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:

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

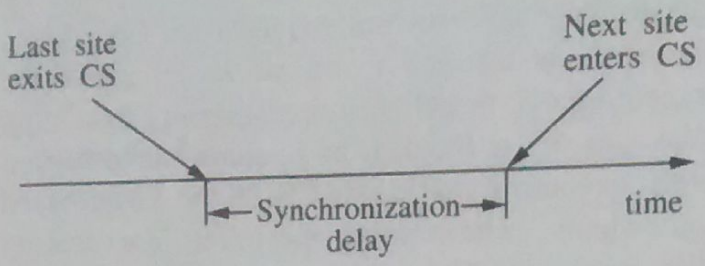

FIGURE 6.1
Synchronization delay.

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.

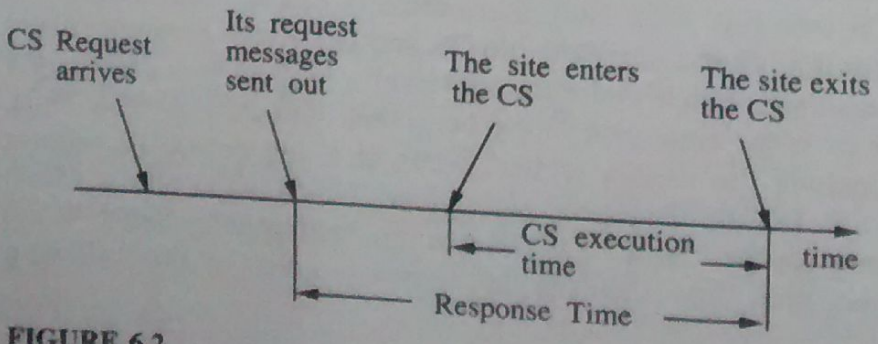

FIGURE 6.2
Response time.

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, 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

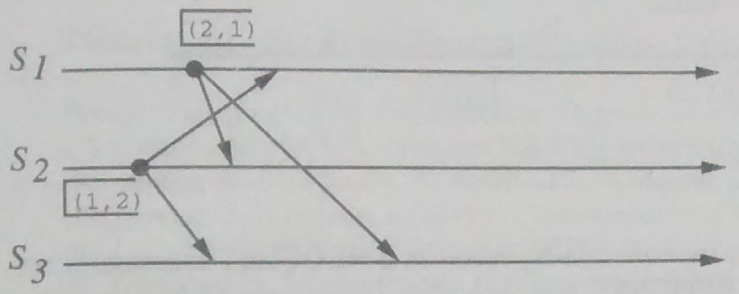

FIGURE 6.3
Sites S_1 and S_2 are making requests for the CS.

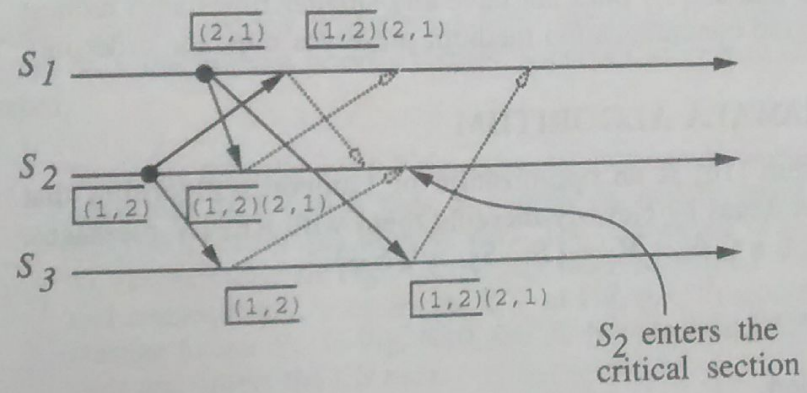

FIGURE 6.4
Site S_2 enters the CS.

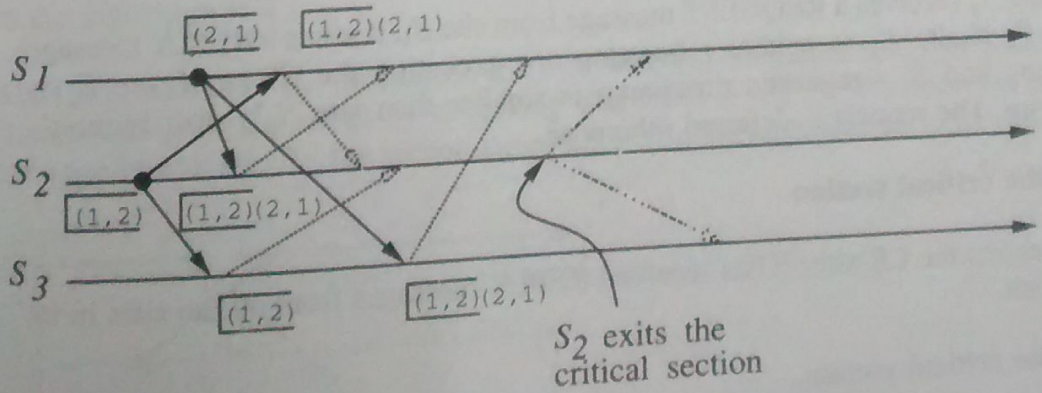

FIGURE 6.5
Site S_2 exits the CS and sends RELEASE messages.

