Logical Ordering

Course: Distributed Computing

Faculty: Dr. Rajendra Prasath

About this topic

This course covers essential aspects of **Logical Clocks in Distributed Systems and its related concepts**

RECAP

What did you learn so far?

- Goals / Challenges in Message Passing systems
- Distributed Sorting
- Space-Time diagram
- Partial Ordering / Total Ordering
- Causal Precedence Relation
  - Happens Before
- Concurrent Events
  - How to define Concurrent Events
  - Logical vs Physical Concurrency
- Causal Ordering
- Local State vs. Global State

Causal Ordering

A Model of Distributed Executions

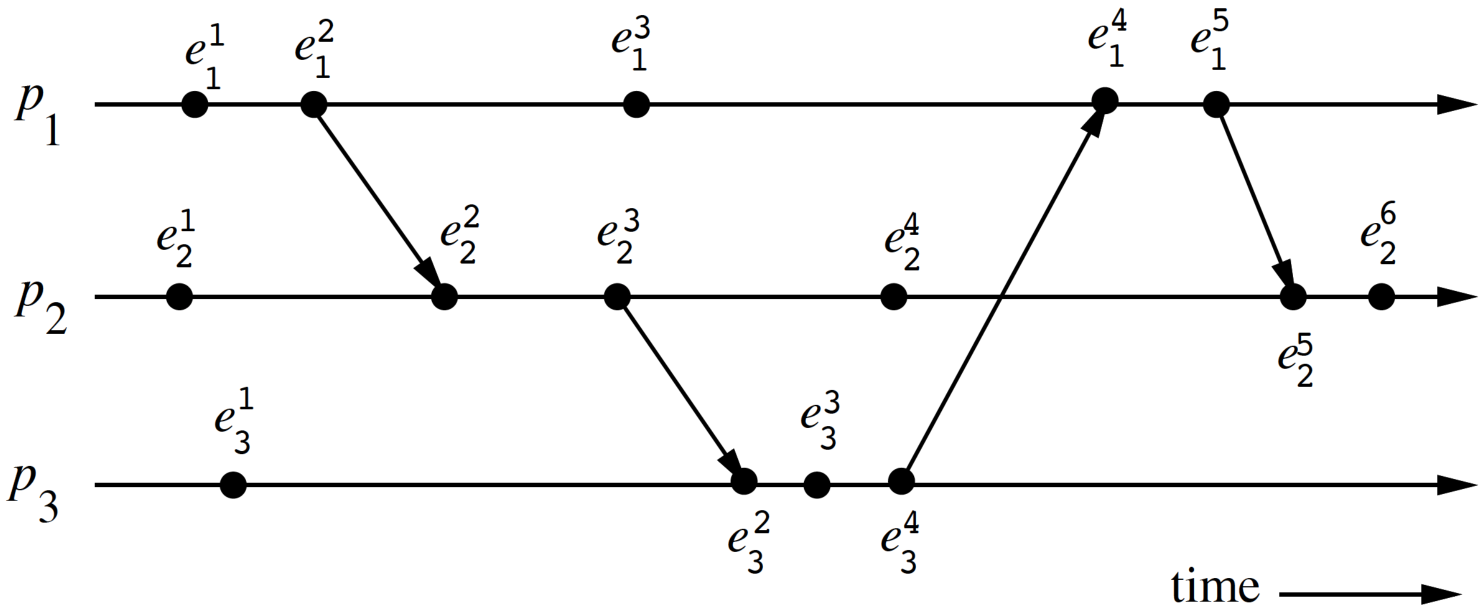
- The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process
- Define a relation $\rightarrow_{msg}$ that captures the causal dependency due to message exchanges as follows:

  For every message m that is exchanged between two processes, we have

  $$send(m) \rightarrow_{msg} receive(m)$$

- Relation $\rightarrow_{msg}$ defines causal dependencies between the pairs of corresponding send and receive events

A State-Time diagram – An Example



- **For Process p_1:**

  Second event is a message send event

  First and Third events are internal events

  Fourth event is a message receive event

Causal Ordering

- The "causal ordering" model is based on Lamport's "happens before" relation
- A system that supports the causal ordering model satisfies the following property:

CO: For any two messages m_{ij} and m_{kj},

if $send(m_{ij}) \longrightarrow send(m_{kj})$,

then $receive(m_{ij}) \longrightarrow receive(m_{kj})$

- This property ensures that causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation.
- Causally ordered delivery of messages implies FIFO message delivery. (Note that CO $\subset$ FIFO $\subset$ Non-FIFO.)
- Causal ordering model considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.

