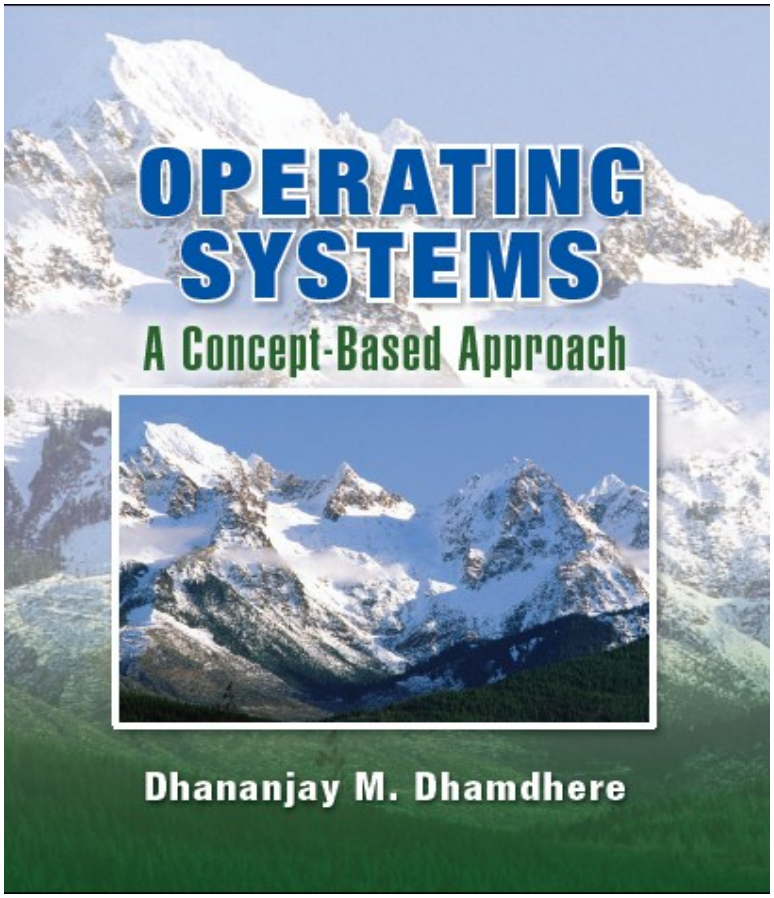

Chapter 17

Theoretical Issues in Distributed Systems

Introduction

- Notions of Time and State
- States and Events in a Distributed System
- Time, Clocks and Event Precedences
- Recording the State of a Distributed System

Notions of Time and State

- *Time* indicates when an event occurred
- *State* of an entity is the condition/mode of its being
  - Depends on its features
- *Global state* of a system comprises the states of all entities in the system at a specific instant of time
- OS uses the notions of time and state for performing scheduling of resources and the CPU:
  - Find *chronological order* in which requests occurred
  - Distributed OS uses them for *recovery*
- Problem: lack of global clock in distributed systems

States and Events in a Distributed System

- Local and Global States
- Events

Local and Global States

- Each entity in a system has its own state
  - State of a memory cell is the value contained in it
  - State of CPU is contents of PSW and GPRs
  - State of process:
    - State of memory allocated to it, CPU state (if running), state of interprocess communication
- The state of an entity is a *local state*
  - State of process P_k at time t: S_k^t
- *Global state:* collection of local states of all entities at the same instant of time
  - Global state of system at time t: $S_t=\{S_1^t, S_2^t, \ldots, S_n^t\}$

Events

- An *event* can be: sending/receiving a message (over a *channel*), or other (no messages involved)
  - Channel: an interprocess communication path
- Process state changes when an event occurs in it
- We represent an event as follows:

  (process id, old state, new state, event description, channel, message)
  - *Channel* and *message* are "–" if event does not involve sending or receiving of a message

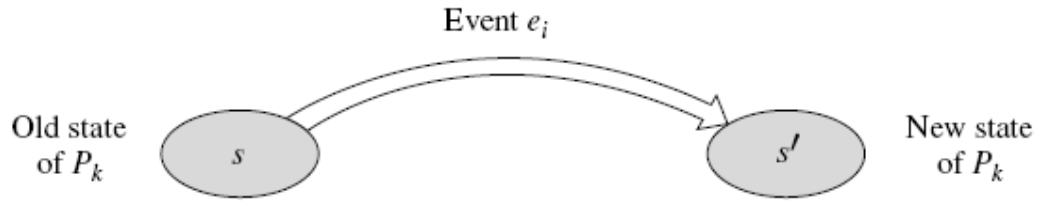

Figure 17.1 Change of state in process P_k on occurrence of event $(P_k, s, s', send, c, m)$.

