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.

coherent- all parts go well ε each other.

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.

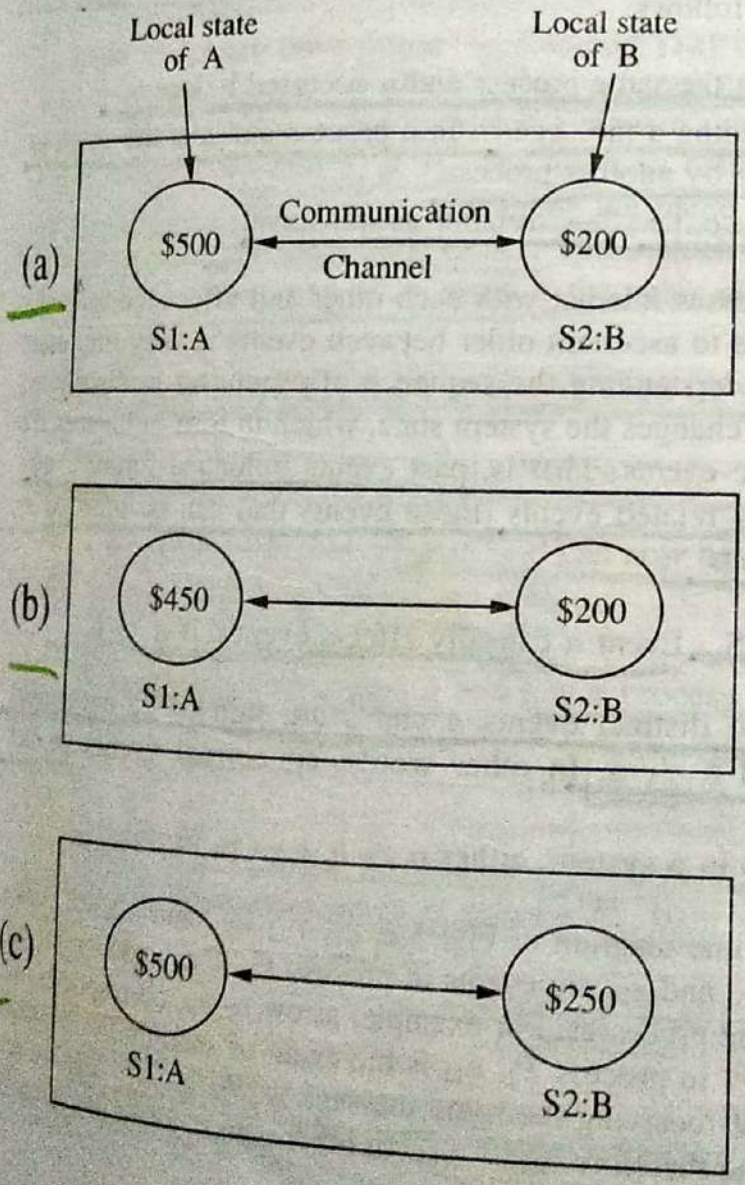

FIGURE 5.1
A distributed system with two sites.

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

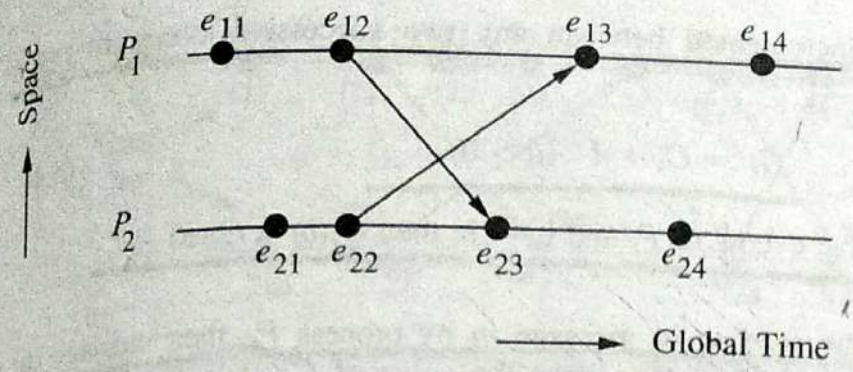

FIGURE 5.2
A space-time diagram.