A Consistent Global State

Definition:

➔ A global state is a consistent global state iff

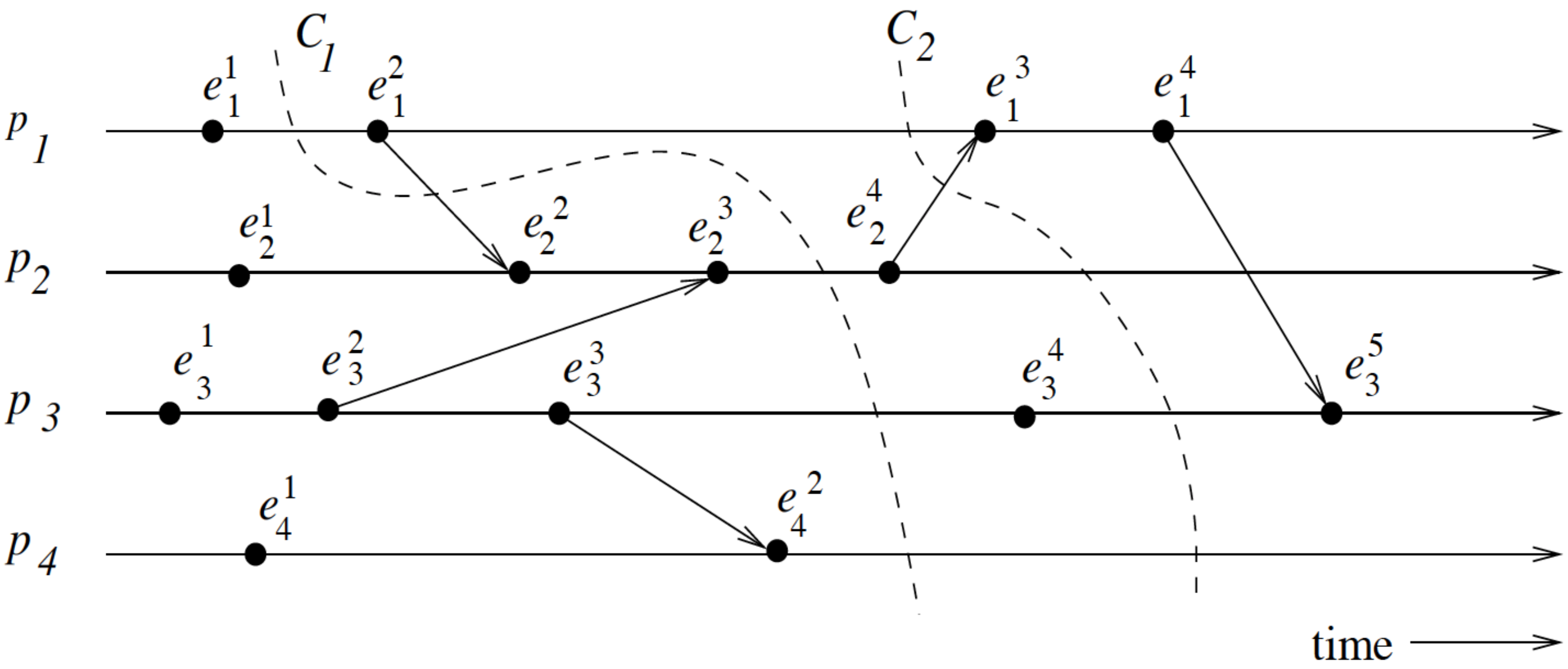
$$\forall m_{ij} : \ send(m_{ij}) \not\preceq LS_i^{x_i} \ \Leftrightarrow \ m_{ij} \notin SC_{ij}^{x_i, y_j} \bigwedge rec(m_{ij}) \not\preceq LS_j^{y_j}$$

Where the global state is given by

$$GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j, z_k}\}$$

➔ This implies that the channel state and process state must not include any message that process p_i sent after executing event

Cuts of a Distributed Computation



Physical vs Logical clocks?

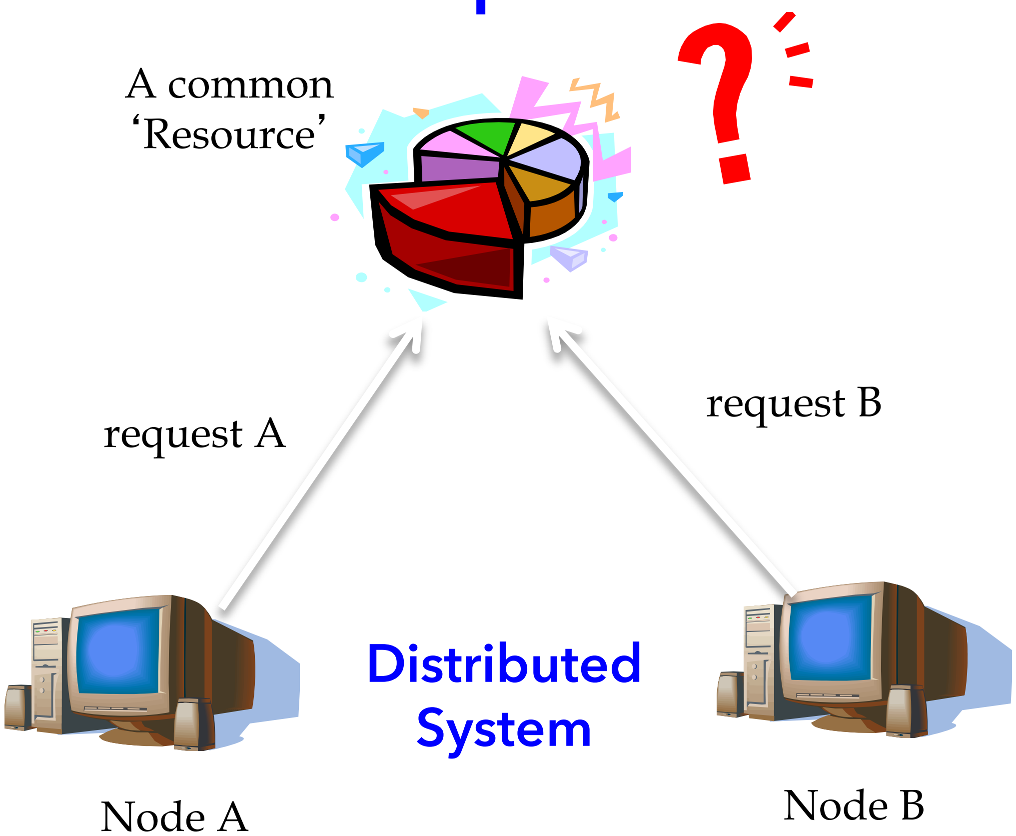
- Logical Clocks
  - Design and Implementation
- Three Different Ways
  - Scalar Time
  - Vector Time
  - Matrix Time
- Virtual Clocks
  - Time Wrap Mechanism
- Clock Synchronization
  - NTP Synchoronization Protocol