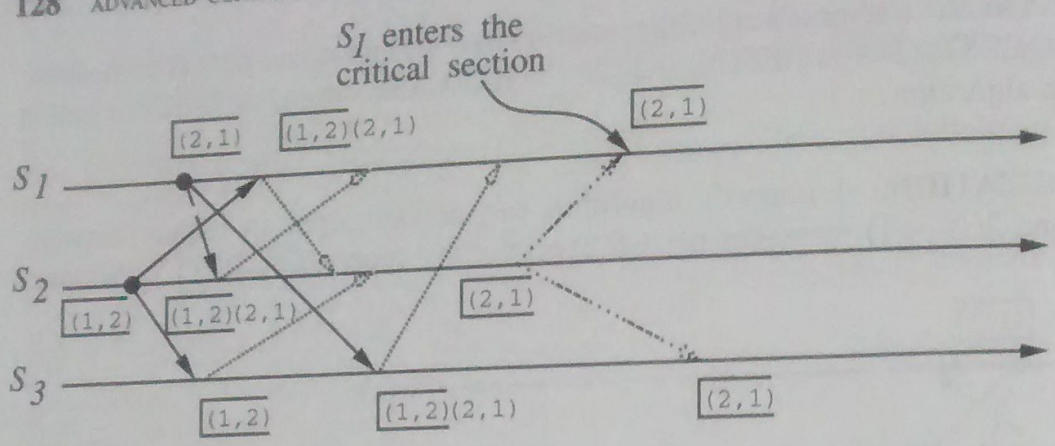

FIGURE 6.6
Site S_1 enters the CS.

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 critical section

3. Site S_i enters the CS after it has received REPLY messages from all the sites in its request set.

Releasing the critical section

4. When site S_i exits the CS, it sends REPLY messages to all the deferred requests.

A site's REPLY messages are blocked only by sites that are requesting the CS with higher priority (i.e., a smaller timestamp). Thus, when a site sends out REPLY messages to all the deferred requests, the site with the next highest priority request receives the last needed REPLY message and enters the CS. The execution of CS requests in this algorithm is always in the order of their timestamps.

CORRECTNESS

Theorem 6.2. The Ricart-Agrawala algorithm achieves mutual exclusion.

Proof: The proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently and S_i's request has a higher priority (i.e., a smaller timestamp) than the request of S_j. Clearly, S_i received S_j's request after it had made its own request. (Otherwise, S_i's request would have lower priority.) Thus, S_j can concurrently execute the CS with S_i only if S_i returns a REPLY to S_j (in response to S_j's request) before S_i exits the CS. However, this is impossible because S_j's request has lower priority. Therefore, the Ricart-Agrawala algorithm achieves mutual exclusion. □

In the Ricart-Agrawala algorithm, for every requesting pair of sites, the site with higher priority request will always defer the request of the lower priority site. At any time, only the highest priority request succeeds in getting all the needed REPLY messages.

Example 6.2. Figures 6.7 through 6.10 illustrate the operation of the Ricart-Agrawala algorithm. In Fig. 6.7, sites S_1 and S_2 are making requests for the CS, sending out REQUEST messages to other sites. The timestamps of the requests are (2, 1) and (1, 2), respectively. In Fig. 6.8, S_2 has received REPLY messages from all other sites and consequently, it enters the CS. In Fig. 6.9, S_2 exits the CS and sends a REPLY mesage to site S_1. In Fig. 6.10, site S_1 has received REPLY messages from all other sites and enters the CS next.

PERFORMANCE. The Ricart-Agrawala algorithm requires $2(N-1)$ messages per CS execution: $(N-1)$ REQUEST and $(N-1)$ REPLY messages. Synchronization delay in the algorithm is T.

AN OPTIMIZATION. Roucairol and Carvalho [4] proposed an improvement to the Ricart-Agrawala algorithm by observing that once a site S_i has received a REPLY message from a site S_j, the authorization implicit in this message remains valid until S_i

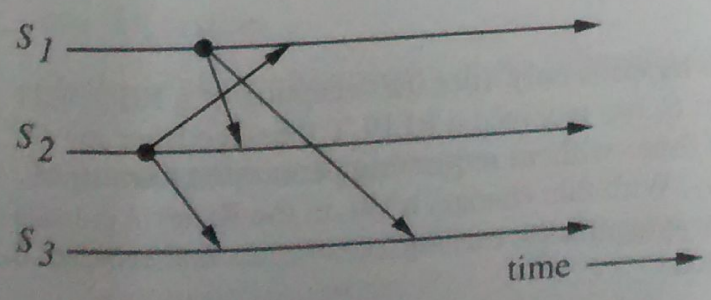

FIGURE 6.7
Sites S_1 and S_2 are making requests for the CS.

