Keeping the clocks in a distributed system synchronized to within 5 or 10 msec is an expensive and nontrivial task. Lamport [1978] observed that for most applications it is not necessary to keep the clocks in a distributed system synchronized. Rather, it is sufficient to ensure that all events that occur in a distributed system be totally ordered in a manner that is consistent with an observed behavior.

For partial ordering of events, Lamport defined a new relation called *happened-before* and introduced the concept of logical clocks for ordering of events based on the happened-before relation. He then gave a distributed algorithm extending his idea of partial ordering to a consistent total ordering of all the events in a distributed system. His idea is presented below.

6.3.1 Happened-Before Relation

The happened-before relation (denoted by $\rightarrow$) on a set of events satisfies the following conditions:

1. If a and b are events in the same process and a occurs before b, then $a \rightarrow b$.
2. If a is the event of sending a message by one process and b is the event of the receipt of the same message by another process, then $a \rightarrow b$. This condition holds by the law of causality because a receiver cannot receive a message until the sender sends it, and the time taken to propagate a message from its sender to its receiver is always positive.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. That is, happened-before is a transitive relation.

Notice that in a physically meaningful system, an event cannot happen before itself, that is, $a \rightarrow a$ is not true for any event a. This implies that happened-before is an irreflexive partial ordering on the set of all events in the system.

In terms of the happened-before relation, two events a and b are said to be *concurrent* if they are not related by the happened-before relation. That is, neither $a \rightarrow b$ nor $b \rightarrow a$ is true. This is possible if the two events occur in different processes that do not exchange messages either directly or indirectly via other processes. Notice that this definition of concurrency simply means that nothing can be said about when the two events happened or which one happened first. That is, two events are concurrent if neither can causally affect the other. Due to this reason, the happened-before relation is sometimes also known as the relation of *causal ordering*.

A space-time diagram (such as the one shown in Fig. 6.3) is often used to illustrate the concepts of the happened-before relation and concurrent events. In this diagram, each

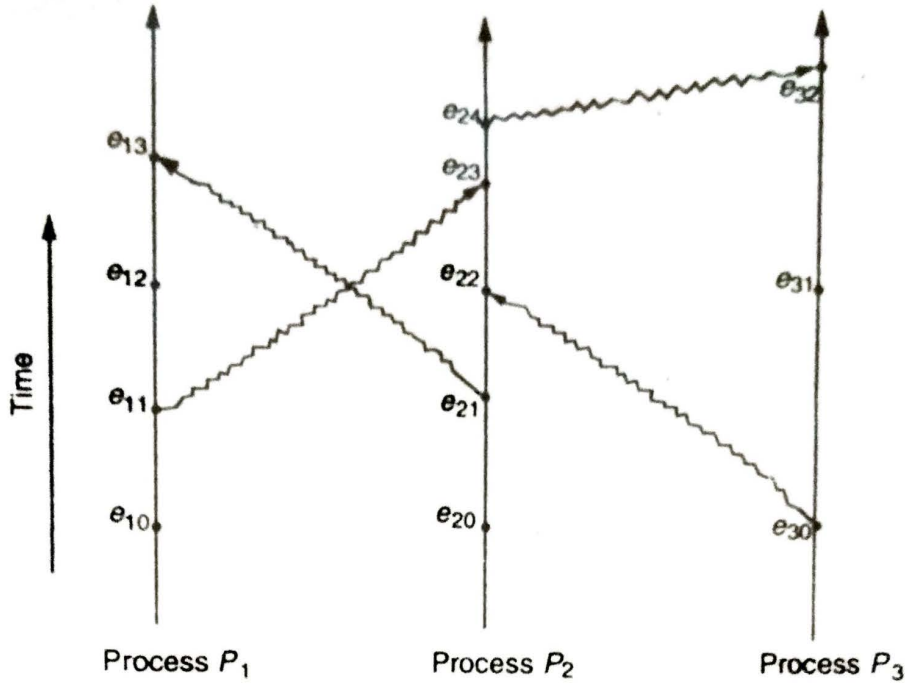

Fig. 6.3 Space-time diagram for three processes.

vertical line denotes a process, each dot on a vertical line denotes an event in the corresponding process, and each wavy line denotes a message transfer from one process to another in the direction of the arrow.