Logical Clocks

- Logical Clocks (Lamport 1978)
  - Based on "Happens Before" concept
- Knowing the ordering of events is important (?!)
- not enough with physical time
- Two simple points [Lamport 1978]
  - the order of two events in the same process
  - the event of sending message always happens before the event of receiving the message

Events Ordering - An Example

➔ **Which request was made first?**

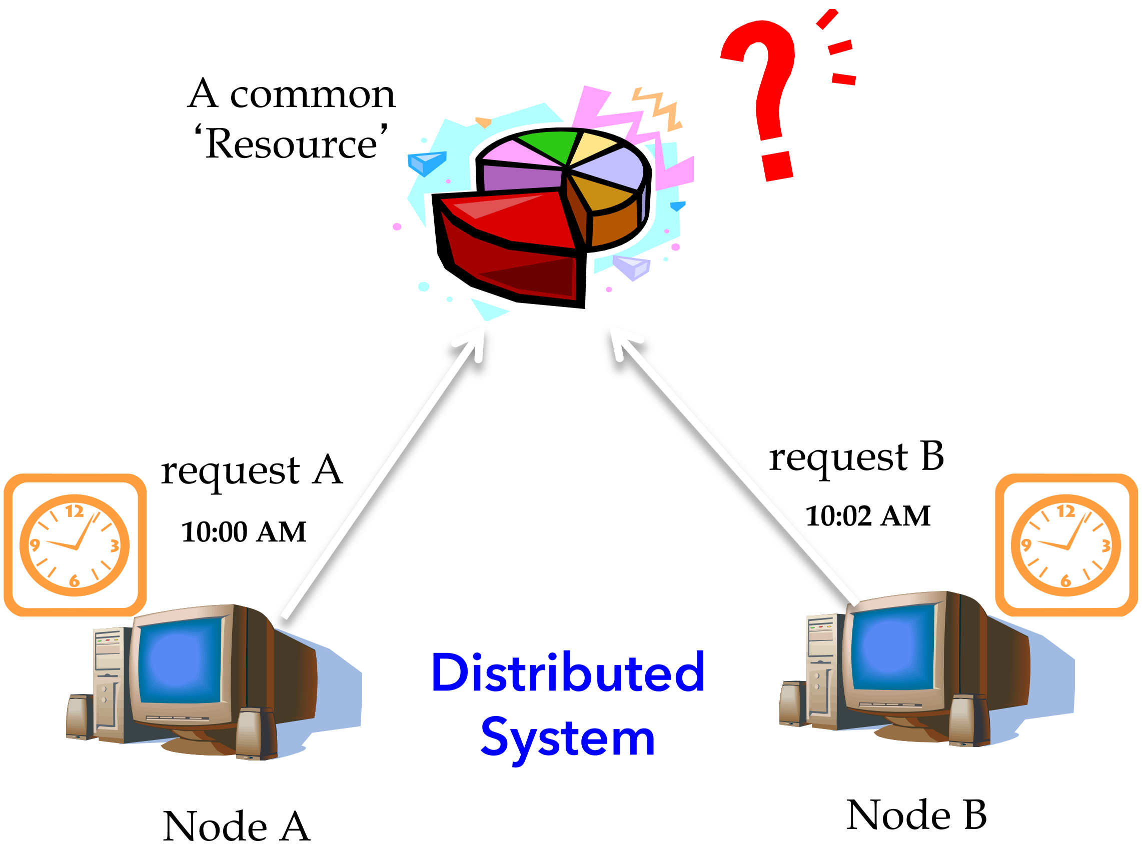


Solution

A Global Clock ??
Global Synchronization?

Events Ordering - An Example (contd)

➔ **Which request was made first?**



Solution

Individual Clocks?