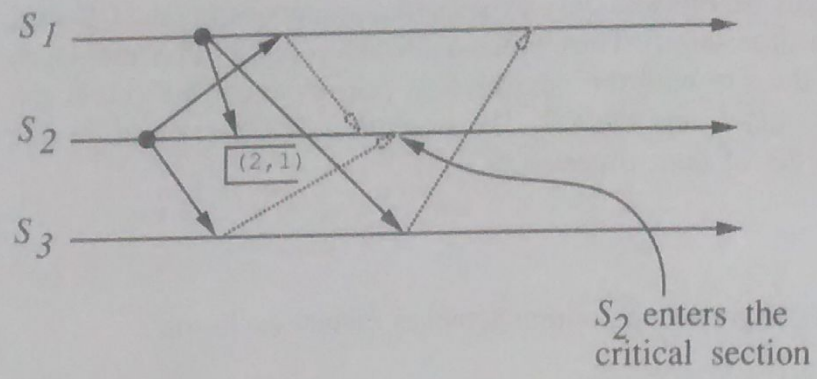

FIGURE 6.8
Site S_2 enter the CS.

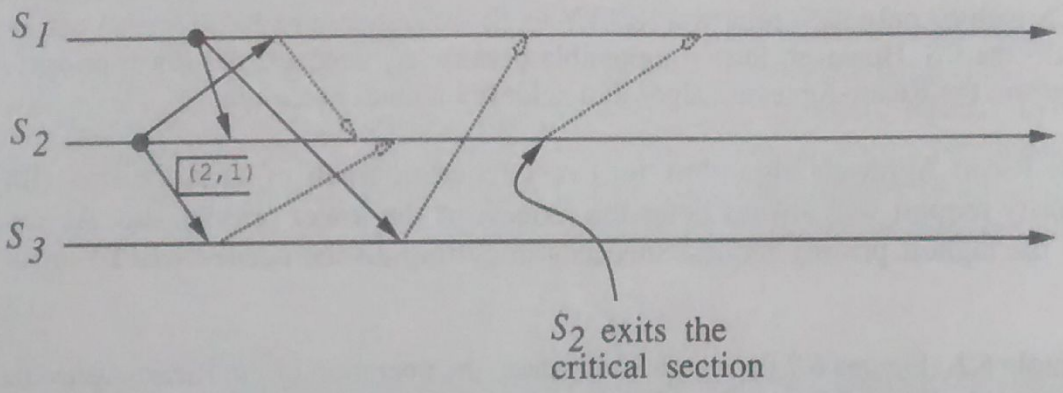

FIGURE 6.9
Site S_2 exits the CS and sends a REPLY message to S_1.

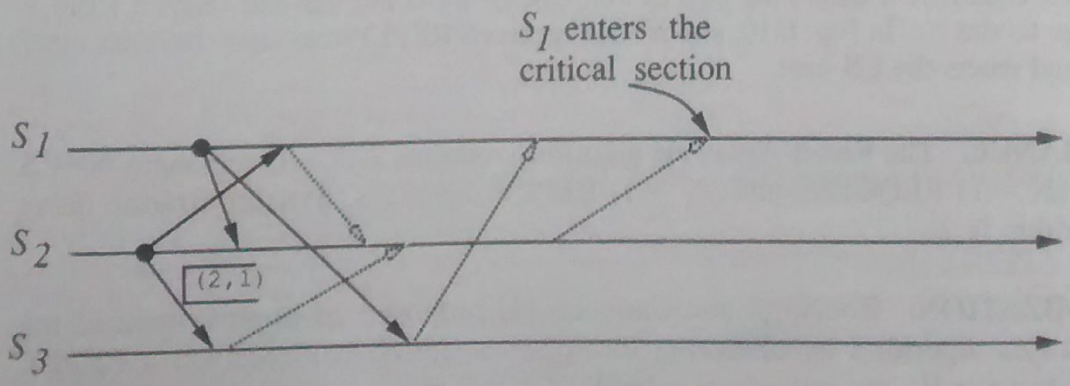

FIGURE 6.10
Site S_1 enters the CS.

sends a REPLY message to S_j (which happens only after the reception of a REQUEST message from S_j). Therefore, after site S_i has received a REPLY message from site S_j, site S_i can enter its CS any number of times without requesting permission from site S_j until S_i sends a REPLY message to S_j. With this change, a site in the Ricart-Agrawala algorithm requests permission from a dynamically varying set of sites and requires 0 to $2(N-1)$ messages per CS execution.

6.8 MAEKAWA'S ALGORITHM

Maekawa's algorithm [10] is a departure from the general trend in the following two ways: First, a site does not request permission from every other site, but only from a subset of the sites. This is a radically different approach as compared to the Lamport and the Ricart-Agrawala algorithms, where all sites participate in the conflict resolution of all other sites. In Maekawa's algorithm, the request set of each site is chosen such that $\forall i \forall j : 1 \leq i, j \leq N :: R_i \cap R_j \neq \phi$. Consequently, every pair of sites has a site that mediates conflicts between that pair. Second, in Maekawa's algorithm a site can send out only one REPLY message at a time. A site can only send a REPLY message only after it has received a RELEASE message for the previous REPLY message. Therefore, a site S_i locks all the sites in R_i in exclusive mode before executing its CS.