Time, Clocks, and Event Precedences

- *Global clock:* abstract clock that can be accessed from different sites of a distributed system with identical results
  - Cannot be implemented in practice due to unbounded communication delays
- Alternative: use *local clocks* in processes
  - Local clocks should be reasonably well synchronized

Event Precedence

- $e_1 \rightarrow e_2$ indicates event e_1 *precedes* e_2 in time
  - i.e., e_1 occurred before e_2
- *Event ordering* implies arranging a set of events in a sequence such that each event in the sequence precedes the next one
- A *total order* (with respect to "→") exists if all events that can occur in a system can be ordered
  - A *partial order* implies that some events can be ordered but not all events can be ordered
- A *casual relationship* is used to deduce precedences
  - A "message send" event precedes "message receive"

Event Precedence (continued)

Table 17.1 Rules for Ordering of Events in a Distributed System

Category	Description of rule
Events within a process	The OS performs event handling, so it knows the order in which events occur within a process.
Events in different processes	In a *causal relationship*, i.e., a cause-and-effect relationship, an event that corresponds to the cause precedes an event in another process that corresponds to the effect.
Transitive precedence	The precedes relation is transitive; i.e., $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ implies $e_1 \rightarrow e_3$.

- e_i *precedes* e_j *:* If e_k and e_l exist such that $e_k \rightarrow e_l$, $e_i \rightarrow e_k$ or $e_i \equiv e_k$, and $e_l \rightarrow e_j$ or $e_l \equiv e_j$
- e_i *follows* e_j *:* If e_g and e_h exist such that $e_g \rightarrow e_h$, $e_j \rightarrow e_g$ or $e_j \equiv e_g$, and $e_h \rightarrow e_i$ or $e_h \equiv e_i$
- e_i *is concurrent with* e_j *:* If e_i neither precedes nor follows e_j

Example: Event Precedence

- *Timing diagram:* plot of the activities of different processes against time
  - Events in P_i are denoted as e_{i1}, e_{i2}, ..
  - e_{23} is a message send event for message m_1
  - e_{12} is a message receive event

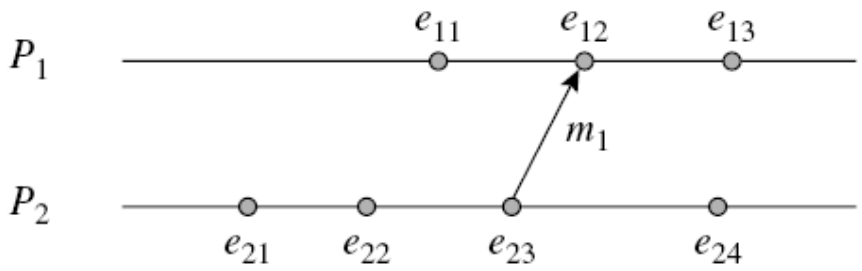

Figure 17.2 Event precedence via timing diagram.

Logical Clocks

- *Timestamping* (according to local clock) of events provides a direct method of event ordering
  - Clocks are *loosely synchronized* using causality

Algorithm 17.1 *Clock Synchronization*

1. *When a process P_k wishes to send a message m to process P_l:* P_k executes a command "send P_l, ($ts(send(m))$, m)," where $ts(send(m))$ is a timestamp obtained just prior to sending message m.
2. *When process P_l receives a message:* Process P_l performs the actions
   if $local\ clock(P_l) < ts(send(m))$ then
   $local\ clock(P_l) := ts(send(m)) + \delta$;
   timestamp the receive event.

   where $local\ clock(P_l)$ is the value in the local clock of process P_l and δ is the average communication delay in the network.

Logical Clocks (continued)

- A *logical clock (LC_k)* is incremented by 1 only when an event occurs in process P_k

R1	A process P_k increments LC_k by 1 whenever an event occurs in it.
R2	When process P_k receives a message m containing $ts(send(m))$, P_k sets its clock by the rule $LC_k = \max(LC_k, ts(send(m))+1)$.

