CHAPTER 5

THEORETICAL FOUNDATIONS

5.1 INTRODUCTION

A distributed system is a collection of computers that are spatially separated and do not share a common memory. The processes executing on these computers communicate with one another by exchanging messages over communication channels. The messages are delivered after an arbitrary transmission delay.

In this chapter, we first discuss the inherent limitations of distributed systems caused by the lack of common memory and a systemwide common clock that can be shared by all the processes. The rest of the chapter is devoted to the discussion of how to overcome these inherent limitations. The theoretical foundations developed in this chapter are the *most* fundamental to distributed computing and are made use of throughout the book.

5.2 INHERENT LIMITATIONS OF A DISTRIBUTED SYSTEM

In this section, we discuss the inherent limitations of distributed systems and their impact on the design and development of distributed systems. With the help of an example, we illustrate the difficulties encountered due to the limitations in distributed systems.

5.2.1 Absence of a Global Clock

In a distributed system, there exists no systemwide common clock (global clock). In other words, the notion of global time does not exist. A reader might think that this problem can be easily solved by either having a clock common to all the computers (processes) in the system or having synchronized clocks, one at each computer. Unfortunately, these two approaches cannot solve the problem for the following reasons.

Suppose a global (common) clock is available for all the processes in the system. In this case, two different processes can observe a global clock value at different instants due to unpredictable message transmission delays. Therefore, two different processes may falsely perceive two different instants in physical time to be a single instant in physical time.

On the other hand, if we provide each computer in the system with a physical clock and try to synchronize them, these physical clocks can drift from the physical time and the drift rate may vary from clock to clock due to technological limitations, Therefore, this approach can also have two different processes running at different computers that perceive two different instants in physical time as a single instant. Hence, we cannot have a system of perfectly synchronized clocks.

IMPACT OF THE ABSENCE OF GLOBAL TIME. The concept of temporal ordering of events pervades our thinking about systems and is integral to the design and development of distributed systems [12]. For example, an operating system is responsible for scheduling processes. A basic criterion used in scheduling is the temporal order in which requests to execute processes arrive (the arrival of a request is an event). Due to the absence of global time, it is difficult to reason about the temporal order of events in a distributed system. Hence, algorithms for a distributed system are more difficult to design and debug compared to algorithms for centralized systems. In addition, the absence of a global clock makes it harder to collect up-to-date information on the state of the entire system. The detailed description and analysis of this shortcoming is discussed next.

5.2.2 Absence of Shared Memory

Since the computers in a distributed system do not share common memory, an up-to-date state of the entire system is not available to any individual process. Up-to-date state of the system is necessary for reasoning about the system's behavior, debugging, recovering from failures (see Chap. 12), etc.

A process in a distributed system can obtain a *coherent* but partial view of the system or a complete but *incoherent* view of the system [13]. A view is said to be coherent if all the observations of different processes (computers) are made at the same physical time. A complete view encompasses the local views (local state) at all the computers and any messages that are in transit in the distributed system. A complete view is also referred to as a *global state*. Similarly, the global state of a distributed computation encompasses the local states of all the processes and any messages that are in transit between the processes. Because of the absence of a global clock in a distributed system, obtaining a coherent global state of the system is difficult.

The following simple situation illustrates the difficulty in obtaining a coherent global state while underlining the need for a coherent global state.

Example 5.1. Let S1 and S2 be two distinct sites (entities) of a distributed system (see Fig. 5.1) that maintain bank accounts A and B, respectively. A site in our example refers to a process. Knowledge of the global state of the system may be necessary to compute the net balance of both accounts. The initial state of the two accounts is shown in Fig. 5.1(a). Let site S1 transfer, say, $50 from account A to account B. During the collection of a global state, if site S1 records the state of A immediately after the debit has occurred, and site S2 saves the state of B before the fund transfer message has reached B, then the global system state will show $50 missing (see Fig. 5.1(b)). Note that the communication channel cannot record its state by itself. Hence, sites have to coordinate their state recording activities in order to record the channel state. On the other hand, if A's state is recorded immediately before the transfer and B's state is recorded after account B has been credited $50, then the global system state will show an extra $50 (see Fig. 5.1(c)).

We next present two schemes that implement an abstract notion of virtual time to order events in a distributed system. In addition, readers will find the application of these schemes throughout the book (see Secs. 6.5 and 6.6, Chap. 7, Secs. 12.10, 13.11, and 20.4). We also describe an application that makes use of one of the schemes.

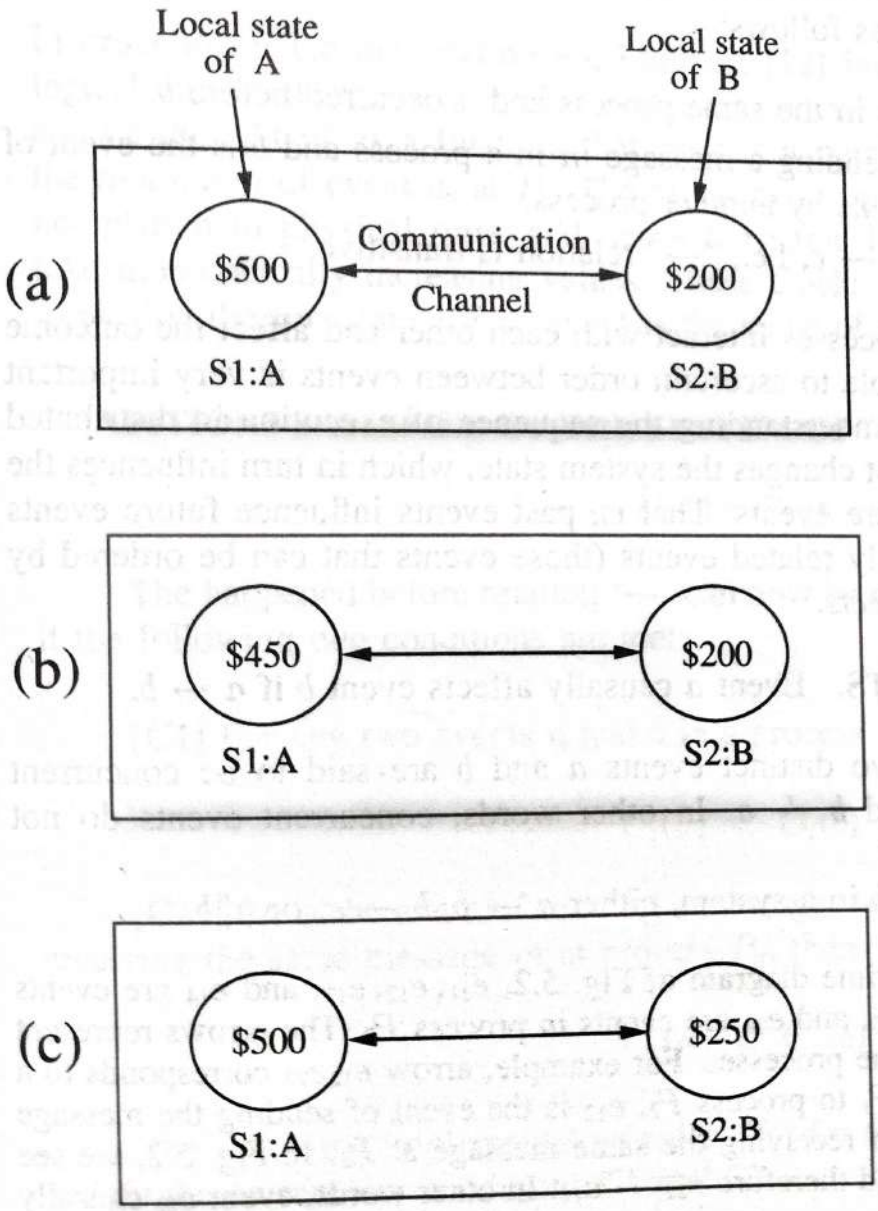

FIGURE 5.1
A distributed system with two sites.

5.3 LAMPORT'S LOGICAL CLOCKS

Lamport [12] proposed the following scheme to order events in a distributed system using logical clocks. The execution of processes is characterized by a sequence of events. Depending on the application, the execution of a procedure could be one event or the execution of an instruction could be one event. When processes exchange messages, sending a message constitutes one event and receiving a message constitutes one event.

Definitions

Due to the absence of perfectly synchronized clocks and global time in distributed systems, the order in which two events occur at two different computers cannot be determined based on the local time at which they occur. However, under certain conditions, it is possible to ascertain the order in which two events occur based solely on the behavior exhibited by the underlying computation. We next define a relation that orders events based on the behavior of the underlying computation.

HAPPENED BEFORE RELATION ($\rightarrow$). The *happened before* relation captures the causal dependencies between events, i.e., whether two events are causally related or not. The relation $\rightarrow$ is defined as follows:

- $a \rightarrow b$, if a and b are events in the same process and a occurred before b.
- $a \rightarrow b$, if a is the event of sending a message m in a process and b is the event of receipt of the same message m by another process.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$, i.e., "$\rightarrow$" relation is transitive.

In distributed systems, processes interact with each other and affect the outcome of events of processes. Being able to ascertain order between events is very important for designing, debugging, and understanding the sequence of execution in distributed computation. In general, an event changes the system state, which in turn influences the occurrence and outcome of future events. That is, past events influence future events and this influence among causally related events (those events that can be ordered by '$\rightarrow$') is referred to as *causal affects*.

CAUSALLY RELATED EVENTS. Event a causally affects event b if $a \rightarrow b$.

CONCURRENT EVENTS. Two distinct events a and b are said to be concurrent (denoted by $a \| b$) if $a \not\rightarrow b$ and $b \not\rightarrow a$. In other words, concurrent events do not causally affect each other.

For any two events a and b in a system, either $a \rightarrow b$, $b \rightarrow a$, or $a \| b$.