Are individual clocks accurate, precise?

One clock might run faster/slower?

Logical Clocks (Lamport 1978)

Synchronization in a Distributed System:

- Event Ordering:
  - Which event occurred first?
- How to sync the clocks across the nodes?

- Can we define the notion of happened-before without using physical clocks?

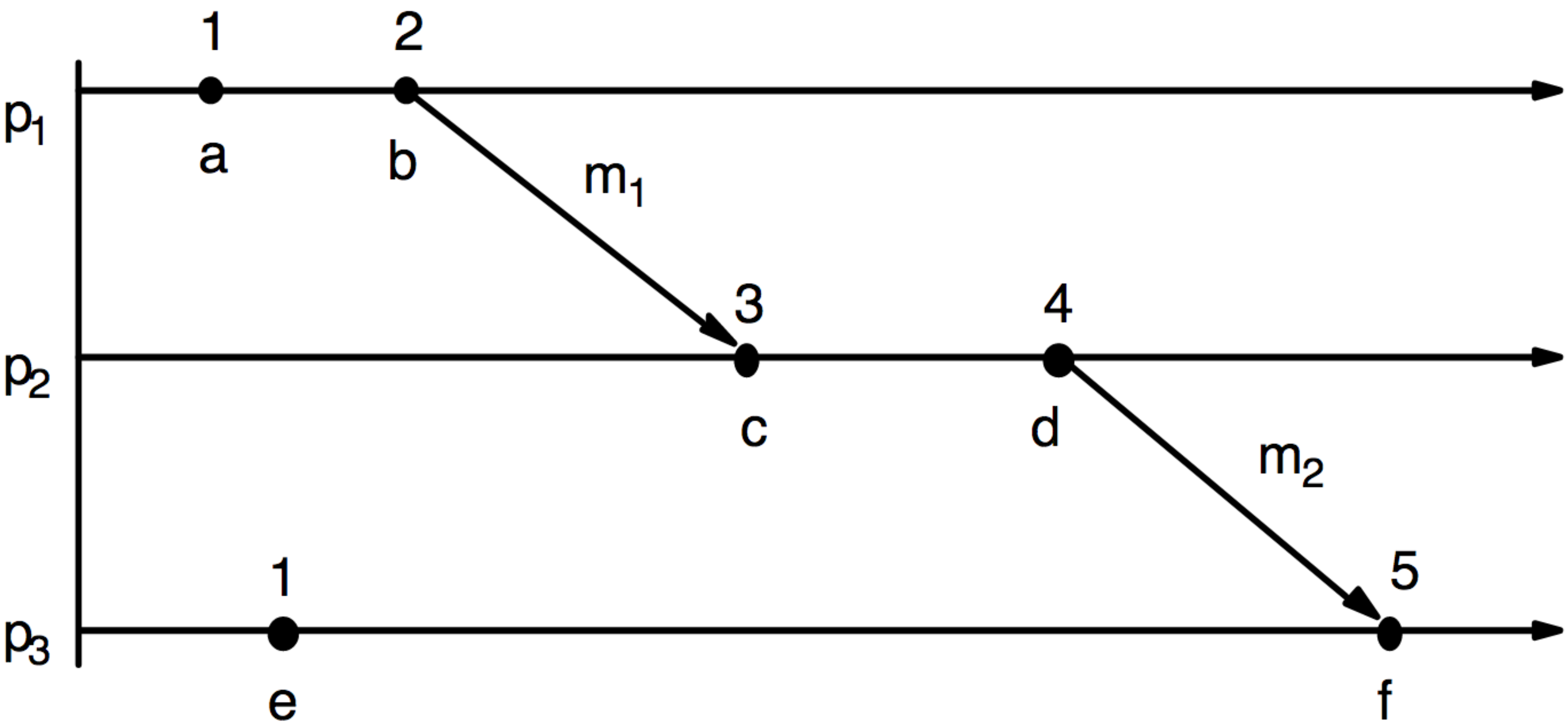
Lamport's Logical clocks

- A monotonically increasing software counter
- It does (need) not relate to a physical clock
- Each process p_i has a logical clock L_i
- LC_1: L_i is incremented by 1 before each event at process p_i
- LC_2:

  A) when process p_i sends message m,
  it piggybacks $t = L_i$

  B) when p_j receives (m,t), it sets $L_j = max(L_j, t)$ and applies LC_1 before timestamping the event $receive(m)$