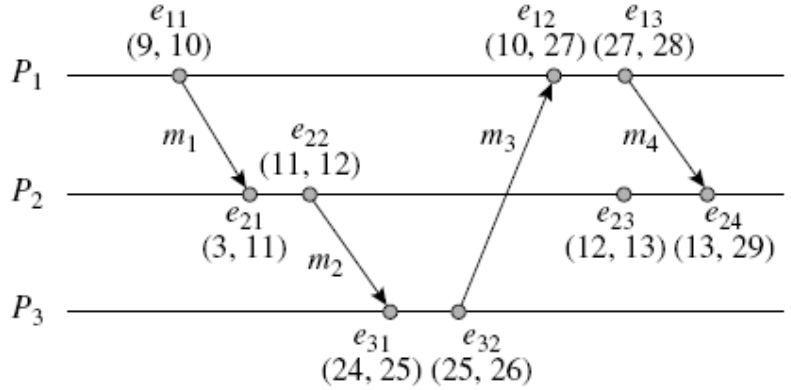

Figure 17.3 Synchronization of logical clocks.

  - $ts(e_i) < ts(e_j)$ if $e_i \rightarrow e_j$
  - Problem: $ts(e_i) < ts(e_j)$ does not imply $e_i \rightarrow e_j$

Obtaining Unique Timestamps

- Logical clock timestamps are not unique, so cannot be used to obtain a total order over events
- Problem can be overcome by using a pair $pts(e_i)$ as the timestamp of e_i, where
  - $pts(e_i) \equiv$ (*local time, process id*)
- Event ordering rules:

e_i precedes e_j iff (i) $pts(e_i).local\ time < pts(e_j).local\ time$, or
(ii) $pts(e_i).local\ time = pts(e_j).local\ time$ and $pts(e_i).process\ id < pts(e_j).process\ id$

Vector Clocks

- A *vector clock* contains *n* elements, where *n* is the number of processes in the distributed system

$VC_k[k]$	The logical clock of process P_k
$VC_k[l], l \neq k$	The highest value in the logical clock of process P_l which is known to process P_k—that is, the highest value of $VC_l[l]$ known to it

R3	A process P_k increments $VC_k[k]$ by 1 whenever an event occurs in it.
R4	When process P_k receives a message m containing $vts(send(m))$, P_k sets its clock as follows: For all l: $VC_k[l] = \max(VC_k[l], vts(send(m))[l])$.

Vector Clocks (continued)

- Precedence rules:
  - e_i *precedes* e_j*:*
    - For all l: $vts(e_i)[l] \leq vts(e_j)[l]$, but
      for some k: $vts(e_i)[k] \neq vts(e_j)[k]$
  - e_i *follows* e_j*:*
    - For all l: $vts(e_i)[l] \geq vts(e_j)[l]$, but
      for some k: $vts(e_i)[k] \neq vts(e_j)[k]$
  - e_i *,* e_j *are concurrent:*
    - For some k, l: $vts(e_i)[k] < vts(e_j)[k]$, and $vts(e_i)[l] > vts(e_j)[l]$
- $vts(e_i) < vts(e_j)[l]$ if and only if $e_i \rightarrow e_j$

Example: Synchronization of Vector Clocks

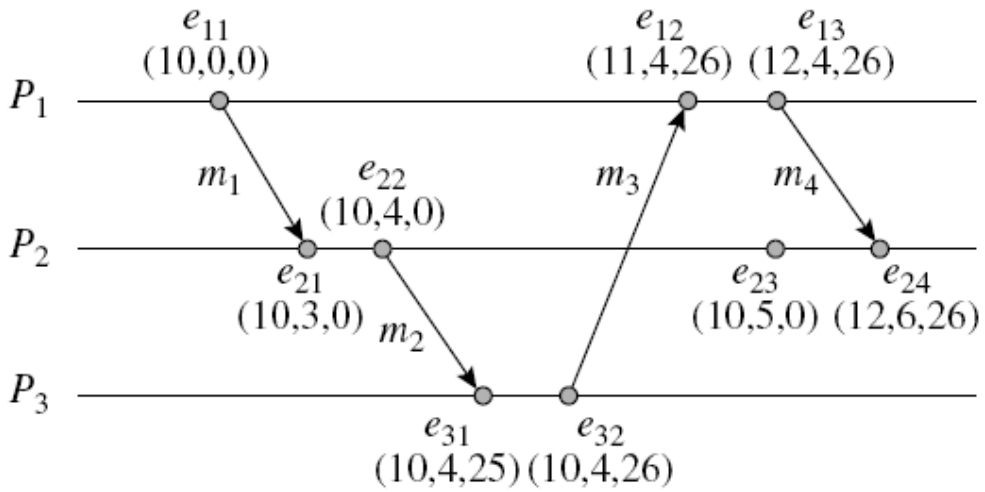

Figure 17.4 Synchronization of vector clocks.