Example 5.2. In the space-time diagram of Fig. 5.2, e_{11}, e_{12}, e_{13}, and e_{14} are events in process P_1 and e_{21}, e_{22}, e_{23}, and e_{24} are events in process P_2. The arrows represent message transfers between the processes. For example, arrow $e_{12}e_{23}$ corresponds to a message sent from process P_1 to process P_2, e_{12} is the event of sending the message at P_1, and e_{23} is the event of receiving the same message at P_2. In Fig. 5.2, we see that $e_{22} \rightarrow e_{13}$, $e_{13} \rightarrow e_{14}$, and therefore $e_{22} \rightarrow e_{14}$. In other words, event e_{22} causally

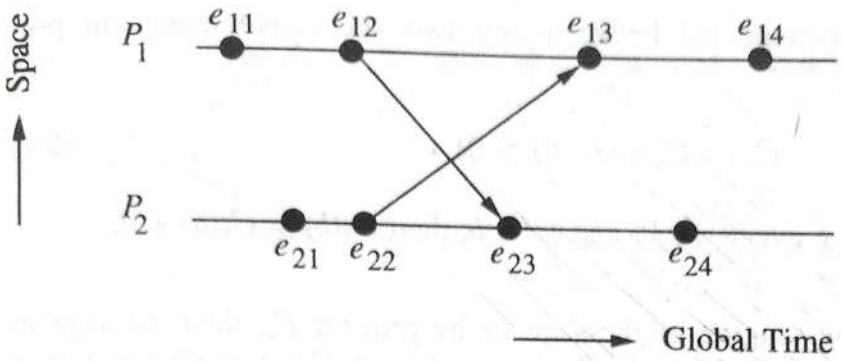

FIGURE 5.2
A space-time diagram.

affects event e_{14}. Note that whenever $a \rightarrow b$ holds for two events a and b, there exists a path from a to b which moves only forward along the time axis in the space-time diagram. Events e_{21} and e_{11} are concurrent even though e_{11} appears to have occurred before e_{21} in real (global) time for a global observer.

Logical Clocks

In order to realize the relation $\rightarrow$, Lamport [12] introduced the following system of logical clocks. There is a clock C_i at each process P_i in the system. The clock C_i can be thought of as a function that assigns a number $C_i(a)$ to any event a, called the *timestamp* of event a, at P_i. The numbers assigned by the system of clocks have no relation to physical time, and hence the name logical clocks. The logical clocks take monotonically increasing values. These clocks can be implemented by counters. Typically, the timestamp of an event is the value of the clock when it occurs.

CONDITIONS SATISFIED BY THE SYSTEM OF CLOCKS. For any events a and b:

$$\text{if } a \rightarrow b, \text{ then } C(a) < C(b)$$

The happened before relation '$\rightarrow$' can now be realized by using the logical clocks if the following two conditions are met:

[C1] For any two events a and b in a process P_i, if a occurs before b, then

$$C_i(a) < C_i(b)$$

[C2] If a is the event of sending a message m in process P_i and b is the event of receiving the same message m at process P_j, then

$$C_i(a) < C_j(b)$$

The following implementation rules (IR) for the clocks guarantee that the clocks satisfy the correctness conditions C1 and C2:

[IR1] Clock C_i is incremented between any two successive events in process P_i:

$$C_i := C_i + d \quad (d > 0) \tag{5.1}$$

If a and b are two successive events in P_i and $a \rightarrow b$, then $C_i(b) = C_i(a) + d$.

[IR2] If event a is the sending of message m by process P_i, then message m is assigned a timestamp $t_m = C_i(a)$ (note that the value of $C_i(a)$ is obtained after applying rule IR1). On receiving the same message m by process P_j, C_j is set to a value greater than or equal to its present value and greater than t_m.

$$C_j := \max(C_j, t_m + d) \quad (d > 0) \tag{5.2}$$

Note that the message receipt event at P_j increments C_j as per rule IR1. The updated value of C_j is used in Eq. 5.2. Usually, d in Eqs. 5.1 and 5.2 has a value of 1.

Lamport's happened before relation, $\rightarrow$, defines an irreflexive partial order among the events. The set of all the events in a distributed computation can be totally ordered (the ordering relation is denoted by $\Rightarrow$) using the above system of clocks as follows: If a is any event at process P_i and b is any event at process P_j then $a \Rightarrow b$ if and only if either

$$C_i(a) < C_j(b) \quad \text{or}$$

$$C_i(a) = C_j(b) \quad \text{and} \quad P_i \prec P_j$$

where $\prec$ is any arbitrary relation that totally orders the processes to break ties. A simple way to implement $\prec$ is to assign unique identification numbers to each process and then $P_i \prec P_j$, if $i < j$.

Lamport's mutual exclusion algorithm, discussed in Sec. 6.6, illustrates the use of the ability to totally order the events in a distributed system.

Example 5.3. Figure 5.3 gives an example of how logical clocks are updated under Lamport's scheme. Both the clock values C_{P_1} and C_{P_2} are assumed to be zero initially and d is assumed to be 1. e_{11} is an internal event in process P_1 which causes C_{P_1} to be incremented to 1 due to IR1. Similarly, e_{21} and e_{22} are two events in P_2 resulting in C_{P_2} = 2 due to IR1. e_{16} is a message send event in P_1 which increments C_{P_1} to 6 due to IR1. The message is assigned a timestamp = 6. The event e_{25}, corresponding to the receive event of the above message, increments the clock C_{P_2} to 7 (max(4+1,6+1)) due to rules IR1 and IR2. Similarly, e_{24} is a send event in P_2. The message is assigned a timestamp = 4. The event e_{17} corresponding to the receive event of the above message increments the clock C_{P_1} to 7 (max(6+1, 4+1)) due to rules IR1 and and IR2.

VIRTUAL TIME. Lamport's system of logical clocks implements an approximation to global/physical time, which is referred to as virtual time. Virtual time advances along

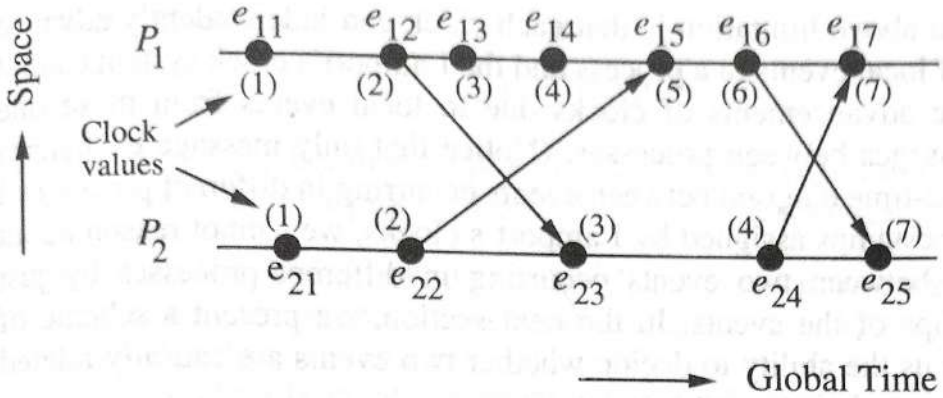

FIGURE 5.3
How Lamport's logical clocks advance.

with the progression of events and is therefore discrete. If no events occur in the system, virtual time stops, unlike physical time which continuously progresses. Therefore, to wait for a virtual time instant to pass is risky as it may never occur [16].

5.3.1 A Limitation of Lamport's Clocks

Note that in Lamport's system of logical clocks, if $a \rightarrow b$ then $C(a) < C(b)$. However, the reverse is not necessarily true if the events have occurred in different processes. That is, if a and b are events in different processes and $C(a) < C(b)$, then $a \rightarrow b$ is not necessarily true; events a and b may be causally related or may not be causally related. Thus, Lamport's system of clocks is not powerful enough to capture such situations. The next example illustrates this limitation of Lamport's clocks.

Example 5.4. Figure 5.4 shows a computation over three processes. Clearly, $C(e_{11}) < C(e_{22})$ and $C(e_{11}) < C(e_{32})$. However, we can see from the figure that event e_{11} is causally related to event e_{22} but not to event e_{32}, since a path exists from e_{11} to e_{22} but not from e_{11} to e_{32}. Note that the initial clock values are assumed to be zero and d of equations 5.1 and 5.2 is assumed to equal 1. In other words, in Lamport's system of clocks, we can guarantee that if $C(a) < C(b)$ then $b \not\rightarrow a$ (i.e., the future cannot influence the past), however, we cannot say whether events a and b are causally related or not (i.e., whether there exists a path between a and b that moves only forward along the time axis in the space-time diagram) by just looking at the timestamps of the events.

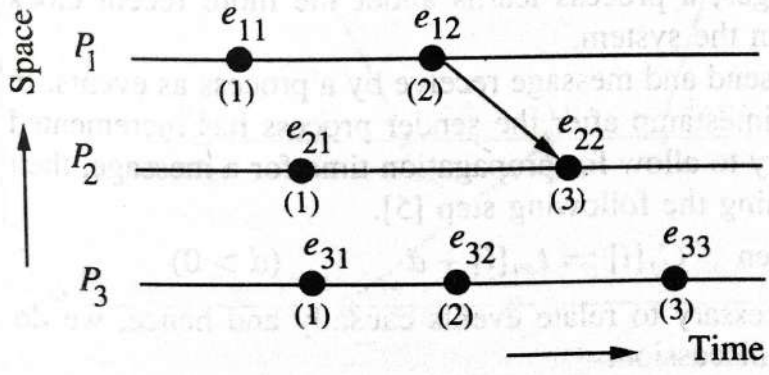

FIGURE 5.4
A space-time diagram.

The reason for the above limitation is that each clock can independently advance due to the occurrence of local events in a process and the Lamport's clock system cannot distinguish between the advancements of clocks due to local events from those due to the exchange of messages between processes. (Notice that only message exchanges establish paths in a space-time diagram between events occurring in different processes.) Therefore, using the timestamps assigned by Lamport's clocks, we cannot reason about the causal relationship between two events occurring in different processes by just looking at the timestamps of the events. In the next section, we present a scheme of vector clocks that gives us the ability to decide whether two events are causally related or not by simply looking at their timestamps.