A Close Look

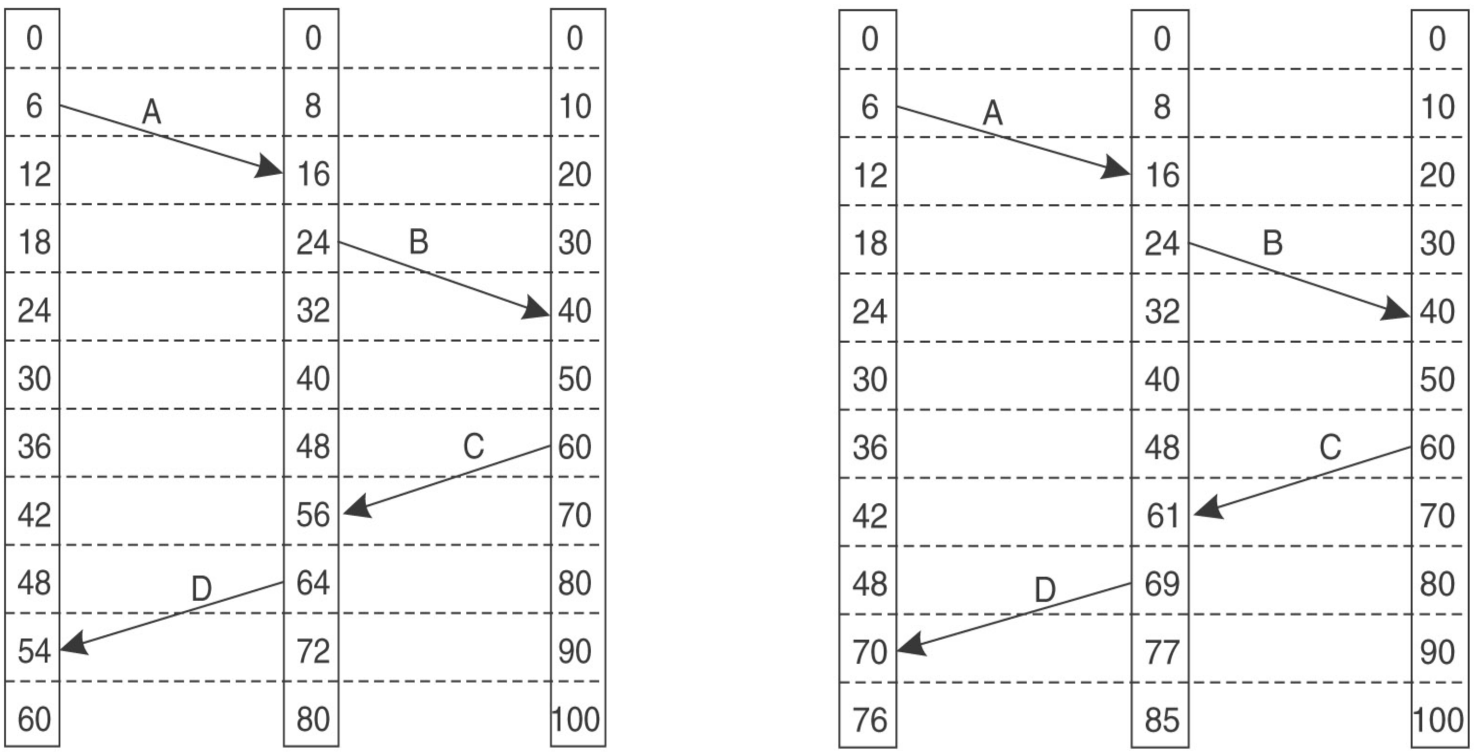


- ➔ $e \rightarrow e' \Rightarrow L(e) < L(e')$ but not vice versa
- ➔ Example: event b and event e

How to implement Lamport's clocks?

- When a message is transmitted from P1 to P2, P1 will encode the send time into the message.
- When P2 receives the message, it will record the time of receipt
- If P2 discovers that the time of receipt is before the send time, P2 will update its software clock to be one greater than the send time (1 milli second at least)
- If the time at P2 is already greater than the send time, then no action is required for P2
- With these actions the "happens-before" relationship of the message being sent and received is preserved