THE CONSTRUCTION OF REQUEST SETS. The request sets for sites in Maekawa's algorithm are constructed to satisfy the following conditions:

M1: $(\forall i \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \phi)$

M2: $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$

M3: $(\forall i : 1 \leq i \leq N :: |R_i| = K)$

M4: Any site S_j is contained in K number of R_is, $1 \leq i, j \leq N$. Maekawa established the following relation between N and K: $N = K(K-1)+1$. This relation gives $|R_i| = \sqrt{N}$.

Since there is at least one common site between the request sets of any two sites (condition M1), every pair of sites has a common site that mediates conflicts between the pair. A site can have only one outstanding REPLY message at any time; that is, it grants permission to an incoming request if it has not granted permission to some other site. Therefore, mutual exclusion is guaranteed. This algorithm requires the delivery of messages to be in the order they are sent between every pair of sites.

Conditions M1 and M2 are necessary for correctness, whereas conditions M3 and M4 provide other desirable features to the algorithm. Condition M3 states that the size of the request sets of all the sites must be equal, implying that all sites should have to do an equal amount of work to invoke mutual exclusion. Condition M4 enforces that exactly the same number of sites should request permission from any site, implying that all sites have equal responsibility in granting permission to other sites.

The Algorithm

Maekawa's algorithm works in the following manner:

Requesting the critical section.

1. A site S_i requests access to the CS by sending REQUEST(i) messages to all the sites in its request set R_i.

2. When a site S_j receives the REQUEST(i) message, it sends a REPLY(j) message to S_i provided it hasn't sent a REPLY message to a site from the time it received the last RELEASE message. Otherwise, it queues up the REQUEST for later consideration.

Executing the critical section.

3. Site S_i accesses the CS only after receiving REPLY messages from all the sites in R_i.

Releasing the critical section.

4. After the execution of the CS is over, site S_i sends RELEASE(i) message to all the sites in R_i.
5. When a site S_j receives a RELEASE(i) message from site S_i, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that the site has not sent out any REPLY message.

CORRECTNESS

Theorem 6.3. Maekawa's algorithm achieves mutual exclusion.

Proof: The proof is by contradiction. Suppose two sites S_i and S_j are concurrently executing the CS. If $R_i \cap R_j = \{S_k\}$, then site S_k must have sent REPLY messages to both S_i and S_j concurrently, which is a contradiction. □

PERFORMANCE. Note that the size of a request set is $\sqrt{N}$. Therefore, the execution of a CS requires $\sqrt{N}$ REQUEST, $\sqrt{N}$ REPLY, and $\sqrt{N}$ RELEASE messages, resulting in $3\sqrt{N}$ messages per CS execution. Synchronization delay in this algorithm is $2T$. As discussed next, Maekawa's algorithm is deadlock-prone. Measures to handle deadlocks require additional messages.

THE PROBLEM OF DEADLOCKS. Maekawa's algorithm is prone to deadlocks because a site is exclusively locked by other sites and requests are not prioritized by their timestamps [10, 17]. Without the loss of generality, assume three sites S_i, S_j, and S_k simultaneously invoke mutual exclusion. (Suppose $R_i \cap R_j = \{S_{ij}\}$, $R_j \cap R_k = \{S_{jk}\}$, and $R_k \cap R_i = \{S_{ki}\}$.) Since sites do not send REQUEST messages to the sites in their request sets in any particular order, it is possible that due to arbitrary message delays, S_{ij} has been locked by S_i (forcing S_j to wait at S_{ij}), S_{jk} has been locked by S_j (forcing S_k to wait at S_{jk}), and S_{ki} has been locked by S_k (forcing S_i to wait at S_{ki}) resulting in a deadlock involving the sites S_i, S_j, and S_k.

HANDLING DEADLOCKS. Maekawa's algorithm handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some

other request waiting for the same lock (unless the former has succeeded in locking all the needed sites) [10, 17]. A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request finds that a lower priority request has already locked the site. Deadlock handling requires the following three types of messages:

FAILED. A FAILED message from site S_i to site S_j indicates that S_i cannot grant S_j's request because it has currently granted permission to a site with a higher priority request.

INQUIRE. An INQUIRE message from S_i to S_j indicates that S_i would like to find out from S_j if it has succeeded in locking all the sites in its request set.

YIELD. A YIELD message from site S_i to S_j indicates that S_i is returning the permission to S_j (to yield to a higher priority request at S_j).

Details of the deadlock handling steps are as follows:

- When a REQUEST(ts, i) from site S_i blocks at site S_j because S_j has currently granted permission to site S_k, then S_j sends a FAILED(j) message to S_i if S_i's request has lower priority. Otherwise, S_j sends an INQUIRE(j) message to site S_k.
- In response to an INQUIRE(j) message from site S_j, site S_k sends a YIELD(k) message to S_j, provided, S_k has received a FAILED message from a site in its request set or if it sent a YIELD to any of these sites, but has not received a new REPLY from it.
- In response to a YIELD(k) message from site S_k, site S_j assumes it has been released by S_k, places the request of S_k at the appropriate location in the request queue, and sends a REPLY(j) to the top request's site in the queue.