5.4 VECTOR CLOCKS

The system of vector clocks was independently proposed by Fidge [5] and Mattern [16]. A concept similar to vector clocks was proposed previously by Strom and Yemini [38] for keeping track of transitive dependencies among processes for recovery purposes. Let n be the number of processes in a distributed system. Each process P_i is equipped with a clock C_i, which is an integer vector of length n. The clock C_i can be thought of as a function that assigns a vector $C_i(a)$ to any event a. $C_i(a)$ is referred to as the timestamp of event a at P_i. $C_i[i]$, the ith entry of C_i, corresponds to P_i's own logical time. $C_i[j]$, $j \neq i$ is P_i's best guess of the logical time at P_j. More specifically, at any point in time, the jth entry of C_i indicates the time of occurrence of the last event at P_j which "happened before" the current point in time at P_i. This "happened before" relationship could be established directly by communication from P_j to P_i or indirectly through communication with other processes.

The implementation rules for the vector clocks are as follows [16]:

[IR1] Clock C_i is incremented between any two successive events in process P_i

$$C_i[i] := C_i[i] + d \qquad (d > 0) \tag{5.3}$$

[IR2] If event a is the sending of the message m by process P_i, then message m is assigned a vector timestamp $t_m = C_i(a)$; on receiving the same message m by process P_j, C_j is updated as follows:

$$\forall k,\ C_j[k] := \max(C_j[k], t_m[k]) \tag{5.4}$$

Note that, on the receipt of messages, a process learns about the more recent clock values of the rest of the processes in the system.

In rule IR1, we treat message send and message receive by a process as events. In rule IR2, a message is assigned a timestamp after the sender process has incremented its clock due to IR1. If it is necessary to allow for propagation time for a message, then IR2 can be performed after performing the following step [5].

$$\text{If} \quad C_j[i] \leq t_m[i] \quad \text{then} \quad C_j[i] := t_m[i] + d \qquad (d > 0)$$

However, the above step is not necessary to relate events causally and hence, we do not make use of it in the following discussion.

Assertion. At any instant,

$$\forall i,\ \forall j:\ C_i[i] \geq C_j[i]$$

The proof is obvious because no process $P_j \neq P_i$ can have more up-to-date knowledge about the clock value of process i and clocks are monotonically nondecreasing.

Example 5.5. Figure 5.5 illustrates an example of how clocks advance and the dissemination of time occurs in a system using vector clocks (d is assumed to be 1 and all clock values are initially zero).

Event e_{11} is an internal event in process P_1 that causes $C_1[1]$ to be incremented to 1 due to IR1. e_{12} is a message send event in P_1 which causes $C_1[1]$ to be incremented to 2 due to IR1. e_{22} is a message receive event in P_2 that causes $C_2[2]$ to be incremented to 2 due to IR1, and $C_2[1]$ to be set to 2 due to IR2. e_{31} is a send event in P_3 which causes $C_3[3]$ to be incremented to 1 due to IR1. Event e_{23}, a receive event in P_2, causes $C_2[2]$ to be incremented to 3 due to IR1, and $C_2[3]$ to be set to 1 due to IR2. e_{24} is a send event in P_2 and e_{13} is the corresponding receive event. Note that $C_1[3]$ is set to 1 due to IR2, and process P_1 has learned that the local clock value at P_3 is at least 1 through a message from P_2.

Vector timestamps can be compared as follows [16]. For any two vector timestamps t^a and t^b of events a and b, respectively:

Equal:	$t^a = t^b$	iff	$\forall i,\ t^a[i] = t^b[i]$;
Not Equal:	$t^a \neq t^b$	iff	$\exists i,\ t^a[i] \neq t^b[i]$;
Less Than or Equal:	$t^a \leq t^b$	iff	$\forall i,\ t^a[i] \leq t^b[i]$;
Not Less Than or Equal To:	$t^a \not\leq t^b$	iff	$\exists i,\ t^a[i] > t^b[i]$;
Less Than:	$t^a < t^b$	iff	$(t^a \leq t^b \ \wedge\ t^a \neq t^b)$;
Not Less Than:	$t^a \not< t^b$	iff	$\neg(t^a \leq t^b \ \wedge\ t^a \neq t^b)$;
Concurrent:	$t^a \| t^b$	iff	$(t^a \not< t^b \ \wedge\ t^b \not< t^a)$;

Note that the relation "$\leq$" is a partial order. However, the relation "$\|$" is not a partial order because it is not transitive.

CAUSALLY RELATED EVENTS. Events a and b are causally related, if $t^a < t^b$ or $t^b < t^a$. Otherwise, these events are concurrent.

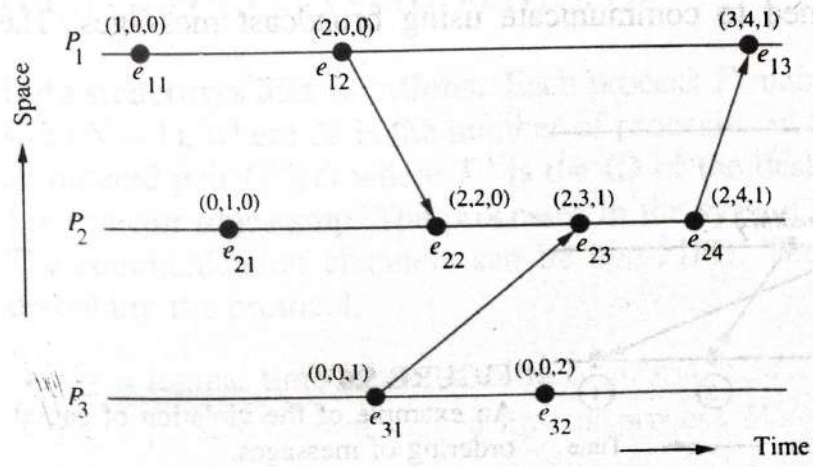

FIGURE 5.5
Dissemination of time in vector clocks.

In the system of vector clocks,

$$a \rightarrow b \quad \text{iff} \quad t^a < t^b \tag{5.5}$$

Thus, the system of vector clocks allows us to order events and decide whether two events are causally related or not by simply looking at the timestamps of the events. If we know the processes where the events occur, the above test can be further simplified (see Problem 5.1). Note that an event e can causally affect another event e' (events e_{12} and e_{22} in Fig. 5.5) if there exists a path that propagates the (local) time knowledge of event e to event e' [16].

In the next section, we present an application of vector clocks for the causal ordering of messages.

5.5 CAUSAL ORDERING OF MESSAGES

The causal ordering of messages was first proposed by Birman and Joseph [1] and was implemented in ISIS. The causal ordering of messages deals with the notion of maintaining the same causal relationship that holds among "message send" events with the corresponding "message receive" events. In other words, if $Send(M_1) \rightarrow Send(M_2)$ (where $Send(M)$ is the event sending message M), then every recipient of both messages M_1 and M_2 must receive M_1 before M_2. The causal ordering of messages should not be confused with the causal ordering of events, which deals with the notion of causal relationship among the events. In a distributed system, the causal ordering of messages is not automatically guaranteed. For example, Fig. 5.6 shows a violation of causal ordering of messages in a distributed system. In this example, $Send(M_1) \rightarrow Send(M_2)$. However, $M2$ is delivered before $M1$ to process P_3. (The numbers circled indicate the correct causal order to deliver messages.)

Techniques for the causal ordering of messages are useful in developing distributed algorithms and may simplify the algorithms themselves. For example, for applications such as replicated database systems, it is important that every process in charge of updating a replica receives the updates in the same order to maintain the consistency of the database [1]. In the absence of causal ordering of messages, each and every update must be checked to ensure that it does not violate the consistency constraints.

We next describe two protocols that make use of vector clocks for the causal ordering of messages in distributed systems. The first protocol is implemented in ISIS [2], wherein the processes are assumed to communicate using broadcast messages. The

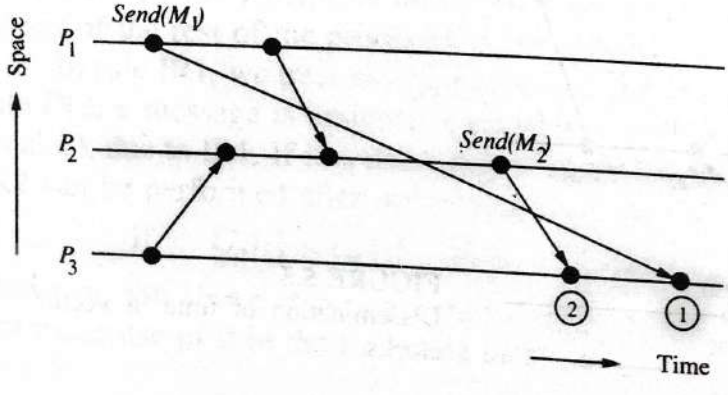

FIGURE 5.6
An example of the violation of causal ordering of messages.

second protocol does not require processes to communicate only through broadcast messages. Both protocols require that the messages be delivered reliably (lossless and uncorrupted).

BASIC IDEA. The basic idea of both the protocols is to deliver a message to a process only if the message immediately preceding it has been delivered to the process. Otherwise, the message is not delivered immediately but is buffered until the message immediately preceding it is delivered. A vector accompanying each message contains the necessary information for a process to decide whether there exists a message preceding it.

BIRMAN-SCHIPER-STEPHENSON PROTOCOL

1. Before broadcasting a message m, a process P_i increments the vector time $VT_{P_i}[i]$ and timestamps m. Note that $(VT_{P_i}[i] - 1)$ indicates how many messages from P_i precede m.
2. A process $P_j \neq P_i$, upon receiving message m timestamped VT_m from P_i, delays its delivery until both the following conditions are satisfied.
   a. $VT_{P_j}[i] = VT_m[i] - 1$
   b. $VT_{P_j}[k] \geq VT_m[k] \ \forall k \in \{1, 2, ..., n\} - \{i\}$
   where n is the total number of processes.
   Delayed messages are queued at each process in a queue that is sorted by vector time of the messages. Concurrent messages are ordered by the time of their receipt.
3. When a message is delivered at a process P_j, VT_{P_j} is updated according to the vector clocks rule IR2 (see Eq. 5.4).

Step 2 is the key to the protocol. Step 2(a) ensures that process P_j has received all the messages from P_i that preceed m. Step 2(b) ensures that P_j has received all those messages received by P_i before sending m. Since the event ordering relation "$\rightarrow$" imposed by vector clocks is acyclic, the protocol is deadlock free.