Correction of Clocks



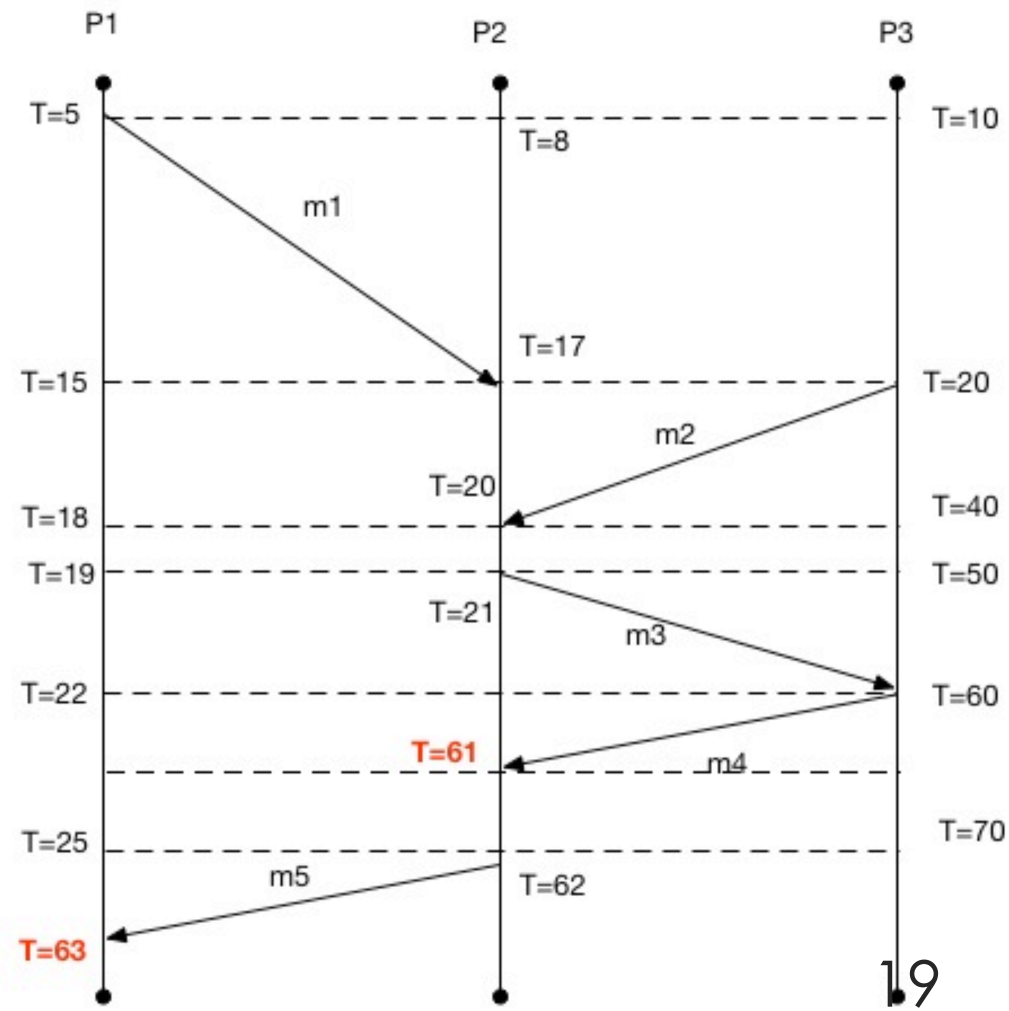
(a) (b)

An Illustration

m1 –> m3
=> C(m1) < C(m3)

m2 –> m3
=> C(m2) < C(m3)

Here which event, either m1 or m2, caused m3 to be sent?



Limitations

- Lamport's logical clocks lead to a situation where all events in a distributed system are totally ordered

  If a → b, then we can say C(a) < C(b)

- Unfortunately, with Lamport's clocks, nothing can be said about the actual time of a and b

  If the logical clock says a → b, that does not mean in reality that a actually happened before b in terms of real time

Issues with Lamport Clocks

- The problem with Lamport clocks is that they do not capture causality
- If we know that a → c and b → c we cannot say which action initiated c
- This kind of information can be important when trying to reply events in a distributed system (such as when trying to recover after a crash)
- If one node goes down, if we know the causal relationships between messages, then we can replay those messages and respect the causal relationship to get that node back up to the state it needs to be in

Vector Clocks

- Vector clocks allow causality to be captured
- Rules of Vector Clocks:
  - A vector clock $VC(a)$ is assigned to an event a
  - If $VC(a) < VC(b)$ for events a and b, then event a is known to causally preceed b
- Each Process P_i maintains a vector VC_i with the following properties:
  - $VC_i[i]$ is the number of events that have occurred so far at P_i that is, $VC_i[i]$ is the local logical clock at process P_i
  - If $VC_i[j]=k$ then P_i knows that k events have occurred at P_j. It is thus P_i's knowledge of the local time at P_j