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

$$\begin{cases} C_i(a) < C_j(b) & \text{or} \\ C_i(a) = C_j(b) & \text{and} \quad P_i \prec P_j \end{cases}$$

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

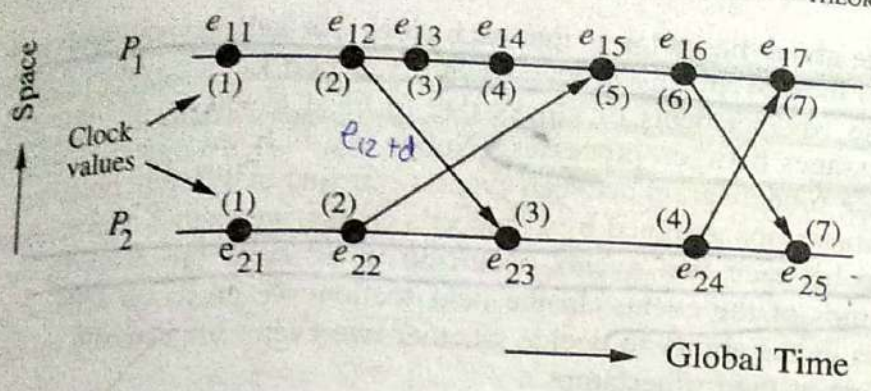

FIGURE 5.3
How Lamport's logical clocks advance.

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.

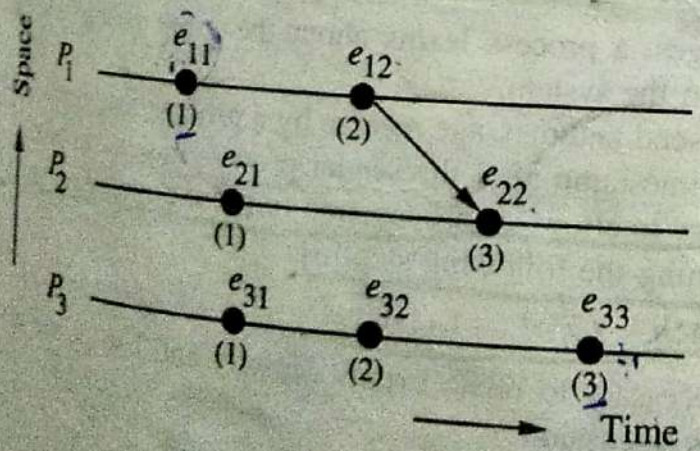

FIGURE 5.4
A space-time diagram.

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

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.

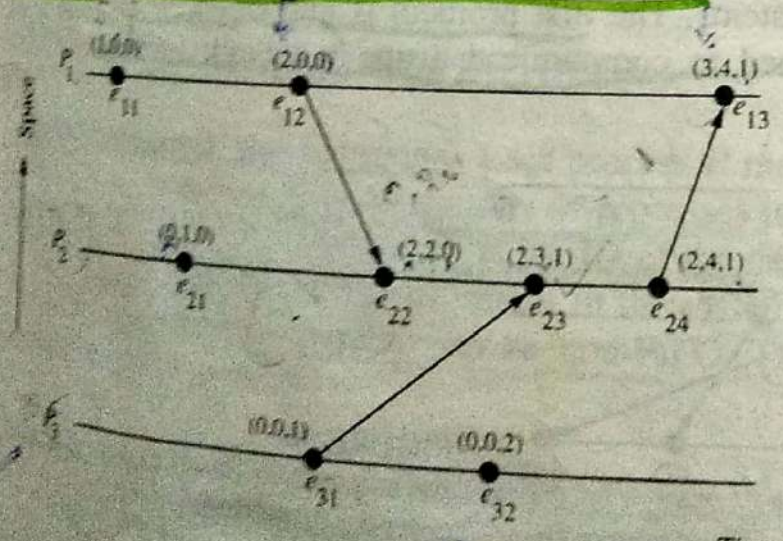

FIGURE 5.5
Dissemination of time in vector clocks.

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

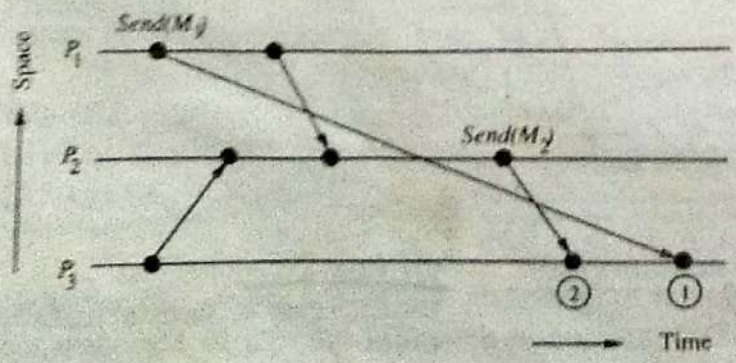

FIGURE 5.6
An example of the violation of causal ordering of messages.

