5, S_2 exits and sends RELEASE messages to all other sites. In Fig. 6.6, site S_1 has received REPLY messages from all other sites and its request is at the top of its $request\_queue$. Consequently, it enters the CS next.

PERFORMANCE. Lamport's algorithm requires $3(N-1)$ messages per CS invocation: $(N-1)$ REQUEST, $(N-1)$ REPLY, and $(N-1)$ RELEASE messages. Synchronization delay in the algorithm is T.

AN OPTIMIZATION. Lamport's algorithm can be optimized to require between $3(N-1)$ and $2(N-1)$ messages per CS execution by suppressing REPLY messages

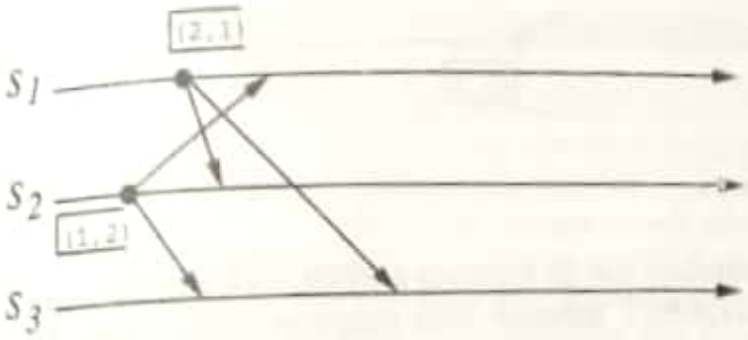

FIGURE 6.3
Sites S_1 and S_2 are making requests for the CS.

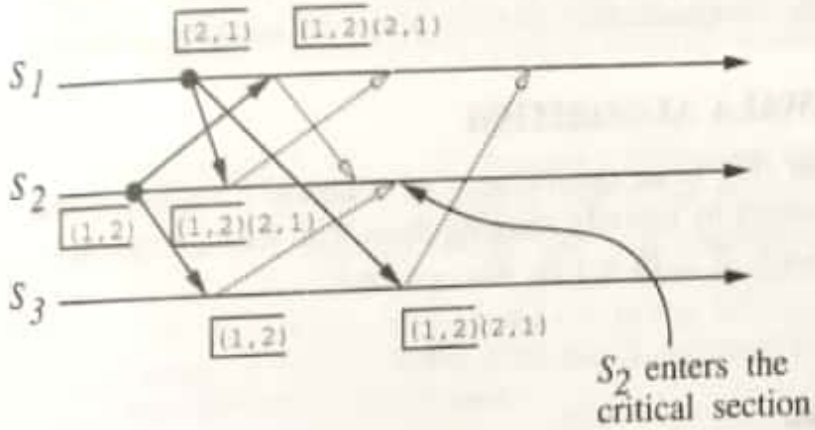

FIGURE 6.4
Site S_2 enters the CS.

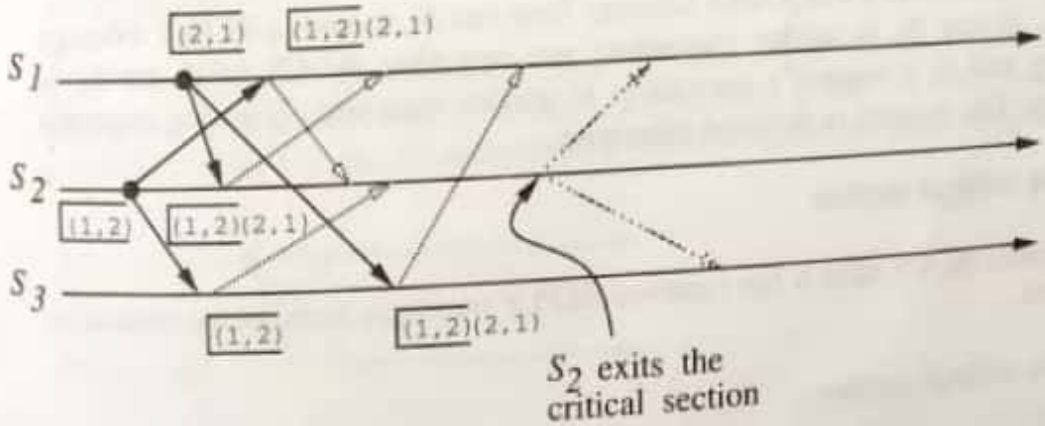

FIGURE 6.5
Site S_2 exits the CS and sends RELEASE messages.

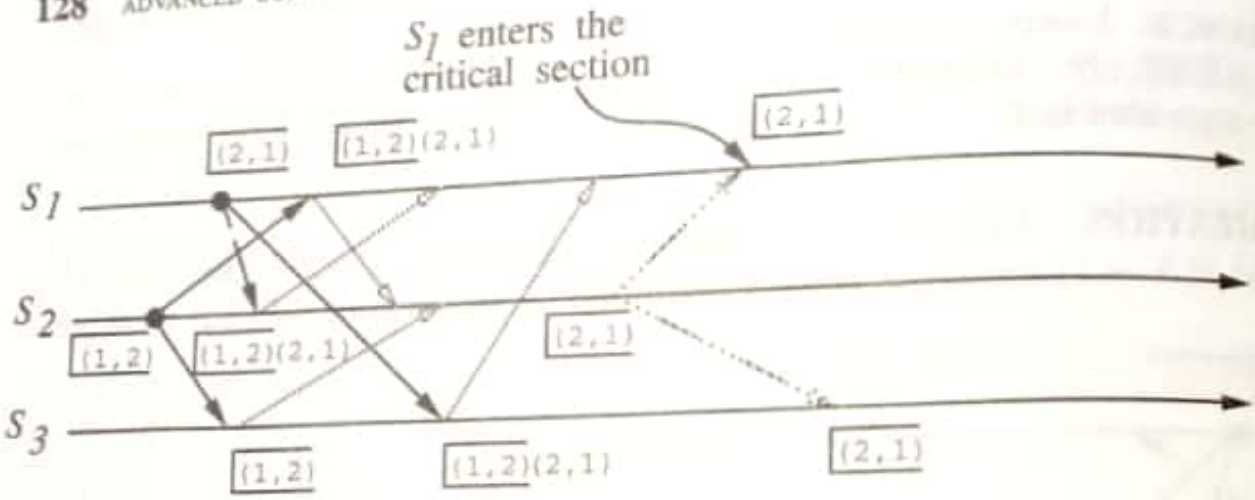

FIGURE 6.6
Site S_1 enters the CS.

in certain situations. For example, suppose site S_j receives a REQUEST message from site S_i after it has sent its own REQUEST message with timestamp higher than the timestamp of site S_i's request. In this case, site S_j need not send a REPLY message to site S_i. This is because when site S_i receives site S_j's request with a timestamp higher than its own, it can conclude that site S_j does not have any smaller timestamp request that is still pending (because the communication medium preserves message ordering).

6.7 THE RICART-AGRAWALA ALGORITHM

The Ricart-Agrawala algorithm [16] is an optimization of Lamport's algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages. In this algorithm also, $\forall i : 1 \leq i \leq N :: R_i = \{S_1, S_2, \ldots, S_N\}$.

The Algorithm

Requesting the critical section.

1. When a site S_i wants to enter the CS, it sends a timestamped REQUEST message to all the sites in its request set.
2. When site S_j receives a REQUEST message from site S_i, it sends a REPLY message to site S_i if site S_j is neither requesting nor executing the CS or if site S_j is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. The request is deferred otherwise.

Executing the