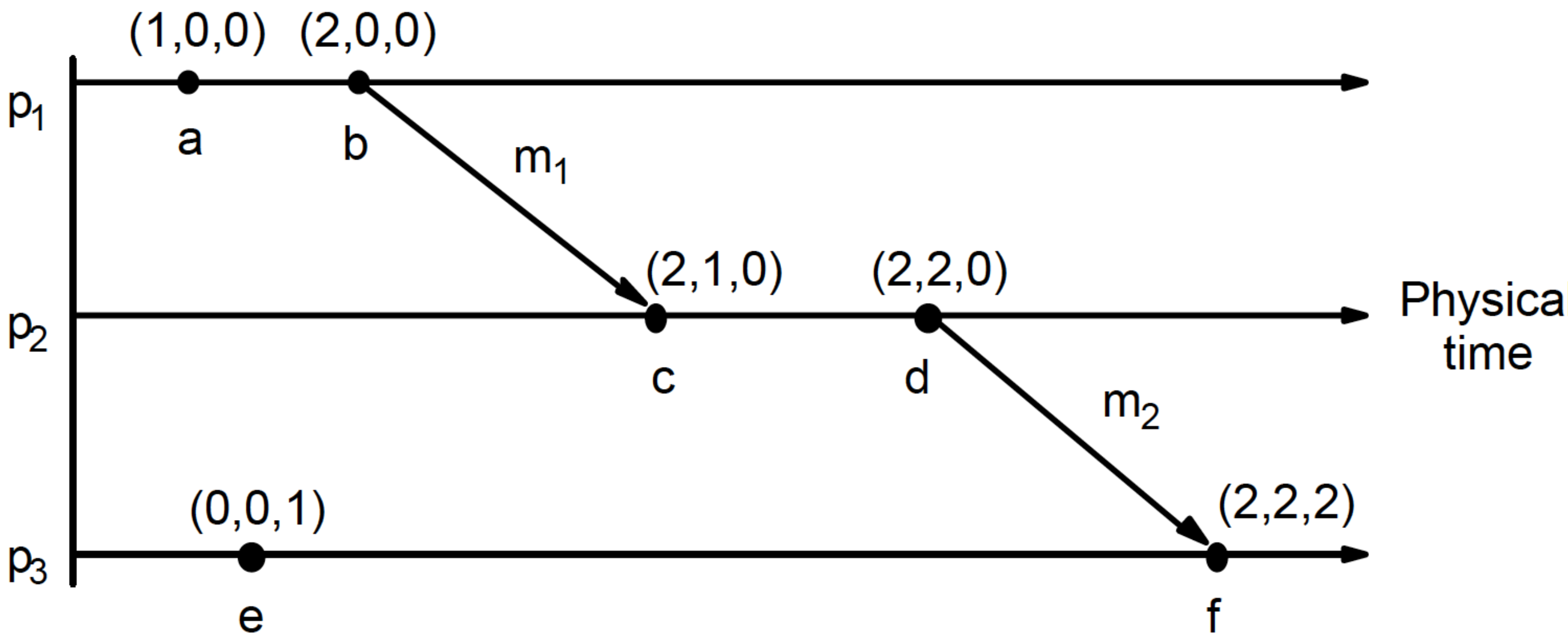
Implementing Vector Clocks

- Increment $VC_i[i]$ at each new event at P_i
- Updating Clocks:
  - Before executing any event (sending a message or an internal event):

    P_i executes $VC_i[i] \leftarrow VC_i[i]+1$
  - When process P_i sends a message m to P_j, it sets m's (vector) timestamp $ts(m)=Vc_i$
  - Upon receiving a message m, process P_j adjusts its own vector by setting $VC_j[k] \leftarrow max(VC_j[k],ts(m)[k])$ for each k

23

An Example



24

Understanding Vector Clocks

Meaning of =, <=, < for vector timestamps

(1) VC = VC′ iff VC[j] = VC′[j] for j = 1, 2, …, N

(2) VC ≤ VC′ iff VC[j] ≤ VC′[j] for j = 1, 2, …, N

(3) VC < VC′ iff VC ≤ VC′ and VC ≠ VC′

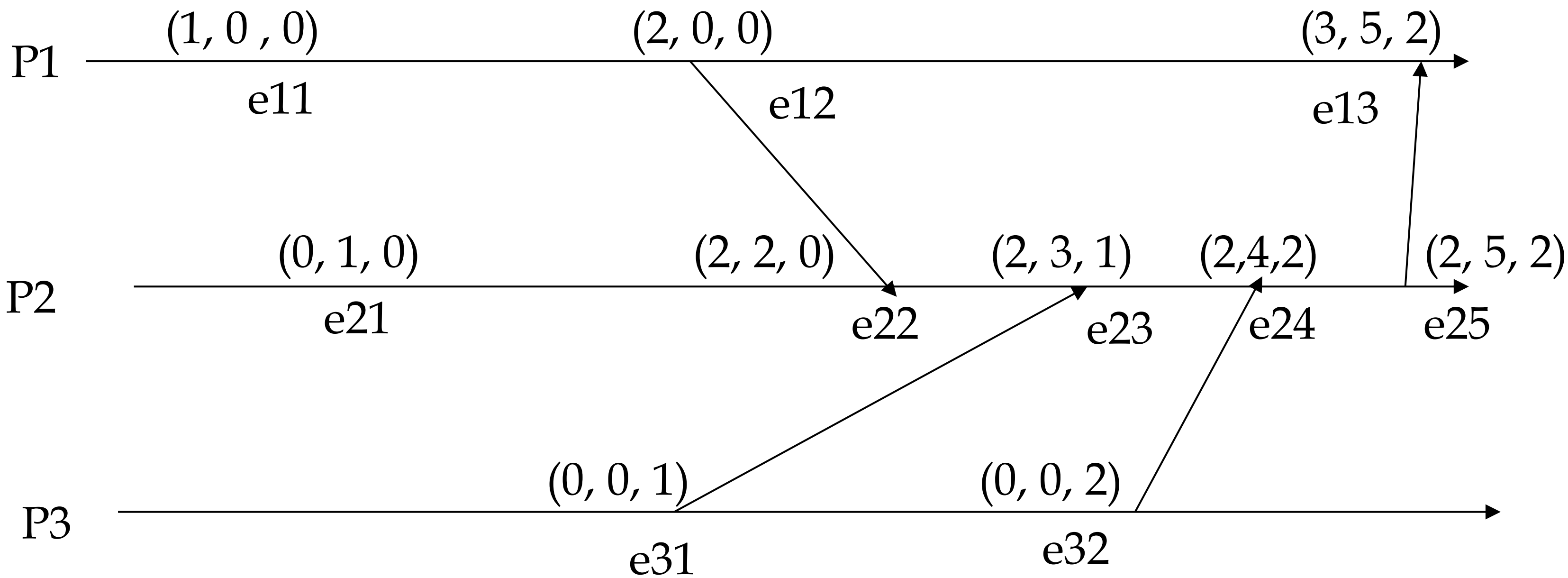
Examples:

$$(1, 3, 2) < (1, 3, 3)$$

$$(1, 3, 2) \,||\, (2, 3, 1)$$

➔ Note: e → e′ implies VC(e) < VC(e′) (The converse is also true)