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. $VT_{P_j}[k] \geq VT_m[k] \quad \forall k \in \{1, 2, \ldots, n\} - \{i\}$

   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 $\forall i, t_{sup}[i] = \max(t[i], t'[i])$. 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

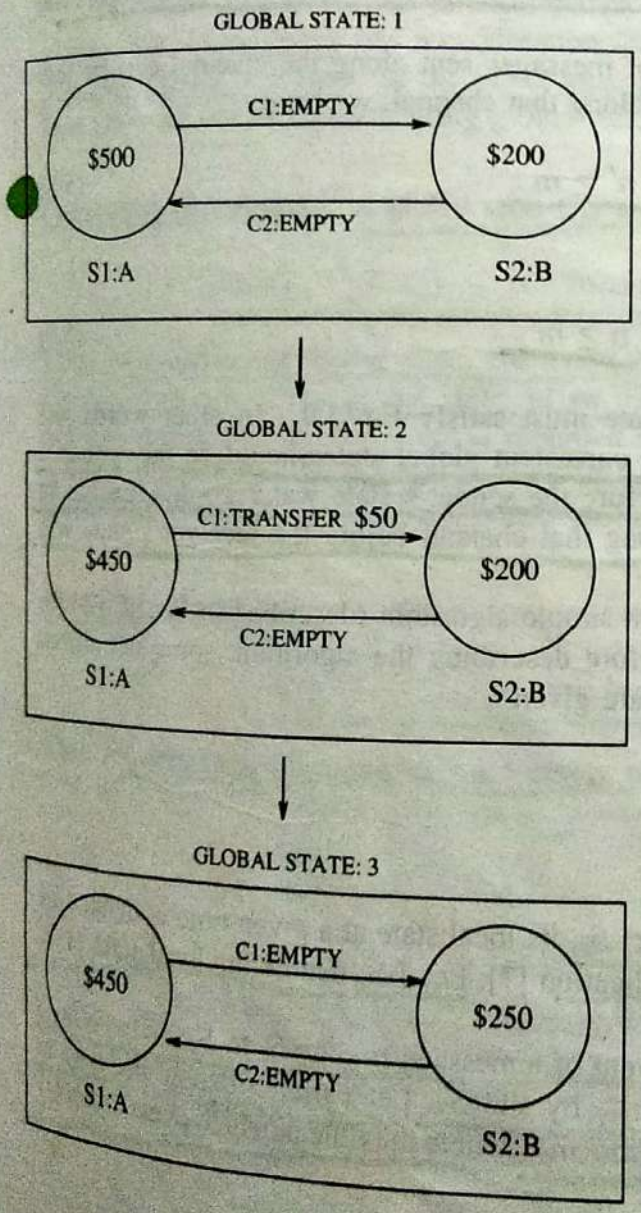

FIGURE 5.7
Global-states and their transitions in the bank accounts example.

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.

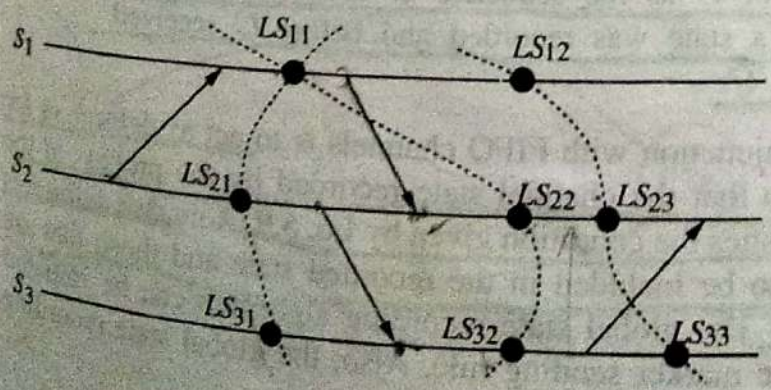

FIGURE 5.8
Global states in a distributed computation.

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.

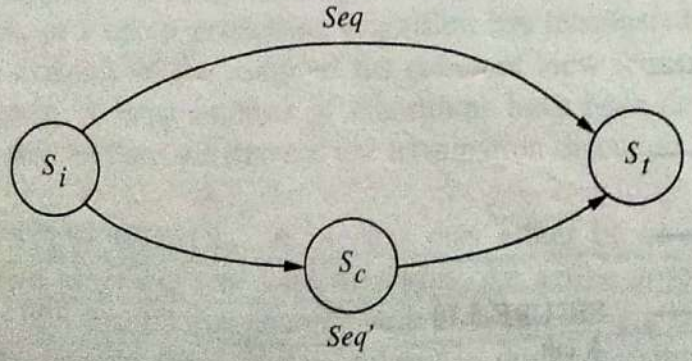

FIGURE 5.9
A property of a collected global state.

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.