Thus, Maekawa-type algorithms require extra messages to handle deadlocks and may exchange these messages even though there is no deadlock. The maximum number of messages required per CS execution in this case is $5\sqrt{N}$.

6.9 A GENERALIZED NON-TOKEN-BASED ALGORITHM

Sanders gave a generalized non-token-based mutual exclusion algorithm for distributed systems [17] and all the existing non-token-based mutual exclusion algorithms are special cases of this algorithm. The concept of *information structure* forms the basis for unifying different non-token-based mutual exclusion algorithms.

6.9.1 Information Structures

The information structure of a mutual exclusion algorithm defines the data structure needed at a site to record the status of other sites. The information kept in the information structure is used by a site in making decisions (i.e., from which sites to request permission) when invoking mutual exclusion.

The information structures at a site S_i consists of the following three sets: a request set R_i, an inform set I_i, and a status set St_i. These sets consist of the ids of the sites of the system. A site must acquire permission from all the sites in its request set before entering CS. Every site must inform all the sites in its inform set of its status change due to the wait to enter the CS and due to the exit from the CS. The status set St_i contains the ids of sites for which S_i maintains status information. Note that the inform set and the status set are dependent on each other. The contents of one is decided by the contents of the other. If $S_i \in I_j \Rightarrow S_j \in St_i$.

A site also maintains a variable CSSTAT, which indicates the site's knowledge of the status of the CS. Every site maintains a queue which contains REQUEST messages, in the order of their timestamps, for which no GRANT message has been sent.

CORRECTNESS CONDITION. To guarantee mutual exclusion, the information structure of sites in the generalized algorithm must satisfy the conditions given by the following theorem [17]:

Theorem 6.4. If $\forall i : 1 \leq i \leq N :: S_i \in I_i$, then the following two conditions are necessary and sufficient to guarantee mutual exclusion:

G1: $\forall i : 1 \leq i \leq N :: I_i \subset R_i$

G2: $\forall i \forall j : 1 \leq i, j \leq N :: (I_i \cap I_j \neq \Phi) \vee (S_i \in R_j \wedge S_j \in R_i)$

The correctness condition G2 states that for every pair of sites, either they request permission from each other or they request permission from a common site (which maintains the status information of both).

6.9.2 The Generalized Algorithm

In the generalized algorithm, each request for a CS is assigned a timestamp that is maintained according to Lamport's scheme [9]. Timestamps are used to prioritize requests in case of conflicts.

Requesting the critical section.

1. To execute CS, a site sends timestamped REQUEST messages to all the sites in its request set.
2. On the receipt of a REQUEST message, a site S_i takes the following actions:
   - It places the request in its queue (which is ordered by timestamps).
   - If CSSTAT indicates that the CS is free, then it sends a GRANT message to the site at the top of the queue and removes its entry from the queue. If the recipient of the GRANT message is in St_i, then CSSTAT is set to indicate that the site is in CS.

Executing the critical section.

3. A site executes the CS only after it has received a GRANT message from all the sites in its request set.

Releasing the critical section.

4. On exiting the CS, the site sends a RELEASE message to every site in its inform set. On receiving a RELEASE message, a site S_i takes the following actions:
   - CSSTAT is set to free.
   - If its queue is nonempty, then it sends a GRANT message to the site at the top of the queue and removes its entry from the queue. If the recipient of the GRANT message is in St_i, then CSSTAT is set to indicate that the site is in the CS.
   - The previous action is repeated until CSSTAT indicates that a site is in the CS or its queue becomes empty.

The proof of correctness is quite involved and is therefore omitted. Interested readers are encouraged to refer to the original paper [17].

DISCUSSION OF THE GENERALIZED ALGORITHM. The generalized algorithm combines the strategies of the Ricart-Agrawala algorithm [16] and Maekawa's [10] algorithm. In both these algorithms, a site invoking mutual exclusion acquires permission from a set of sites. However, the semantics of permission are very different in both the algorithms. In the Ricart-Agrawala algorithm, a site can grant permission to many sites simultaneously. A site grants permission to a requesting site immediately if it is not requesting the CS or its own request has lower priority. Otherwise, it defers granting permission until its execution of the CS is over. The Semantics of granting permission in this algorithm is essentially, *"As far as I am concerned, it is OK for you to enter the CS."* A site handles a REQUEST message to take care of its mutual exclusion with respect to all other sites.

In Maekawa's algorithm, a site can grant permission only to one site at a time. A site grants permission to a site only if it has not currently granted permission to another site. Otherwise, it delays granting permission until the currently granted permission has been released. Thus, acquiring permission is like locking the site in the exclusive mode. The semantics of granting permission in this algorithm is *"As far as all the sites in my status set are concerned, it is OK for you to enter the CS."* Thus, a site S_i handles a REQUEST message so that the requesting site can have mutual exclusion with respect to sites in S_i's status set. By granting permission to a site, the site guarantees that no other sites in its status set can execute the CS concurrently.