The Birman-Schiper-Stephenson causal ordering protocol requires that the processes communicate through broadcast messages. We next describe a protocol proposed by Schiper, Eggli, and Sandoz [20], which does not require processes to communicate only by broadcast messages.

SCHIPER-EGGLI-SANDOZ PROTOCOL

Data structures and notations. Each process P maintains a vector denoted by V_P of size $(N-1)$, where N is the number of processes in the system. An element of V_P is an ordered pair (P', t) where P' is the ID of the destination process of a message and t is a vector timestamp. The processes in the system are assumed to use vector clocks. The communication channels can be non-FIFO. The following notations are used in describing the protocol:

- t_M = logical time at the sending of message M.
- t_{P_i} = present/current logical time at process P_i.

THE PROTOCOL

Sending of a message M from process P_1 to process P_2

- Send message M (timestamped t_M) along with V_P_1 to process P_2.
- Insert pair (P_2, t_M) into V_P_1. If V_P_1 contains a pair (P_2, t), it simply gets overwritten by the new pair (P_2, t_M). Note that the pair (P_2, t_M) was not sent to P_2. Any future message carrying the pair (P_2, t_M) cannot be delivered to P_2 until $t_M < t_{P_2}$.

Arrival of a message M at process P_2

If V_M (the vector accompanying message M) does not contain any pair (P_2, t)
then the message can be delivered
else (* A pair (P_2, t) exists in V_M *)
If $t \not< t_{P_2}$ then
the message cannot be delivered (*it is buffered for later delivery*)
else
the message can be delivered.

If message M can be delivered at process P_2, then the following three actions are taken:

1. Merge V_M accompanying M with V_P_2 in the following manner:
   - If $(\exists\ (P, t) \in V_M,$ such that $P \neq P_2)$ and $(\forall (P', t) \in V_P_2,\ P' \neq P)$, then insert (P, t) into V_P_2. This rule performs the following: if there is no entry for process P in V_P_2, and V_M contains an entry for process P, insert that entry into V_P_2.
   - $\forall P, P \neq P_2$, if $((P, t) \in V_M) \wedge ((P, t') \in V_P_2)$, then the algorithm takes the following actions: $(P, t) \in V_P_2$ can be substituted by the pair (P, t_{sup}) where t_{sup} is such that $\forall i, t_{\text{sup}}[i] = \max(t[i], t'[i])$. This rule is simply performing the step in Eq. 5.4 for each entry in V_P2.

   Due to the above two actions, the algorithm satisfies the following two conditions:

   a. No message can be delivered to P as long as $t' < t_P$ is not true.
   b. No message can be delivered to P as long as $t < t_P$ is not true.
2. Update site P_2's logical clock.
3. Check for the buffered messages that can now be delivered since local clock has been updated.

A pair (P, t) can be deleted from the vector maintained at a site after ensuring that the pair (P, t) has become obsolete (i.e., no longer needed) (see Problem 5.3).

5.6 GLOBAL STATE

We now address the problem of collecting or recording a coherent (consistent) global state in distributed systems, a challenging task due to the absence of a global clock and

shared memory. First, we reexamine the bank account example of Sec. 5.2 to develop the correctness criteria for a consistent global state recording algorithm.

Figure 5.7 shows the stages of a computation when \$50 is transferred from account A to account B. The communication channels C1 and C2 are assumed to be FIFO.

Suppose the state of account A at site S1 was recorded when the global state was 1 (see Fig. 5.7). Now assume that the global state changes to 2, and the state of communication channels C1 and C2 and of account B are recorded when the global state is 2. Then the composite of all the states recorded would show account A's balance as \$500, account B's balance as \$200, and a message in transit to transfer \$50. In other words, an extra amount of \$50 would appear in the global state. The reason for this inconsistency is that A's state was recorded before the message was sent and the channel C1's state was recorded after the message was sent. Therefore, a recorded global state may be inconsistent if $n < n'$ where n is the number of messages sent by A along

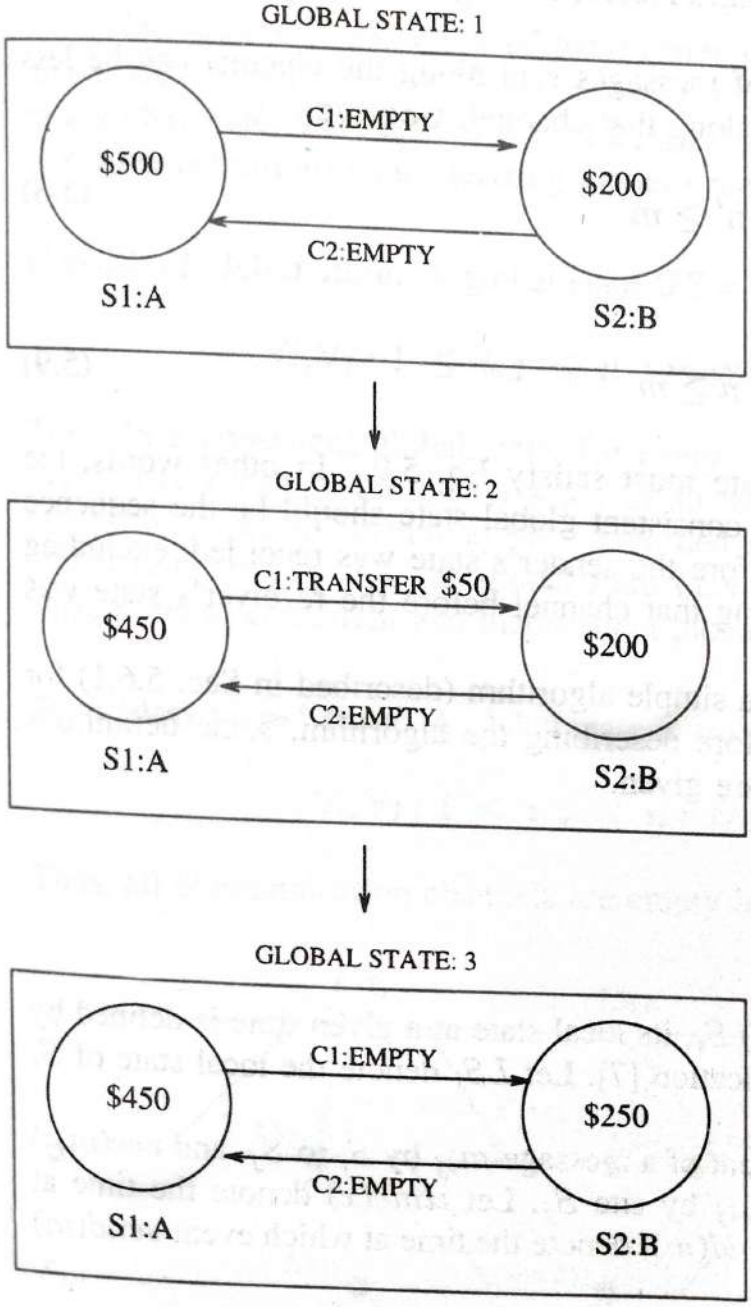

FIGURE 5.7
Global-states and their transitions in the bank accounts example.

the channel before A's state was recorded and n' is the number of messages sent by A along the channel before the channel's state was recorded.

Suppose channels state were recorded when the global state was 1 and A and B's states were recorded when the global state was 2. Then the composite of A, B, and the channels state will show a deficit of \$50. This means that the recorded global state may be inconsistent if $n > n'$. Hence, a consistent global state requires

$$n = n' \tag{5.6}$$

On similar lines, one can show that a consistent global state requires

$$m = m' \tag{5.7}$$

where m' = number of messages received along the channel before account B's state was recorded and m = number of messages received along the channel by B before the channel's state was recorded.

Since in no system the number of messages sent along the channel can be less than the number of messages received along that channel, we have

$$n' \geq m \tag{5.8}$$

From Eqs. 5.6 and 5.8, we get

$$n \geq m \tag{5.9}$$

Therefore, a consistent global state must satisfy Eq. 5.9. In other words, the state of a communication channel in a consistent global state should be the sequence of messages sent along that channel before the sender's state was recorded, excluding the sequence of messages received along that channel before the receiver's state was recorded [3].

The above observations result in a simple algorithm (described in Sec. 5.6.1) for recording a consistent global state. Before describing the algorithm, some definitions for formally describing a system state are given.

Definitions

LOCAL STATE. For a site (computer) S_i, its local state at a given time is defined by the local context of the distributed application [7]. Let LS_i denote the local state of S_i at any time.

Let $send(m_{ij})$ denote the send event of a message m_{ij} by S_i to S_j, and $rec(m_{ij})$ denote the receive event of message m_{ij} by site S_j. Let $time(x)$ denote the time at which state x was recorded and $time(send(m))$ denote the time at which event $send(m)$ occurred.

For a message m_{ij} sent by S_i to S_j, we say that

- $send(m_{ij}) \in LS_i$ iff $time(send(m_{ij})) < time(LS_i)$.
- $rec(m_{ij}) \in LS_j$ iff $time(rec(m_{ij})) < time(LS_j)$.

For the local states LS_i and LS_j of any two sites S_i and S_j, we define two sets of messages. These sets contain messages that were sent from site S_i to site S_j.

Transit: $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$
Inconsistent: $inconsistent(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \notin LS_i \wedge rec(m_{ij}) \in LS_j\}$

GLOBAL STATE. A global state, GS, of a system is a collection of the local states of its sites; That is, $GS = \{LS_1, LS_2, \ldots, LS_n\}$ where n is the number of sites in the system.

Note that any collection of local states of sites need not represent a consistent global state. Consistency has a connotation that for every effect or outcome recorded in a global state, the cause of the effect must also be recorded in the global state. We next give definitions characterizing global states.

Consistent global state. A global state $GS = \{LS_1, LS_2, \ldots, LS_n\}$ is *consistent* iff

$$\forall i, \forall j : 1 \leq i, j \leq n :: inconsistent(LS_i, LS_j) = \Phi$$

Thus, in a consistent global state, for every received message a corresponding send event is recorded in the global state. In an inconsistent global state, there is at least one message whose receive event is recorded but its send event is not recorded in the global state. In Fig. 5.8, the global state $\{LS_{12}, LS_{23}, LS_{33}\}$ and $\{LS_{11}, LS_{22}, LS_{32}\}$ correspond to consistent and inconsistent global states, respectively.