From this space-time diagram it is easy to see that for two events a and b, $a \rightarrow b$ is true if and only if there exists a path from a to b by moving forward in time along process and message lines in the direction of the arrows. For example, some of the events of Figure 6.3 that are related by the happened-before relation are

$$e_{10} \rightarrow e_{11} \qquad e_{20} \rightarrow e_{24} \qquad e_{11} \rightarrow e_{23} \qquad e_{21} \rightarrow e_{13}$$
$$e_{30} \rightarrow e_{24} \quad (\text{since } e_{30} \rightarrow e_{22} \text{ and } e_{22} \rightarrow e_{24})$$
$$e_{11} \rightarrow e_{32} \quad (\text{since } e_{11} \rightarrow e_{23},\ e_{23} \rightarrow e_{24},\ \text{and } e_{24} \rightarrow e_{32})$$

On the other hand, two events a and b are concurrent if and only if no path exists either from a to b or from b to a. For example, some of the concurrent events of Figure 6.3 are

$$e_{12} \text{ and } e_{20}, \quad e_{21} \text{ and } e_{30}, \quad e_{10} \text{ and } e_{30}, \quad e_{11} \text{ and } e_{31}, \quad e_{12} \text{ and } e_{32}, \quad e_{13} \text{ and } e_{22}$$

6.3.2 Logical Clocks Concept

To determine that an event a happened before an event b, either a common clock or a set of perfectly synchronized clocks is needed. We have seen that neither of these is available in a distributed system. Therefore, in a distributed system the happened-before relation must be defined without the use of globally synchronized physical clocks.

Lamport [1978] provided a solution for this problem by introducing the concept of logical clocks.

The logical clocks concept is a way to associate a timestamp (which may be simply a number independent of any clock time) with each system event so that events that are related to each other by the happened-before relation (directly or indirectly) can be properly ordered in that sequence. Under this concept, each process P_i has a clock C_i associated with it that assigns a number $C_i(a)$ to any event a in that process. The clock of each process is called a logical clock because no assumption is made about the relation of the numbers $C_i(a)$ to physical time. In fact, the logical clocks may be implemented by counters with no actual timing mechanism. With each process having its own clock, the entire system of clocks is represented by the function C, which assigns to any event b the number $C(b)$, where $C(b) = C_j(b)$ if b is an event in process P_j.

The logical clocks of a system can be considered to be correct if the events of the system that are related to each other by the happened-before relation can be properly ordered using these clocks. Therefore, the timestamps assigned to the events by the system of logical clocks must satisfy the following *clock condition:*

For any two events a and b, if $a \rightarrow b$, then $C(a) < C(b)$.

Note that we cannot expect the converse condition to hold as well, since that would imply that any two concurrent events must occur at the same time, which is not necessarily true for all concurrent events.

6.3.3 Implementation of Logical Clocks

From the definition of the happened-before relation, it follows that the clock condition mentioned above is satisfied if the following conditions hold:

C1: If a and b are two events within the same process P_i and a occurs before b, then $C_i(a) < C_i(b)$.

C2: If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i(a) < C_j(b)$.

In addition to these conditions, which are necessary to satisfy the clock condition, the following condition is necessary for the correct functioning of the system:

C3: A clock C_i associated with a process P_i must always go forward, never backward. That is, corrections to time of a logical clock must always be made by adding a positive value to the clock, never by subtracting value.

Obviously, any algorithm used for implementing a set of logical clocks must satisfy all these three conditions. The algorithm proposed by Lamport is given below.

To meet conditions C1, C2, and C3, Lamport's algorithm uses the following implementation rules:

IR1: Each process P_i increments C_i between any two successive events.

IR2: If event a is the sending of a message m by process P_i, the message m contains a timestamp $T_m = C_i(a)$, and upon receiving the message m a process P_j sets C_j greater than or equal to its present value but greater than T_m.

Rule IR1 ensures that condition C1 is satisfied and rule IR2 ensures that condition C2 is satisfied. Both IR1 and IR2 ensure that condition C3 is also satisfied. Hence the simple implementation rules IR1 and IR2 guarantee a correct system of logical clocks.

The implementation of logical clocks can best be illustrated with an example. How a system of logical clocks can be implemented either by using counters with no actual timing mechanism or by using physical clocks is shown below.