In the generalized algorithm, a site S_i acquires permission of the Maekawa type from all the sites in its inform set I_i, and acquires permission of the Ricart-Agrawala type from all the sites in set $R_i - I_i$. In response to a REQUEST message, a site S_j sends Maekawa type permission to sites in its status set and sends Ricart-Agrawala type permission to all the other sites. After a site has granted Maekawa type permission, it cannot grant permission to any other site unless it has been released. However, in the generalized algorithm a site can concurrently grant many Ricart-Agrawala type permissions preceding a Maekawa type permission.

If the first predicate of correctness condition G2 is false for all S_i and S_j, then the resulting algorithm of the Ricart-Agrawala type. If the second predicate of condition G2 is false for all S_i and S_j, then the resulting algorithm is of the Maekawa type. If the first predicate is true for some sites and the second predicate is true for other sites, then the resulting algorithm is a generalized mutual exclusion algorithm.

Example 6.3. Figure 6.11 illustrates three mutual exclusion algorithms in terms of their information structures [17]. A solid arrow from S_i to S_j indicates that $S_j \in I_i$ and $S_j \in R_i$. A dashed arrow from S_i to S_j indicates that $S_j \in R_i$ and $S_j \notin I_i$. Figure 6.11(a) shows the information structure of a mutual exclusion algorithm in which a single site, S_1, controls entry into the CS [3]. Figure 6.11(b) shows the information structure of a mutual exclusion algorithm where every site requests permission of every other site to enter the CS and, a site maintains information about its own status. An example of such an algorithm is the Ricart-Agrawala algorithm [16]. Figure 6.11(c) shows the information structure of Maekawa's mutual exclusion algorithm with 4 sites. Note that in this case, the request set and the inform set of every site is identical and $\forall i\ \forall j : i \neq j, 1 \leq i, j \leq 4 :: R_i \cap R_j \neq \phi$.

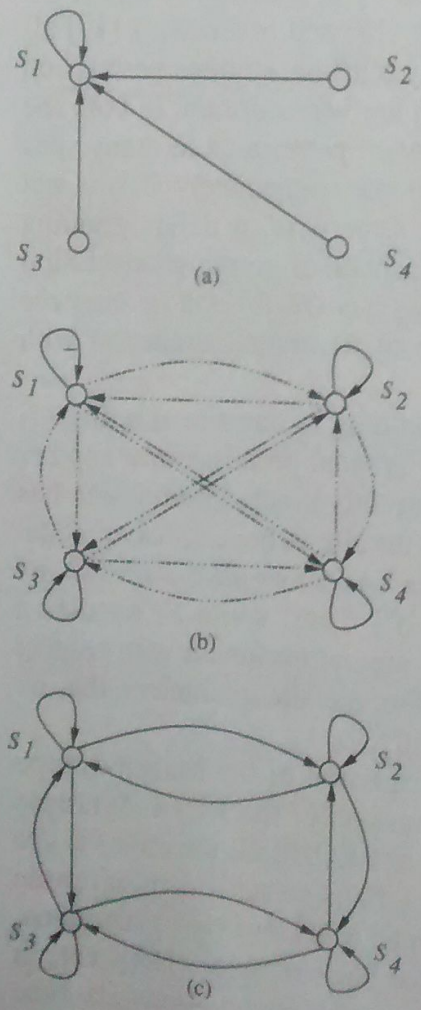

FIGURE 6.11
Examples of information structures.

6.9.3 Static vs. Dynamic Information Structures

Non-token-based mutual exclusion algorithms can be classified as either *static* or *dynamic* information structure algorithms. In static information structure algorithms, the contents of request sets, inform sets, and status sets remain fixed and do not change as sites execute CS. Examples of such algorithms are Lamport's [9], Maekawa's [10], and Ricart-Agrawala's [16] algorithms. In dynamic information structure algorithms, the contents of these sets change as the sites execute CS. Examples of such algorithms are found in [4] and [19]. The design of dynamic information structure mutual exclusion algorithms is much more complex because it requires rules for updating the information structure such that the conditions for mutual exclusion are always satisfied. This is the reason that most mutual exclusion algorithms for distributed systems have static information structures.

6.10 TOKEN-BASED ALGORITHMS

In token-based algorithms, a unique token is shared among all sites. A site is allowed to enter its CS if it possesses the token. Depending upon the way a site carries out its search for the token, there are numerous token-based algorithms. Next, we discuss some representative token-based mutual exclusion algorithms.

Before we start with the discussion of token-based algorithms, two comments are in order: First, token-based algorithms use sequence numbers instead of timestamps. Every request for the token contains a sequence number and the sequence numbers of sites advance independently. A site increments its sequence number counter every time it makes a request for the token. A primary function of the sequence numbers is to distinguish between old and current requests. Second, a correctness proof of token-based algorithms to ensure that mutual exclusion is enforced is trivial because an algorithm guarantees mutual exclusion so long as a site holds the token during the execution of the CS. Rather, the issues of freedom from starvation and freedom from deadlock are prominent.