An illustrative example



Less than or equal:

- $ts(a) \leq ts(b)$ **if** $ts(a)[i] \leq ts(b)[i]$ **for all** i

  (2,4,2) ≤ (3,5,2)

- $ts(e11) = (1, 0, 0)$ **and** $ts(e22) = (2, 2, 0)$

  This implies $e11 \rightarrow e22$

26

Summary

- A model of Distributed Computations
  - Causal Precedence Relations
  - Global State and Cuts of a DS
  - PAST and FUTURE events
  - What about the ordering of events?
    - How do we efficiently handle the ordering of events (discrete events)?
    - Lamport's Logical Clocks ?
    - Vector Clocks
    - Many more to come up … stay tuned in !!

How to reach me?

- ➔ Please leave me an email:
  rajendra [DOT] prasath [AT] iiits [DOT] in

- ➔ Visit my homepage @

  - ➔ http://www.iiits.ac.in/FacPages/index-rajendra.html

  OR

  - ➔ http://rajendra.2power3.com

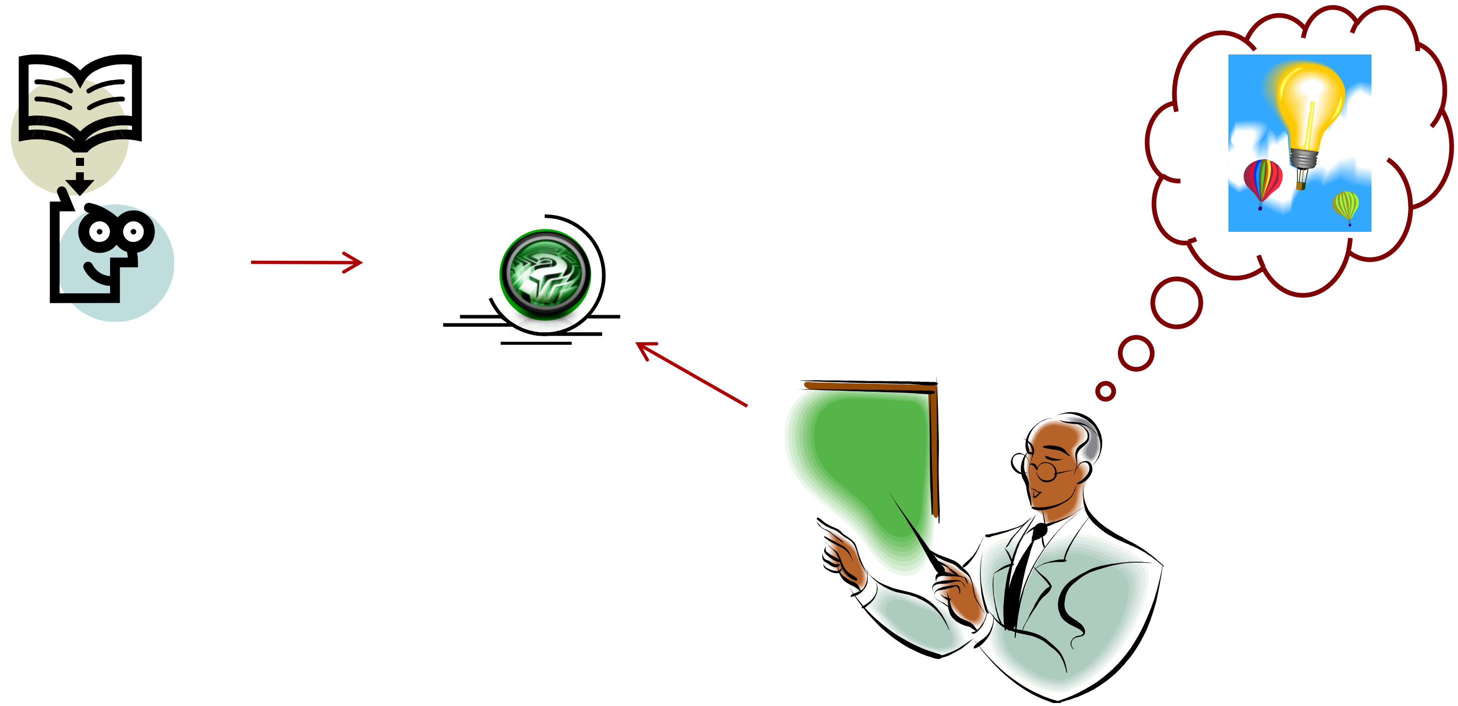
Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

Thanks ...



... **Questions ???**