✓Implementation of Logical Clocks by Using Counters

As shown in Figure 6.4, two processes P_1 and P_2 each have a counter C_1 and C_2, respectively. The counters act as logical clocks. At the beginning, the counters are initialized to zero and a process increments its counter by 1 whenever an event occurs in that process. If the event is sending of a message (e.g., events e_{08} and e_{14}), the process includes the incremented value of the counter in the message. On the other hand, if the event is receiving of a message (e.g., events e_{13} and e_{08}), instead of simply incrementing the counter by 1, a check is made to see if the incremented counter value is less than or equal to the timestamp in the received message. If so, the counter value is corrected and

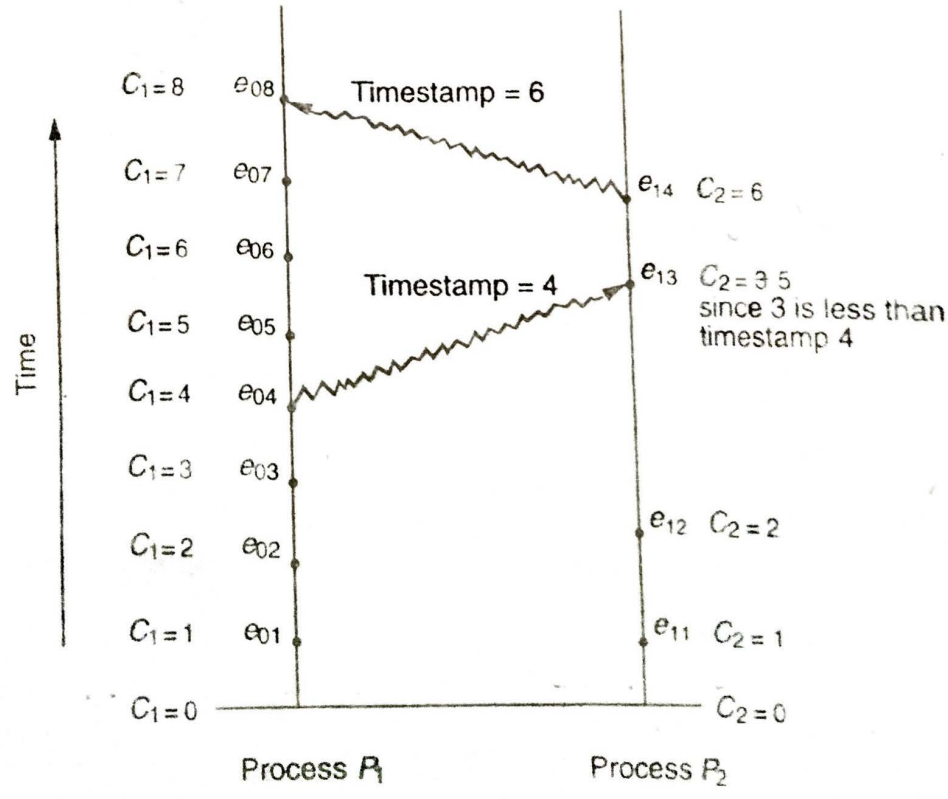

Fig. 6.4 Example illustrating the implementation of logical clocks by using counters.

set to 1 plus the timestamp in the received message (e.g., in event e_{13}). If not, the counter value is left as it is (e.g., in event e_{18}).

Implementation of Logical Clocks by Using Physical Clocks

The implementation of the example of Figure 6.4 by using physical clocks instead of counters is shown in Figure 6.5. In this case, each process has a physical clock associated with it. Each clock runs at a constant rate. However, the rates at which different clocks run are different. For instance, in the example of Figure 6.5, when the clock of process P_1 has ticked 10 times, the clock of process P_2 has ticked only 8 times.