6.11 SUZUKI-KASAMI'S BROADCAST ALGORITHM

In the Suzuki-Kasami's algorithm [21], if a site attempting to enter the CS does not have the token, it broadcasts a REQUEST message for the token to all the other sites. A site that possesses the token sends it to the requesting site upon receiving its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has exited the CS. A site holding the token can enter its CS repeatedly until it sends the token to some other site.

The main design issues in this algorithm are: (1) distinguishing outdated REQUEST messages from current REQUEST messages and (2) determining which site has an outstanding request for the CS.

Outdated REQUEST messages are distinguished from current REQUEST messages in the following manner: A REQUEST message of site S_j has the form REQUEST(j, n) where n ($n = 1, 2, \ldots$) is a sequence number that indicates that site S_j is requesting its n^{th} CS execution. A site S_i keeps an array of integers RN_i[1..N] where

RN_i[j] is the largest sequence number received so far in a REQUEST message from site S_j. A REQUEST(j, n) message received by site S_i is outdated if RN_i[j] > n. When site S_i receives a REQUEST(j, n) message, it sets RN_i[j]:= max(RN_i[j], n).

Sites with outstanding requests for the CS are determined in the following manner: The token consists of a queue of requesting sites, Q, and an array of integers LN[1..N] where LN[j] is the sequence number of the request that site S_j executed most recently. After executing its CS, a site S_i updates LN[i]:=RN_i[i] to indicate that its request corresponding to sequence number RN_i[i] has been executed. The token array LN[1..N] permits a site to determine if some other site has an outstanding request for the CS. Note that at site S_i if RN_i[j] = LN[j]+1, then site S_j is currently requesting the token. After having executed the CS, a site checks this condition for all the j's to determine all the sites that are requesting the token and places their ids in queue Q if not already present in this queue Q. Then the site sends the token to the site at the head of the queue Q.

The Algorithm

Requesting the critical section

1. If the requesting site S_i does not have the token, then it increments its sequence number, RN_i[i], and sends a REQUEST(i, sn) message to all other sites. (sn is the updated value of RN_i[i].)
2. When a site S_j receives this message, it sets RN_j[i] to max(RN_j[i], sn). If S_j has the idle token, then it sends the token to S_i if RN_j[i]=LN[i]+1.

Executing the critical section.

3. Site S_i executes the CS when it has received the token.

Releasing the critical section. Having finished the execution of the CS, site S_i takes the following actions:

4. It sets LN[i] element of the token array equal to RN_i[i].
5. For every site S_j whose ID is not in the token queue, it appends its ID to the token queue if RN_i[j]=LN[j]+1.
6. If token queue is nonempty after the above update, then it deletes the top site ID from the queue and sends the token to the site indicated by the ID.

Thus, after having executed its CS, a site gives priority to other sites with outstanding requests for the CS (over its pending requests for the CS). The Suzuki-Kasami algorithm is not symmetric because a site retains the token even if it does not have a request for the CS, which is contrary to the spirit of Ricart and Agrawala's definition of a symmetric algorithm: *"no site possesses the right to access its CS when it has not been requested."*

CORRECTNESS

Theorem 6.5. A requesting site enters the CS in finite time.

Proof: Token request messages of a site S_i reach other sites in finite time. Since one of these sites will have the token in finite time, site S_i's request will be placed in the token queue in finite time. Since there can be at most $N-1$ requests in front of this request in the token queue, site S_i will execute the CS in finite time. □

PERFORMANCE. The beauty of the Suzuki-Kasami algorithm lies in its simplicity and efficiency. The algorithm requires 0 or N messages per CS invocation. Synchronization delay in this algorithm is 0 or T. No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request.

6.12 SINGHAL'S HEURISTIC ALGORITHM

In Singhal's token-based heuristic algorithm [20], each site maintains information about the state of other sites in the system and uses it to select a set of sites that are likely to have the token. The site requests the token only from these sites, reducing the number of messages required to execute the CS. It is called a heuristic algorithm because sites are heuristically selected for sending token request messages.

When token request messages are sent only to a subset of sites, it is necessary that a requesting site sends a request message to a site that either holds the token or is going to obtain the token in the near future. Otherwise, there is a potential for deadlocks or starvation. Thus, one design requirement is that a site must select a subset of sites such that at least one of those sites is guaranteed to get the token in near future.