- Total order can be obtained using a pair
  $pvts(e_i) \equiv$ (*local time*, *process id*) as timestamp of e_i

Recording the State of a Distributed System

- Problem: it is not possible to get all nodes to record their states at the same time instant
  - Local clocks are not perfectly synchronized
  - Any other collection of local states may be inconsistent
- Alternative: algorithm for obtaining a consistent collection of local states
  - Collected state not a substitute for the global state
    - However, has properties that facilitate some of the control functions in a distributed OS

Recording the State of a Distributed System (continued)

- Example: inconsistent state recording
  - Banking application: P_1 in N_1 and P_2 in N_2

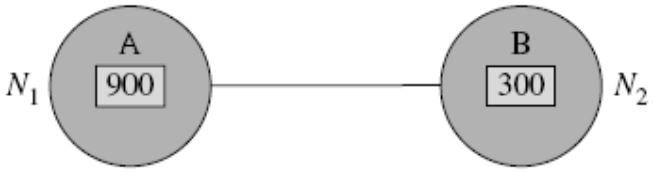

Figure 17.5 A funds transfer system.

    - Actions:
      1. P_1 debits \$100 to account A in node N_1
      2. P_1 sends message to P_2 to credit \$100 to account B
      3. P_2 credits \$100 to account B in node N_2
  - Inconsistent if A's balance is recorded before (1) and B's balance is recorded after (3)

Consistent State Recording

- *State recording:* collection of local states of entities in a system obtained through some algorithm
- *Consistent state recording:* one in which process states of every pair of processes in the system are consistent according to:

Definition 17.1 Mutually Consistent Local States Local states of processes P_k and P_l are mutually consistent if

1. Every message recorded as "received from P_l" in P_k's state is recorded as "sent to P_k" in P_l's state, and
2. Every message recorded as "received from P_k" in P_l's state is recorded as "sent to P_l" in P_k's state.

Properties of a Consistent State Recording

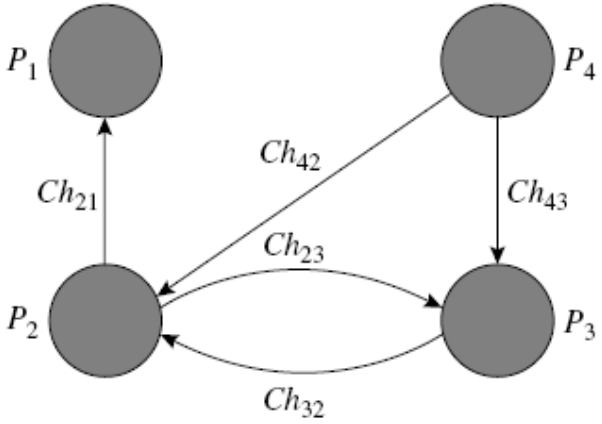

Figure 17.6 A distributed computation for state recording.

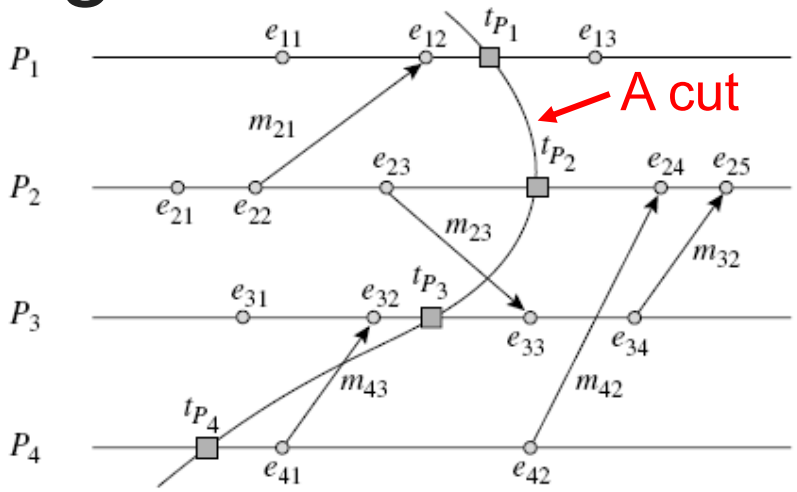

Figure 17.7 A timing diagram for the distributed computation of Figure 17.6.