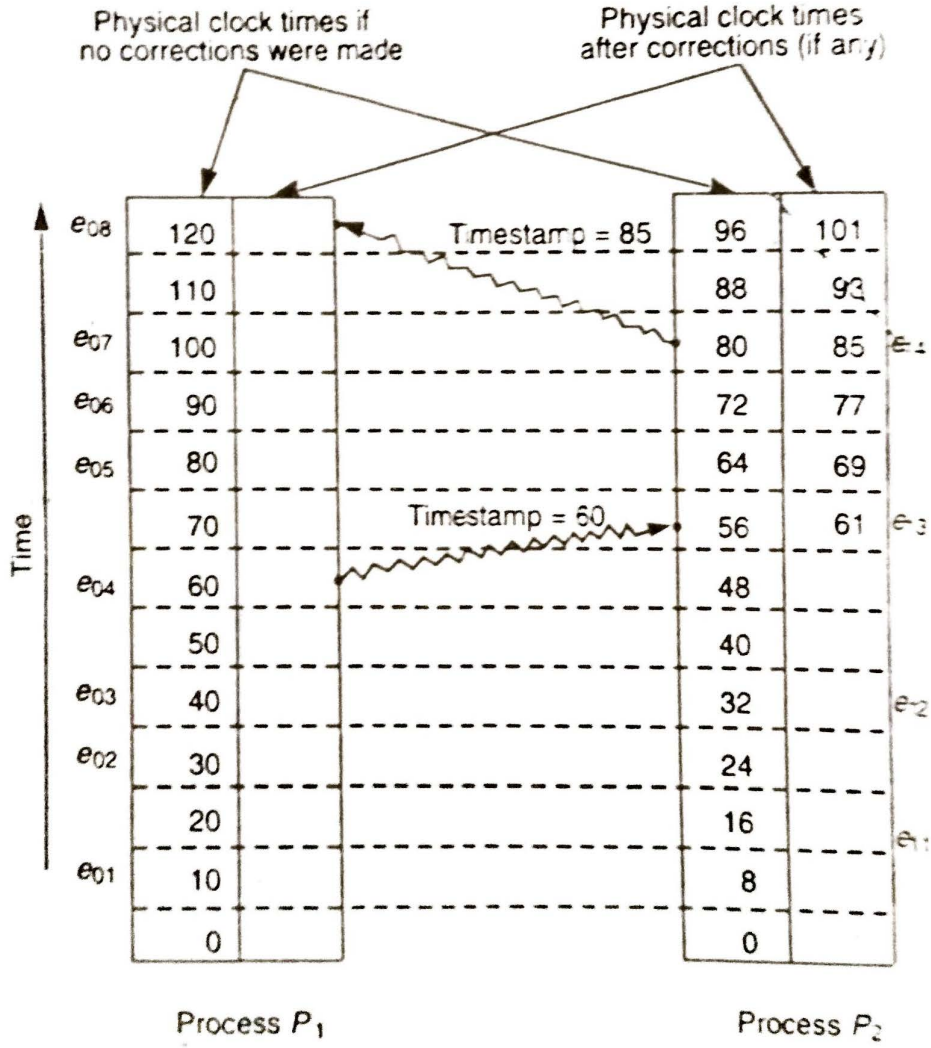

Fig. 6.5 Example illustrating the implementation of logical clocks by using physical clocks.

To satisfy condition C1, the only requirement is that the physical clock of a process must tick at least once between any two events in that process. This is usually not a problem because a computer clock is normally designed to click several times between two events that happen in quick succession. To satisfy condition C2, for a

message-sending event (e.g., events e_{04} and e_{14}), the process sending the message includes its current physical time in the message. And for a message-receiving event (e.g., events e_{13} and e_{08}), a check is made to see if the current time in the receiver's clock is less than or equal to the time included in the message. If so, the receiver's physical clock is corrected by fast forwarding its clock to be 1 more than the time included in the message (e.g., in event e_{13}). If not, the receiver's clock is left as it is (e.g., in event e_{08}).

6.3.4 Total Ordering of Events

We have seen how a system of clocks satisfying the clock condition can be used to order the events of a system based on the happened-before relationship among the events. We simply need to order the events by the times at which they occur. However, recall that the happened-before relation is only a partial ordering on the set of all events in the system. With this event-ordering scheme, it is possible that two events a and b that are not related by the happened-before relation (either directly or indirectly) may have the same timestamps associated with them. For instance, if events a and b happen respectively in processes P_1 and P_2, when the clocks of both processes show exactly the same time (say 100), both events will have a timestamp of 100. In this situation, nothing can be said about the order of the two events. Therefore, for total ordering on the set of all system events, an additional requirement is desirable: No two events ever occur at exactly the same time. To fulfill this requirement, Lamport proposed the use of any arbitrary total ordering of the processes. For example, process identity numbers may be used to break ties and to create a total ordering of events. For instance, in the situation described above, the timestamps associated with events a and b will be 100.001 and 100.002, respectively, where the process identity numbers of processes P_1 and P_2 are 001 and 002, respectively. Using this method, we now have a way to assign a unique timestamp to each event in a distributed system to provide a total ordering of all events in the system.