Transitless global state. A global state is *transitless* if and only if

$$\forall i, \forall j : 1 \leq i, j \leq n :: transit(LS_i, LS_j) = \Phi$$

Thus, all communication channels are empty in a transitless global state.

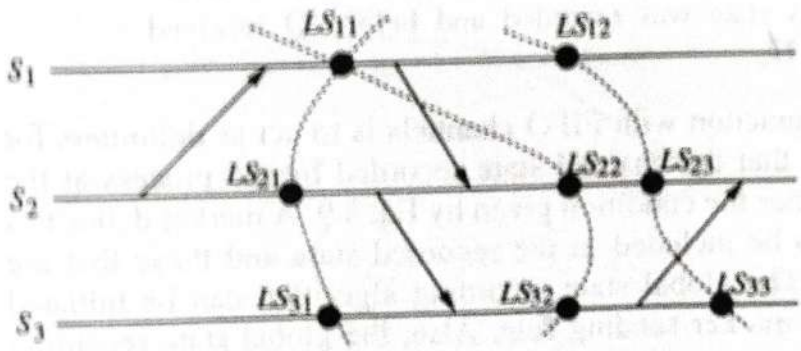

FIGURE 5.8
Global states in a distributed computation.

Strongly consistent global state. A global state is *strongly consistent* if it is consistent and transitless. In a strongly consistent state, not only the send events of all the recorded received events are recorded, but the receive events of all the recorded send events are also recorded. Thus, a strongly consistent state corresponds to a consistent global state in which all channels are empty. In Fig. 5.8, the global state $\{LS_{11}, LS_{21}, LS_{31}\}$ is a strongly consistent global state.

A note. While the definitions of this section are defined for a group of sites, they can also be applied to a group of cooperating processes by simply replacing sites with processes in the definitions. For instance, $GS = \{LS_1, LS_2, \ldots, LS_n\}$ represents a global state of n cooperating processes, where LS_i is the local state of process P_i.

5.6.1 Chandy-Lamport's Global State Recording Algorithm

Chandy and Lamport [3] were the first to propose a distributed algorithm to capture a consistent global state. The algorithm uses a marker (a special message) to initiate the algorithm and the marker has no effect on the underlying computation. The communication channels are assumed to be FIFO. The recorded global state is also referred to as a *snapshot* of the system state.

Marker Sending Rule for a process P

- P records its state.
- For each outgoing channel C from P on which a marker has not been already sent, P sends a marker along C before P sends further messages along C.

Marker Receiving Rule for a process Q. On receipt of a marker along a channel C:

```
If Q has not recorded its state
  then
      begin
          Record the state of C as an empty sequence.
          Follow the "Marker Sending Rule."
      end
  else
      Record the state of C as the sequence of messages received
      along C after Q's state was recorded and before Q received
      the marker along C.
```

The role of markers in conjunction with FIFO channels is to act as delimiters for the messages in the channels so that the channel state recorded by the process at the receiving end of the channel satisfies the condition given by Eq. 5.9. A marker delineates messages into those that need to be included in the recorded state and those that are not to be recorded in the state. The global state recording algorithm can be initiated by any process by executing the marker sending rule. Also, the global state recording

algorithm can be initiated by several processes concurrently with each process getting its own version of a consistent global state. Each initiation needs its own unique marker (such as <process-id, sequence number>) and different initiations by a process can be distinguished by a local sequence number. A simple way to collect all the recorded information is for each process to send the information it recorded to the initiator of the recording process. The identification of the initiator process can be easily carried by the marker.

A Note on the Collected Global State

It is possible that the global state recorded by the above algorithm is not identical to any of the global states the system actually went through during the computation. This can happen because a site can change its state asynchronously before the markers sent by it are received by other sites. If the state changes while the markers are in transit, the composite of all the states recorded will not correspond to the state of the system at any instant of time. The question of the significance of the recorded global state if the system may have never passed through it arises. Before discussing the utility of a collected global state, we state a result from [3] without giving its proof. This result gives an important property of a collected global state.

Suppose the algorithm is initiated when the system is in global state S_i and it terminates when the system is in global state S_t. Let S_c denote the collected global state by the algorithm, and Seq denote the sequence of actions which take the system from state S_i to S_t. Then, there exists a sequence Seq' that is a permutation of Seq such that S_c can be reached from S_i by executing a prefix of Seq' and S_t can be reached from S_c by executing the rest of the actions in Seq' (see Fig. 5.9).

The usefulness of the recorded global state lies in its ability to detect *stable properties* (a stable property is one that persists) such as the termination of a computation and a deadlock among processes. Note that if a stable property holds before the recording algorithm begins execution, it continues to hold (unless resolved in the case of a deadlock), and will therefore be included in the recorded global state.

Even though the global state recording algorithm can be used for termination detection, it is an expensive way of doing it. In Sec. 5.8, we give an efficient algorithm for termination detection.

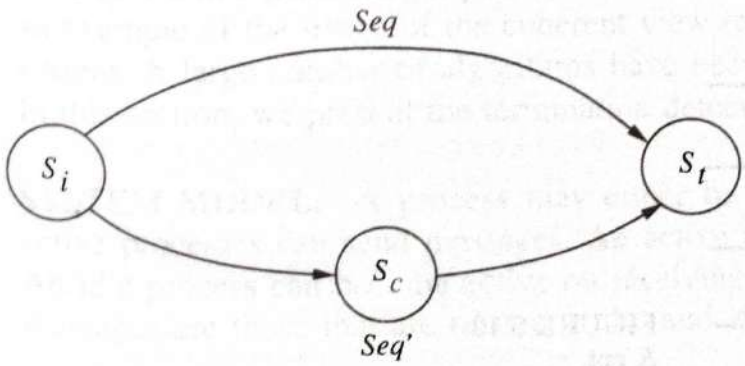

FIGURE 5.9
A property of a collected global state.

5.7 CUTS OF A DISTRIBUTED COMPUTATION

The notion of a *cut* captures a global state in a distributed computation. A cut is a graphical representation of a global state in the history of a distributed computation. A consistent cut is a graphical representation of a consistent global state in the history of a distributed computation.

CUT. A *cut* of a distributed computation is a set $C = \{c_1, c_2, \ldots, c_n\}$, where c_i is the cut event at site S_i in the history of the distributed computation.

Graphically, a cut is a zig-zig line that connects the corresponding cut events in the time-space diagram. For example, in Fig. 5.10, events c_1, c_2, c_3, and c_4 form a cut.

If a cut event c_i at site S_i is S_i's local state at that instant, then clearly a cut denotes a global state of the system.

CONSISTENT CUT. Let e_k denote an event at site S_k. A cut $C = \{c_1, c_2, \ldots, c_n\}$ is a consistent cut iff

$$\forall S_i, \forall S_j, \nexists e_i, \nexists e_j \quad \text{such that} \quad (e_i \rightarrow e_j) \wedge (e_j \rightarrow c_j) \wedge (e_i \not\rightarrow c_i)$$

where $c_i \in C$ and $c_j \in C$.

That is, a cut is a consistent cut if every message that was received before a cut event was sent before the cut event at the sender site in the cut. For example, the cut in Fig. 5.10 is not consistent because the message sent by S_2 is received before c_3 but the corresponding send did not occur before event c_2. That is, $e \rightarrow e'$, $e' \rightarrow c_3$, and $e \not\rightarrow c_2$.

Theorem 5.1. A cut $C = \{c_1, c_2, \ldots, c_n\}$, is a consistent cut if and only if no two cut events are causally related, that is, $\forall c_i \, \forall c_j :: \neg(c_i \rightarrow c_j) \wedge \neg(c_j \rightarrow c_i)$ [16].

Thus, a set of concurrent cut events form a consistent cut and vice versa. We will omit a detailed proof of this theorem, but will give the intuition behind it. A detailed proof can be found in [18].

Consider two events c_1 and c_2 in Fig. 5.11 which are not concurrent because $c_1 \rightarrow c_2$. Clearly, c_1 and c_2 do not lie on a consistent cut. On the other hand, if two events are concurrent, it implies that no message sent after one of them has been received before the other one. Thus, concurrent cut events form a consistent cut.

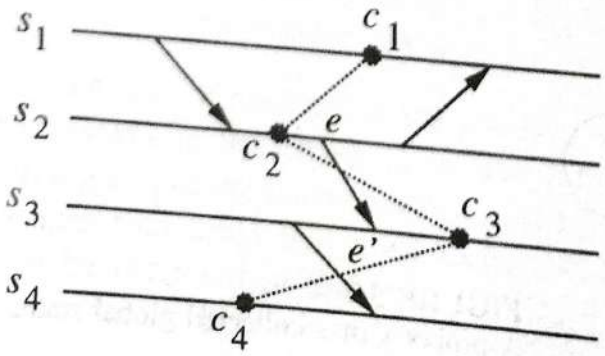

FIGURE 5.10
A cut.

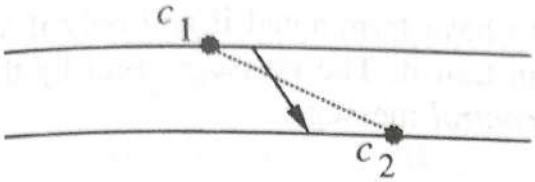

FIGURE 5.11
A cut where cut events are not concurrent.