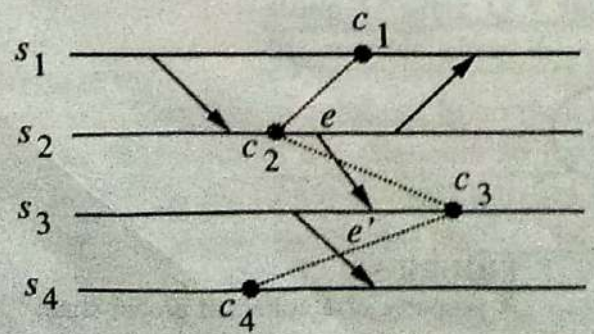

FIGURE 5.10
A cut.

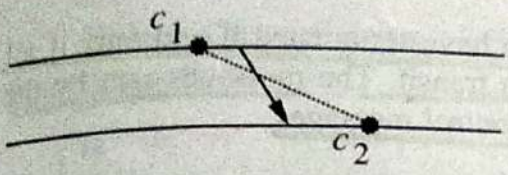

FIGURE 5.11
A cut where cut events are not concurrent.

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

the cooperating processes. A computation is said to have terminated if and only if all the processes are idle and there are no messages in transit. The messages sent by the termination detection algorithm are referred to as *control* messages.

BASIC IDEA. One of the cooperating processes monitors the computation and is called the *controlling agent*. Initially all processes are idle, the controlling agent's weight equals 1, and the weight of the rest of the processes is zero. The computation starts when the controlling agent sends a computation message to one of the processes. Any time a process sends a message, the process's weight is split between itself and the process receiving the message (the message carries the weight for the receiving process). The weight received along with a message is added to the weight of the process. Thus, the algorithm assigns a weight W $(0 < W \leq 1)$ to each active process (including the controlling agent) and to each message in transit. The weights assigned are such that, at any time, they satisfy an invariant $\sum W = 1$. On finishing the computation, a process sends its weight to the controlling agent, which adds the received weight to its own weight. When the weight of the controlling agent is once again equal to 1, it concludes that the computation has terminated.

NOTATIONS. The following notations are used in the algorithm:

- $B(DW)$ = Computation message sent as a part of the computation and DW is the weight assigned to it.
- $C(DW)$ = Control message sent from the processes to the controlling agent and DW is the weight assigned to it.

Huang's Termination Detection Algorithm

Rule 1. The controlling agent or an active process having weight W may send a computation message to a process P by doing:

```
Derive W1 and W2 such that
W1 + W2 = W,    W1 > 0,   W2 > 0;
W := W1;
Send B(W2) to P;
```

Rule 2. On receipt of $B(DW)$, a process P having weight W does:

```
W := W + DW;
If P is idle, P becomes active;
```

Rule 3. An active process having weight W may become idle at any time by doing:

```
Send C(W) to the controlling agent;
W := 0;
(The process becomes idle);
```

Rule 4. On receiving $C(DW)$, the controlling agent having weight W takes the following actions:

$W := W + DW$;
If $W = 1$, conclude that the computation has terminated.

PROOF OF CORRECTNESS. Let

A: The set of weights of all the active processes.
B: The set of weights of all the computation messages in transit.
C: The set of weights of all the control messages in transit.
W_c: Weight of the controlling agent.

The following P_1 and P_2 are the invariants.

$$P_1: \; W_c + \sum_{W \in (A \cup B \cup C)} W = 1$$

$$P_2: \; \forall W \in (A \cup B \cup C), \; W > 0$$

Hence,

$$W_c = 1 \; \Rightarrow \sum_{W \in (A \cup B \cup C)} W = 0 \text{ by } P_1$$

$$\sum_{W \in (A \cup B \cup C)} W = 0 \Rightarrow; \; (A \cup B \cup C) = \phi \text{ by } P_2$$

$$(A \cup B \cup C) = \phi \; \Rightarrow \; (A \cup B) = \phi$$

$(A \cup B) = \phi$ implies the termination of computation by definition. We can verify that the algorithm never detects a false termination as follows:

$$(A \cup B) = \phi \Rightarrow W_c + \sum_{W \in C} W = 1 \text{ by } P_1$$

Since the message transmission delay is finite, eventually $W_c = 1$. Therefore, the algorithm detects every true termination in finite time.

The above algorithm can be extended to dynamic systems where an active process may spawn off another active process or may migrate from one site to another. When one process creates another, its weight can be split as in Rule 1 and the two resulting weights can be assigned to the creating and the created processes.

An efficient scheme to implement the system of weights is presented in [8]. This scheme can generate a large number of weights while maintaining the invariants. The details of the implementation of the scheme of weights are beyond the scope of this book and the reader is referred to [8].