Process states are recorded at t_{P1}.. t_{P4}

Table 17.2 Local States of Processes

Process	Description of recorded state
P_1	No messages have been sent. Message m_{21} has been received.
P_2	Messages m_{21} and m_{23} have been sent. No messages have been received.
P_3	No messages have been sent. Message m_{43} has been received.
P_4	No messages have been sent. No messages have been received.

Properties of a Consistent State Recording (continued)

Definition 17.2 Cut of a System A curve that connects the points in a timing diagram at which states of processes are recorded, in increasing order by process number.

- "A cut is taken" means that a collection of local states is recorded
- A cut represents a consistent state recording of a system if the states of each pair of processes satisfy Definition 17.1
- State of a channel is the set of messages it contains
- A cut may intersect with a message:
  - *Forward or backward intersection*
  - Backward intersection makes a cut inconsistent

Properties of a Consistent State Recording (continued)

- Consistency condition for a cut:
  - CC: Cut *C* represents a consistent state recording of a distributed system if future of cut is closed under the precedes relation on events (closed under "→")

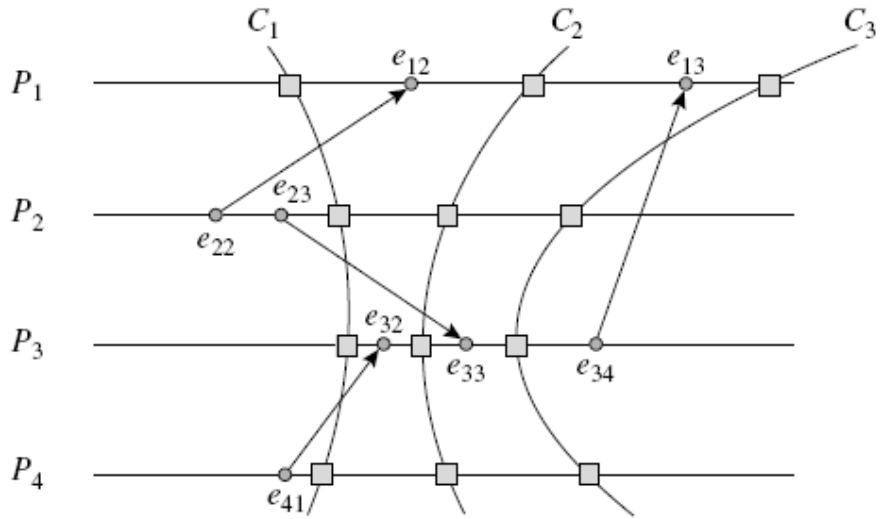

Figure 17.8 Consistency of cuts—cuts C_1, C_2 are consistent while C_3 is inconsistent.

Chandy–Lamport Algorithm for state recording

- Assumptions of the algorithm
  - Channels are FIFO
  - Channels are unidirectional
  - Channels have unbounded message buffering capacities
- The state of a process indicates the messages sent and received by it
- A special message called a *marker* is sent to ask a process to record its state
  - Process receives markers over all channels incident on it
    - Records state of channel over which it received a marker
    - If it is the first marker received, it also records its own state

An Algorithm for Consistent State Recording

- Notation:

  $Received_{ij}$: Messages received by P_j over channel Ch_{ij}

  $Recorded\_recd_{ij}$: Messages recorded as received in the state of P_j

Algorithm 17.2 *Chandy–Lamport Algorithm*

1. *When a process P_i initiates the state recording:* P_i records its own state and sends a marker on each outgoing channel connected to it.
2. *When process P_j receives a marker over an incoming channel Ch_{ij}:* Process P_j performs the following actions:
   - **a.** If P_j had not received any marker earlier, then
     - **i.** Record its own state.
     - **ii.** Record the state of channel Ch_{ij} as *empty*.
     - **iii.** Send a marker on each outgoing channel connected to it.
   - **b.** Otherwise, record the state of channel Ch_{ij} as the set of messages $Received_{ij} - Recorded\_recd_{ij}$.