DATA STRUCTURES. A site S_i maintains two arrays, viz., SV_i[1..N] and SN_i[1..N], to store the information about sites in the system. These arrays store the state and the highest known sequence number for each site, respectively. Similarly, the token contains two such arrays as well (denoted by TSV[1..N] and TSN[1..N]). Sequence numbers are used to detect outdated requests. A site can be in one of the following states:

$\mathcal{R}$ –requesting the CS

$\mathcal{E}$ –executing the CS

$\mathcal{H}$ –holding the idle token

$\mathcal{N}$ –none of the above

The arrays are initialized as follows:

For every site S_i, $i = 1 \ldots N$ do
{SV_i[j]:= $\mathcal{N}$ for j= $N \ldots$ i; SV_i[j]:= $\mathcal{R}$ for j= i−1 ... 1;
SN_i[j]:= 0 for j = 1 ... N}

Initially, site S_1 is in state $\mathcal{H}$ (i.e., $S_1[1] := \mathcal{H}$).
For the token
{TSV[j]:= $\mathcal{N}$ and TSN[j]:= 0 for j= 1 ... N}

Note that arrays SV[1..N] of sites are initialized such that for any two sites S_i and S_j, either $SV_i[j] = \mathcal{R}$ or $SV_j[i] = \mathcal{R}$. Since the heuristic selects every site that is requesting the CS according to local information (i.e., the SV array), for any two sites that are requesting the CS concurrently, one will always send a token request message to the other. This ensures that sites are not isolated from each other and a site's request message reaches a site that either holds the token or is going to get the token in near future.

The Algorithm

Requesting the critical section

1. If the requesting site S_i does not have the token, then it takes the following actions:
   - It sets $SV_i[i] := \mathcal{R}$.
   - It increments $SN_i[i] := SN_i[i] + 1$.
   - It sends REQUEST(i, sn) message to all sites S_j for which $SV_i[j] = \mathcal{R}$. (sn is the updated value of $SN_i[i]$.)
2. When a site S_j receives the REQUEST(i, sn) message, it discards the message if $SN_j[i] \geq sn$ because the message is out dated. Otherwise, it sets $SN_j[i]$ to 'sn' and takes the following actions based on its own state:
   - $SV_j[j]=\mathcal{N}$: Set $SV_j[i] := \mathcal{R}$.
   - $SV_j[j]=\mathcal{R}$: If $SV_j[i] \neq \mathcal{R}$, then set $SV_j[i] := \mathcal{R}$ and send a REQUEST(j, $SN_j[j]$) message to S_i (else do nothing).
   - $SV_j[j]=\mathcal{E}$: Set $SV_j[i] := \mathcal{R}$.
   - $SV_j[j]=\mathcal{H}$: Set $SV_j[i] := \mathcal{R}$, TSV[i]:= $\mathcal{R}$, TSN[i]:= sn, $SV_j[j] := \mathcal{N}$, and send the token to site S_i.

Executing the critical section

3. S_i executes the CS after it has received the token. Before entering the CS, S_i sets $SV_i[i]$ to $\mathcal{E}$.

Releasing the critical section

4. Having finished the execution of the CS, site S_i sets $SV_i[i] := \mathcal{N}$ and TSV[i]:= $\mathcal{N}$, and updates its local and token vectors in the following way:

```
For all S_j, j= 1 to N do
    if SN_i[j] > TSN[j]
        then
            (* update token information from local information *)
            {TSV[j]:= SV_i[j]; TSN[j]:= SN_i[j]}
        else
            (* update local information from token information *)
            {SV_i[j]:= TSV[j]; SN_i[j]:= TSN[j]}
```

5. If $(\forall j :: SV_i[j]=\mathcal{N})$, then set $SV_i[i]:=\mathcal{H}$, else send the token to a site S_j such that $SV_i[j]=\mathcal{R}$.

The fairness of the algorithm depends upon the degree of fairness with which a site is selected for receiving the token, after a site is finished with the execution of its CS. Ideally, the token should not be granted to a site twice or more, while other sites are waiting for the token. Two arbitration rules to ensure fairness in scheduling the token among requesting sites are proposed in [20].

EXPLANATION. When a site requests access to the critical section, it sends request messages to all the sites which, according to its local state information, are also currently requesting the CS. The central idea behind the algorithm is that a site's request for the token reaches a site that has the token even though the site does not send request messages to all sites. This is a consequence of the following two factors: (1) How state vectors are initialized and updated and (2) How the sites are selected to send token request messages.

A site updates its state information (arrays SN and SV) from the request messages it receives from other sites and from the information in the token that it receives for critical section access. A site S_i sets $SV_i[j]$ to $\mathcal{R}$ when it receives a current request message from site S_j or when it receives the token, which has more up-to-date information about site S_j and TSV[j]= $\mathcal{R}$. Site S_i sets $SV_i[j]$ to $\mathcal{N}$ when it receives the token, which has more up-to-date information about site S_j and TSV[j]=$\mathcal{N}$. Note that if at site S_i, $SN_i[j] >$ TSN[j], then site S_i has more up-to-date information about site S_j, otherwise the token has more up-to-date information about site S_j. Since a site does not send request messages to all sites in the system and a site does not send messages to cancel its request messages, the token plays an important role in the dissemination of system state information.

CORRECTNESS

Theorem 6.6. A requesting site enters the CS in finite time.

Proof: Even though a requesting site does not send token request messages to all other sites, its token request message reaches a site that has the token in finite time. (The proof of this is very complicated and is therefore omitted. Interested readers are referred to [20].) When this site updates the token vector, entry for the requesting site is set to $\mathcal{R}$ and consequently, the requesting site gets the token in finite time. □