TIME OF A CUT. If each cut event is assigned a vector timestamp, then a cut in a distributed computation can be assigned a timestamp in the following manner:

If $C = \{c_1, c_2, \ldots, c_n\}$ is a cut, where c_i is a cut event at site S_i with vector timestamp VT_{c_i}, then the vector time of the cut, VT_c, is defined as follows:

$$VT_c = \sup(VT_{c_1}, VT_{c_2}, \ldots, VT_{c_n})$$

where sup is a componentwise maximum operation. That is,

$$VT_c[i] = max(VT_{c_1}[i], VT_{c_2}[i], \ldots, VT_{c_n}[i]).$$

Theorem 5.2. If $C = \{c_1, c_2, \ldots, c_n\}$ is a cut with a vector time VT_c, then the cut is consistent iff

$$VT_c = (VT_{c_1}[1], VT_{c_2}[2], \ldots, VT_{c_n}[n])$$

Proof. If C is a consistent cut, then from the previous theorem, events $c_1, c_2, \ldots, c_n$ are concurrent. In other words, $\forall i \forall j,\ c_i[i] \geq c_j[i]$. Therefore,

$$sup(VT_{c_1}, VT_{c_2}, \ldots, VT_{c_n}) = (VT_{c_1}[1], VT_{c_2}[2], \ldots, VT_{c_n}[n])$$

For sufficiency, note that $VT_c = (VT_{c_1}[1], VT_{c_2}[2], \ldots, VT_{c_n}[n])$ implies that no message sent after a cut event c_i has been received before another cut event c_j. Thus, cut events $c_1, c_2, \ldots, c_n$ form a consistent cut. □

5.8 TERMINATION DETECTION

A distributed computation generally consists of a set of cooperating processes which communicate with each other by exchanging messages. In the case of a distributed computation, it is important to know when the computation has terminated. The problem of termination detection arises in many distributed algorithms and computations. For example, how to determine when an election, a deadlock detection, a deadlock resolution, or a token generating algorithm has terminated. Termination detection, in fact, is an example of the usage of the coherent view (consistent global state) of a distributed system. A large number of algorithms have been developed for termination detection. In this section, we present the termination detection algorithm proposed by Huang [8].

SYSTEM MODEL. A process may either be in an *active* state or idle state. Only active processes can send messages. An active process may become idle at any time. An idle process can become active on receiving a *computation* message. Computation messages are those that are related to the underlying computation being performed by

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:

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

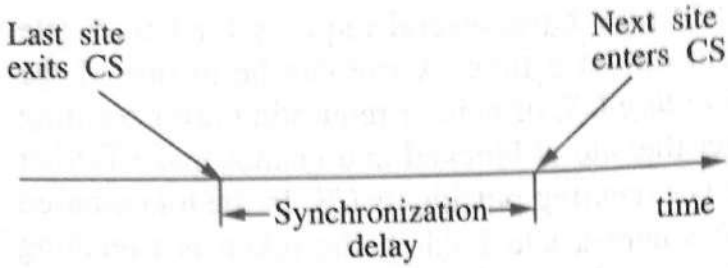

FIGURE 6.1
Synchronization delay.

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.

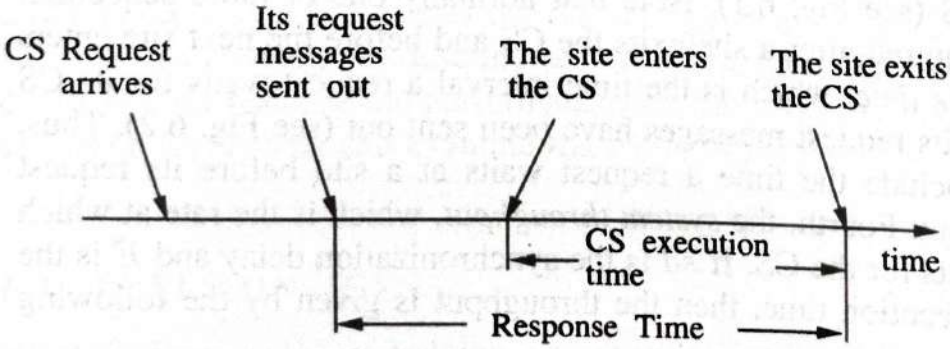

FIGURE 6.2
Response time.

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

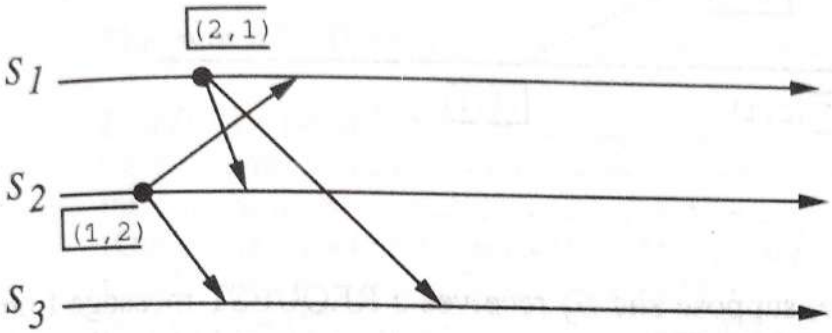

FIGURE 6.3
Sites S_1 and S_2 are making requests for the CS.

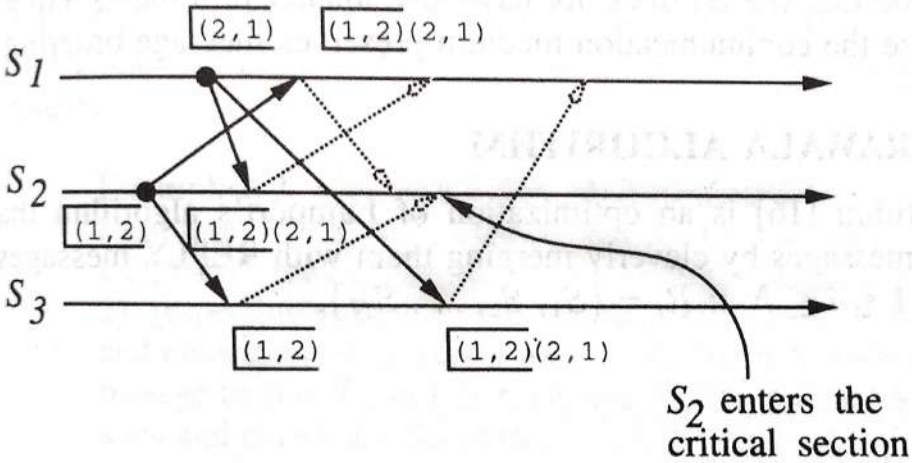

FIGURE 6.4
Site S_2 enters the CS.

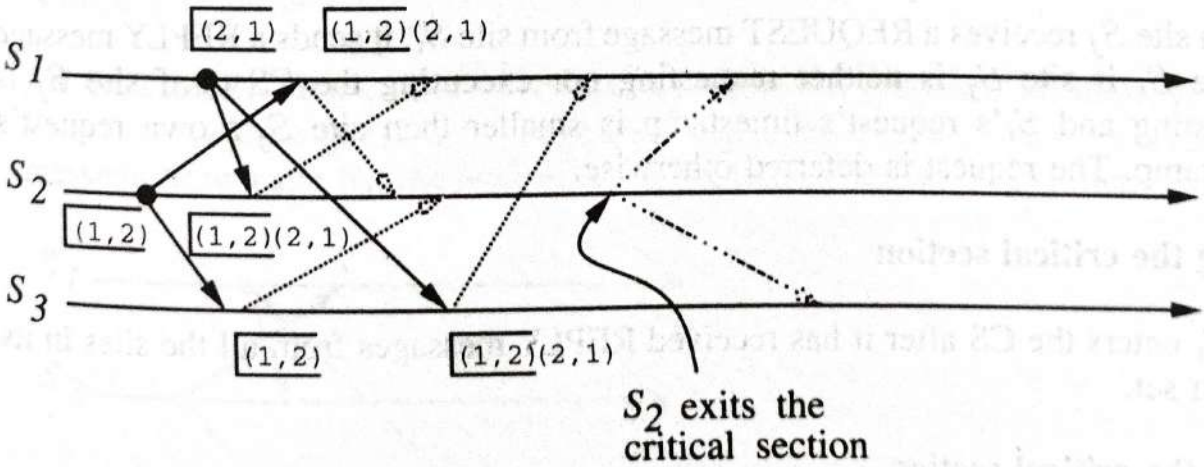

FIGURE 6.5
Site S_2 exits the CS and sends RELEASE messages.

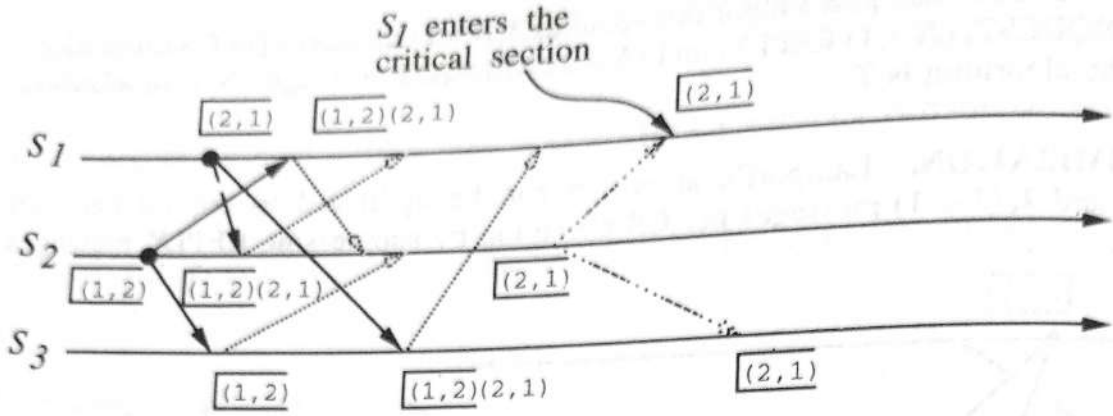

FIGURE 6.6
Site S_1 enters the CS.

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

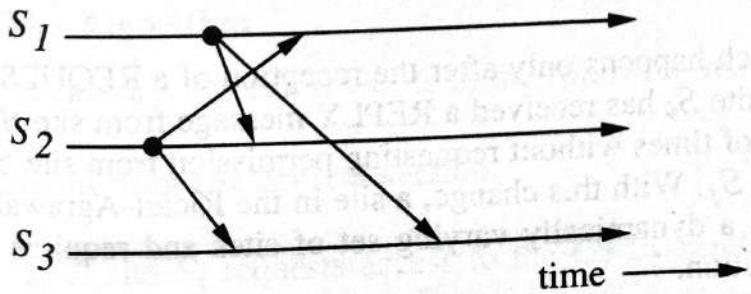

FIGURE 6.7
Sites S_1 and S_2 are making requests for the CS.