Example: Operation of the Chandy–Lamport Algorithm

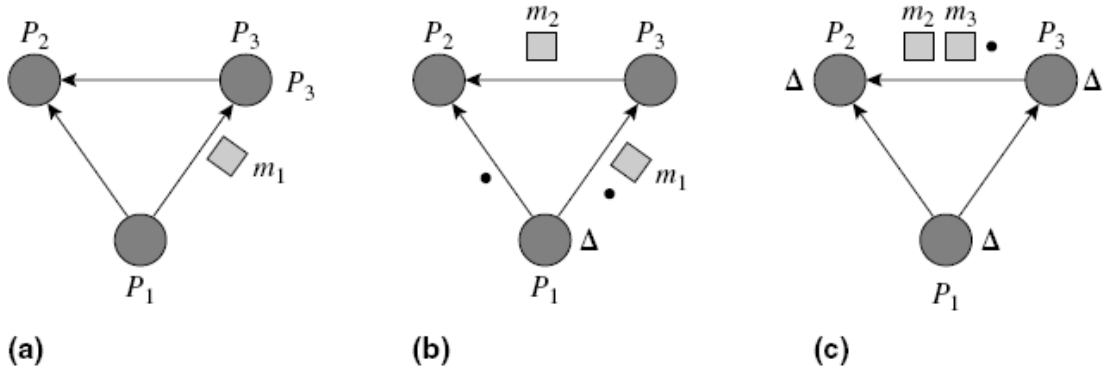

Figure 17.9 Example of the Chandy–Lamport algorithm: system at times $0, 2^+$, and 5^+.

Table 17.3 Recorded States of Processes and Channels in Figure 17.9

Entity	Description of recorded state
P_1	Message m_1 has been sent. No messages have been received.
P_2	No messages have been sent or received.
P_3	Messages m_2 and m_3 have been sent. Message m_1 has been received.
Ch_{12}	Empty
Ch_{13}	Empty
Ch_{23}	Contains the messages m_2 and m_3

Example: Recorded State versus Global State

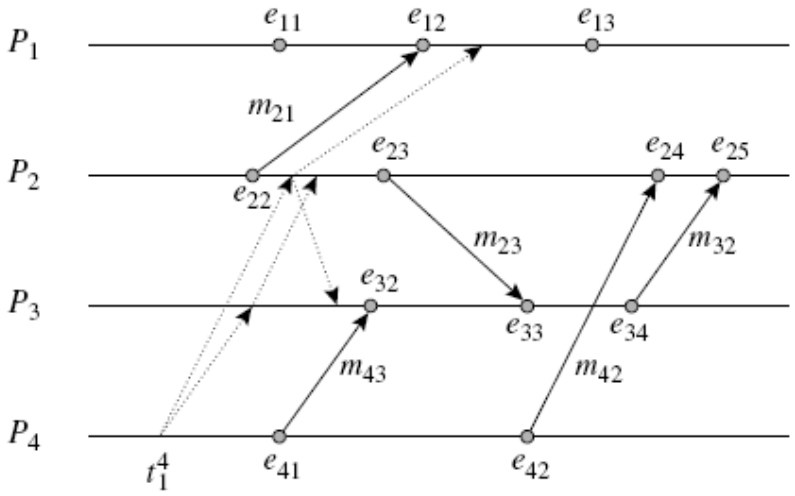

Figure 17.10 State recording of the system of Figures 17.6 and 17.7.

Table 17.4 A Recorded State that Does Not Match Any Global State

Entity*	Description of recorded state
P_1	No messages have been sent. Message m_{21} has been received.
P_2	Message m_{21} has been sent. No messages have been received.
P_3	No messages have been sent or received.
P_4	No messages have been sent or received.

* States of all channels are recorded as empty.

Summary

- Operating systems use notions of time and state
  - Local state: State of an entity
  - Global state: States of all entities at the same time instant
- *Precedence* of events may be deduced using the *causal relationship,* i.e., cause-and-effect relationship, and *transitivity* property of precedence
- Some events may be *concurrent*
- It is laborious to deduce the precedence of events by using transitivity, hence *timestamps* are used instead
  - Use local clocks in processes

Summary

- Using local clocks in processes
  - Logical clocks
  - Vector clocks
- Include process ids in timestamps for total ordering
- It is not possible to record the global state of a system
- Chandy-Lamport algorithm obtains *consistent recording of process states* using special messages called *markers*