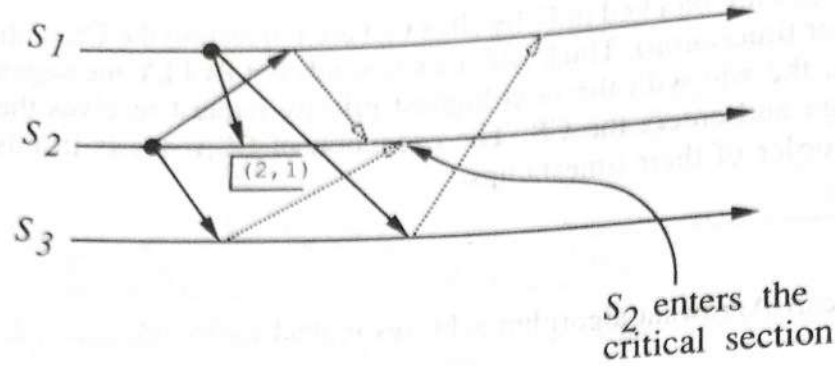

FIGURE 6.8
Site S_2 enter the CS.

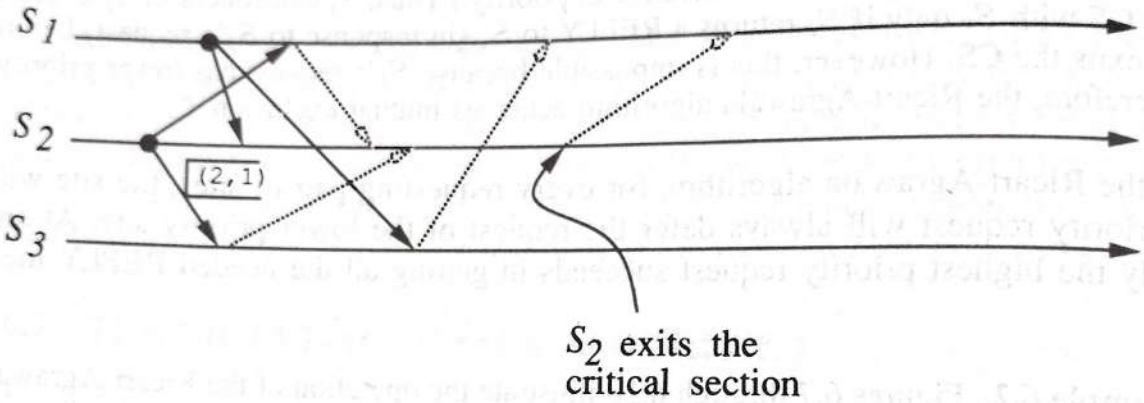

FIGURE 6.9
Site S_2 exits the CS and sends a REPLY message to S_1.

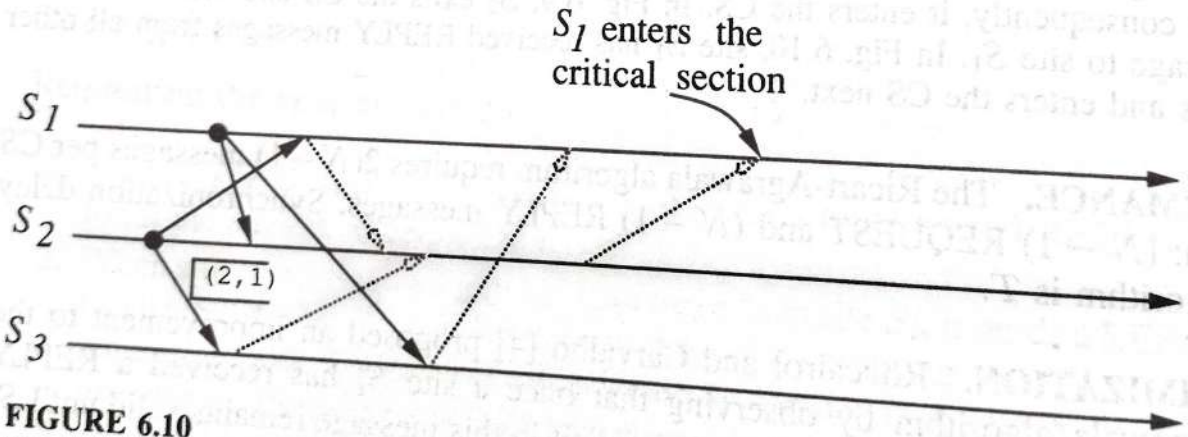

FIGURE 6.10
Site S_1 enters the CS.

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: $(\forall i\ \forall j : i \neq j, 1 \leq i, j \leq N :: \mathrm{R}_i \cap \mathrm{R}_j \neq \phi)$

M2: $(\forall i : 1 \leq i \leq N :: S_i \in \mathrm{R}_i)$

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.

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 nth 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 $\ldots$ 1;
SN_i[j]:= 0 for j = 1 $\ldots$ N}

PERFORMANCE. A salient feature of this algorithm is that a site can access the critical section without communicating with every site in the system. In low to moderate loads, the average message traffic is $N/2$ because each site sends REQUEST messages to half the sites on average. It increases to N at high loads as most sites will be requesting the CS (which is reflected at site S_i by $SV_i[j] = \mathcal{R}$ for most j's). The synchronization delay in this algorithm is T. An interesting feature of this algorithm is that it adapts itself to the environment of nonuniform traffic of CS requests and to statistical fluctuations in the traffic of CS requests to further reduce the number of messages exchanged.

The algorithm does not have any additional message overhead for the dissemination of state information, except for a slightly larger token message (which is passed comparatively infrequently). Since entries in the token state array (TSV) are either $\mathcal{R}$ or $\mathcal{N}$, this array can be a binary array.

6.13 RAYMOND'S TREE-BASED ALGORITHM

In Raymond's tree-based algorithm [14], sites are logically arranged as a directed tree such that the edges of the tree are assigned directions toward the site (root of the tree) that has the token. Every site has a local variable *holder* that points to an immediate neighbor node on a directed path to the root node. Thus, *holder* variables at the sites define logical tree structure among the sites. If we follow *holder* variables at sites, every site has a directed path leading to the site holding the token. At root site, *holder* points to itself. An example of a tree configuration is shown in Fig. 6.12.

Every site keeps a FIFO queue, called *request_q*, which stores the requests of those neighboring sites that have sent a request to this site, but have not yet been sent the token.

The Algorithm

Requesting the critical section

1. When a site wants to enter the CS, it sends a REQUEST message to the node along the directed path to the root, provided it does not hold the token and its *request_q*

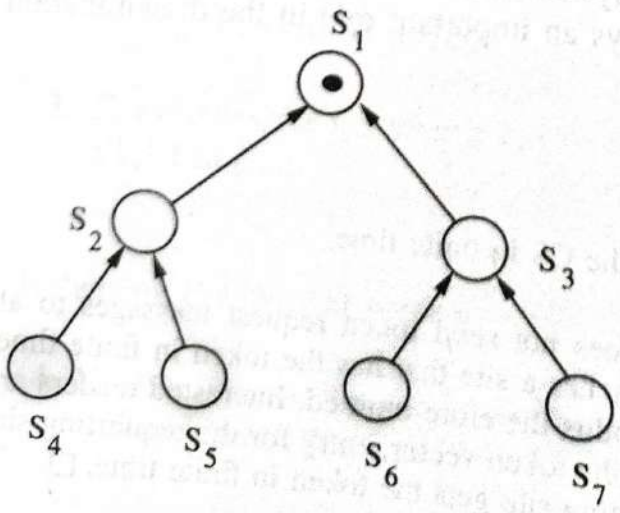

FIGURE 6.12
Sites arranged in a tree configuration.

is empty. It then adds its request to its *request_q*. (Note that a nonempty *request_q* at a site indicates that the site has sent a REQUEST message to the root node for the top entry in its *request_q*.)

2. When a site on the path receives this message, it places the REQUEST in its *request_q* and sends a REQUEST message along the directed path to the root provided it has not sent out a REQUEST message on its outgoing edge (for a previously received REQUEST on its *request_q*).
3. When the root site receives a REQUEST message, it sends the token to the site from which it received the REQUEST message and sets its *holder* variable to point at that site.
4. When a site receives the token, it deletes the top entry from its *request_q*, sends the token to the site indicated in this entry, and sets its *holder* variable to point at that site. If the *request_q* is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by *holder* variable.

Executing the critical section

5. A site enters the CS when it receives the token and its own entry is at the top of its *request_q*. In this case, the site deletes the top entry from its *request_q* and enters the CS.

Releasing the critical section. After a site has finished execution of the CS, it takes the following actions:

6. If its *request_q* is nonempty, then it deletes the top entry from its *request_q*, sends the token to that site, and sets its *holder* variable to point at that site.
7. If the *request_q* is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by the *holder* variable.

CORRECTNESS. The algorithm is free from deadlocks because the acyclic nature of tree configuration eliminates the possibility of *circular wait* among requesting sites. We next show that the algorithm is free from starvation.

Theorem 6.7. A requesting site enters the CS in finite time.

Proof: A formal correctness proof is long and complex. Thus, an informal correctness proof is provided. For a formal proof, readers are referred to the original paper [14].

The essence of proof is based on the following two facts: (1) a site serves requests in its *request_q* in the FCFS order and (2) every site has a path leading to the site that has the token. Due to the latter fact and Step 2 of the algorithm, when a site S_i is making a request, there exists a chain of requests from site S_i to site S_h, which holds the token. Let the chain be denoted by S_i, S_{i1}, S_{i2}, ..., S_{ik-1}, S_{ik}, S_h. When S_h receives a REQUEST message from S_{ik}, it sends the token to S_{ik}. There are two possibilities: S_{ik-1}'s request is at the top of S_{ik}'s *request_q* or it is not at

the top. In the first case, S_{ik} sends the token to site S_{ik-1}. In the second case, S_{ik} sends the token to the site, say S_j, at the top of its $request_q$ and also sends it a REQUEST message (see Step 4 of the algorithm). This extends the chain of requests to S_i, S_{i1}, S_{i2}, ..., S_{ik-1}, S_{ik}, S_j, ..., S_l, where site S_l executes the CS next. Note that due to fact (1) above, all the sites in the chain S_j, ..., S_l will execute the CS at most once before the token is returned to site S_{ik}. Thus, site S_{ik} sends the token to S_{ik-1} in finite time. Likewise, S_{ik-1} sends the token to S_{ik-2} in finite time and so on. Eventually, S_{i1} sends the token to S_i. Consequently, a requesting site eventually receives the token. □

Example 6.4. In Fig. 6.13, site S_5 sends a REQUEST message to S_2, which propagates it to the root S_1. Root S_1 sends the token to S_2 which in turn sends the token to S_5 (Fig. 6.14). The token travels along the same path traveled by the REQUEST message (but in the opposite direction) and it also reverses the direction of the edges on the path. Consequently, the site that executes the CS last becomes the new root. (See Fig. 6.15.)

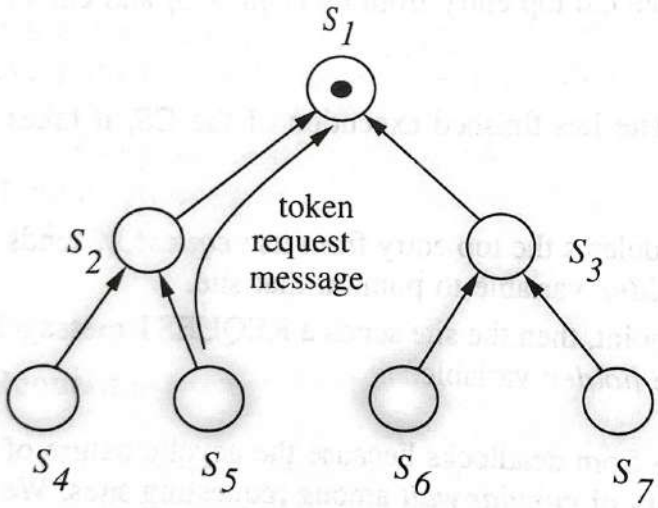

FIGURE 6.13
Site S_5 is requesting the token.

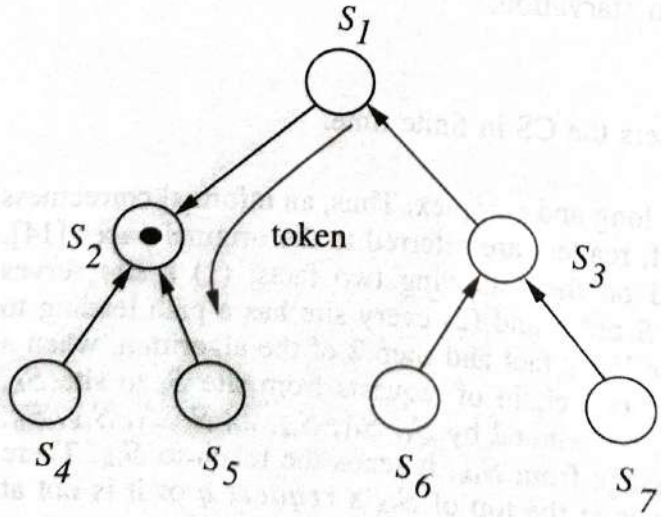

FIGURE 6.14
The token is in transit to S_5.

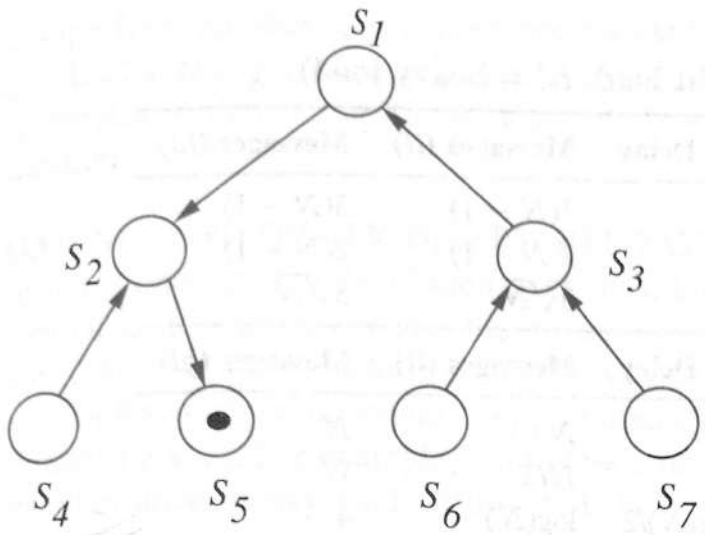

FIGURE 6.15
State after S_5 has received the token.

PERFORMANCE. The average message complexity of Raymond's algorithm is $O(\log N)$ because the average distance between any two nodes in a tree with N nodes is $O(\log N)$. Synchronization delay in this algorithm is $(T \log N)/2$ because the average distance between two sites to successively execute the CS is $(\log N)/2$.

Raymond's algorithm can use *greedy strategy*, where a site receiving the token executes the CS even though its request is not at the top of its *request_q*. It is important to note that in heavy loads, the synchronization delay in this case becomes T because a site executes the CS every time the token is transferred. Needless to say, the greedy strategy has an adverse effect on the fairness of the algorithm and can cause starvation.

6.14 A COMPARATIVE PERFORMANCE ANALYSIS

In this section, we present a performance comparison among various mutual exclusion algorithms. Table 6.1 summarizes the response time, the number of messages required, and synchronization delay for mutual exclusion algorithms discussed in this chapter.

6.14.1 Response Time

At low loads, there is hardly any contention among requests for the CS. Therefore, the response time under low load conditions for many algorithms is simply a round trip message delay ($= 2T$) to acquire permission or token plus the time to execute the CS ($= E$). In Raymond's algorithm, the average distance between a requesting site and the site holding the token is $(\log N)/2$. Thus, the average round trip delay to acquire the token is $T(\log N)$.

As the load is increased, response time increases in all mutual exclusion algorithms because contention for access to the CS increases. Different algorithms see different increases in response time with respect to load. A closed form expression of response time as a function of load is not known for these algorithms. The response time under heavy load conitions is discussed in Sec. 6.14.4 (see Table 6.2 under "Maximum average response time").

TABLE 6.1
A comparison of performance (ll = light load, hl = heavy load)

NON-TOKEN	Resp. time (ll)	Sync Delay	Messages (ll)	Messages (hl)
Lamport	$2T+E$	T	$3(N-1)$	$3(N-1)$
Ricart-Agrawala	$2T+E$	T	$2(N-1)$	$2(N-1)$
Maekawa	$2T+E$	$2T$	$3\sqrt{N}$	$5\sqrt{N}$
TOKEN	**Resp. time (ll)**	**Sync Delay**	**Messages (ll)**	**Messages (hl)**
Suzuki and Kasami	$2T+E$	T	N	N
Singhal's Heuristic	$2T+E$	T	$N/2$	N
Raymond	$T(\log N)+E$	$T \log(N)/2$	$\log(N)$	4

6.14.2 Synchronization Delay

Recall from Sec. 6.3.2 that the synchronization delay is the delay due to the sequential message exchanges required after a site leaves the CS and before the next site enters the CS. In many of algorithms, a site exiting the CS directly sends a REPLY message or the token to the next site to enter the CS, resulting in a synchronization delay of T. Maekawa's algorithm has a synchronization delay of $2T$ because a site exiting the CS unlocks an arbiter site (by sending it a RELEASE message) and then the arbiter site sends a GRANT message to the next site to enter the CS (thus, subject to two serial message delays). In Raymond's algorithm, two sites that consecutively execute the CS can be located at any position in the tree. The token propagates serially along the edges in the tree to the site to enter the CS next. Since the average distance between two nodes in a typical tree with N nodes is $(\log N)/2$, the synchronization delay in this algorithm is $T(\log N)/2$. However, when structure-based token algorithms (such as Raymond [14]) use a greedy strategy, synchronization delay in heavy loads becomes T.

6.14.3 Message Traffic

The Lamport's, Ricart-Agrawala, and Suzuki-Kasami algorithms respectively require $3*(N-1)$, $2*(N-1)$, and N messages per CS execution, irrespective of the load. In other algorithms, the number of messages needed depends upon the load and is discussed below:

Light Load. Raymond's algorithm requires $\log N$ messages per CS execution. This is because a request for a token must travel along the edges of the tree from the requester node to the root node and the token must travel back from the root node to the requester node. Note that the average distance between the root and a node in a typical tree with N nodes is $(\log N)/2$. Singhal's heuristic algorithm requires $N/2$ message per CS execution under light load conditions.

Heavy Load. In Raymond's algorithm, message traffic decreases as load increases due to the sharing of request messages at high loads—when a site receives a request

message from another site, it does not forward it if it has already sent a request message. Under heavy load conditions, message traffic in Singhal's heuristic algorithm degenerates to that in the Suzuki-Kasami algorithm and requires N messages per CS execution.

LOAD NONUNIFORMLY DISTRIBUTED OVER SITES. When the rate of CS requests is nonuniformly distributed over sites, Singhal's heuristic algorithm [20] has the potential to substantially reduce the message traffic by adapting to the prevailing system state. Likewise, piggybacking and greedy strategies can substantially reduce message traffic in Raymond's algorithm [14] provided requests for the CS are localized to a few neighboring sites. For example, with these strategies, the message traffic in Raymond's algorithm under heavy load conditions is four messages per CS execution.

6.14.4 Universal Performance Bounds

Due to their idiosyncrasies, different mutual exclusion algorithms have widely varying performance characteristics. Nevertheless, we can determine universal performance bounds for mutual exclusion algorithms. These bounds depend upon system characteristics, not upon any particular mutual exclusion algorithm. In Table 6.2, we summarize these bounds.

6.15 SUMMARY

The problem of distributed mutual exclusion has received considerable attention over the last decade. In this chapter, a common framework has been provided with which to

TABLE 6.2
Universal performance bounds

Metrics	Bound	Explanation
Minimum synchronization delay	T	At least one message has to be transferred before the next site can receive the notification of permission to enter the CS.
Maximum throughput	$1/(T+E)$	The minimum time interval between two successive CS execution is $T+E$.
Minimum response time	$2T+E$	One round trip message delay to acquire permission or the token plus time to execute the CS.
Maximum average response time	$N(T+E)$	This occurs at heavy loads (when all sites always have pending requests). Note that system executes requests at a rate $r = 1/(T+E)$. From Little's Law, response time $= N/r = N(T+E)$.