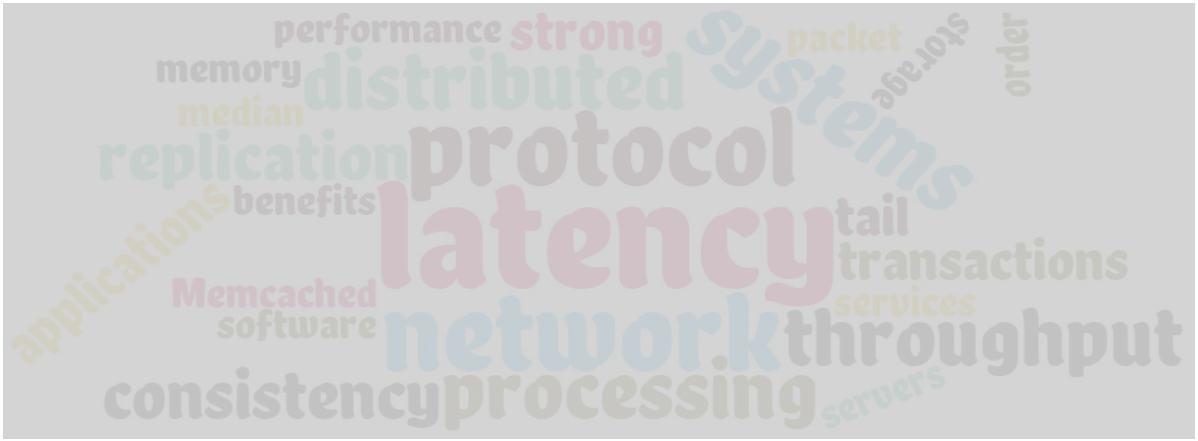




# Logical Ordering

## Course: Distributed Computing

## Faculty: Dr. Rajendra Prasath

Spring 2019

# 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

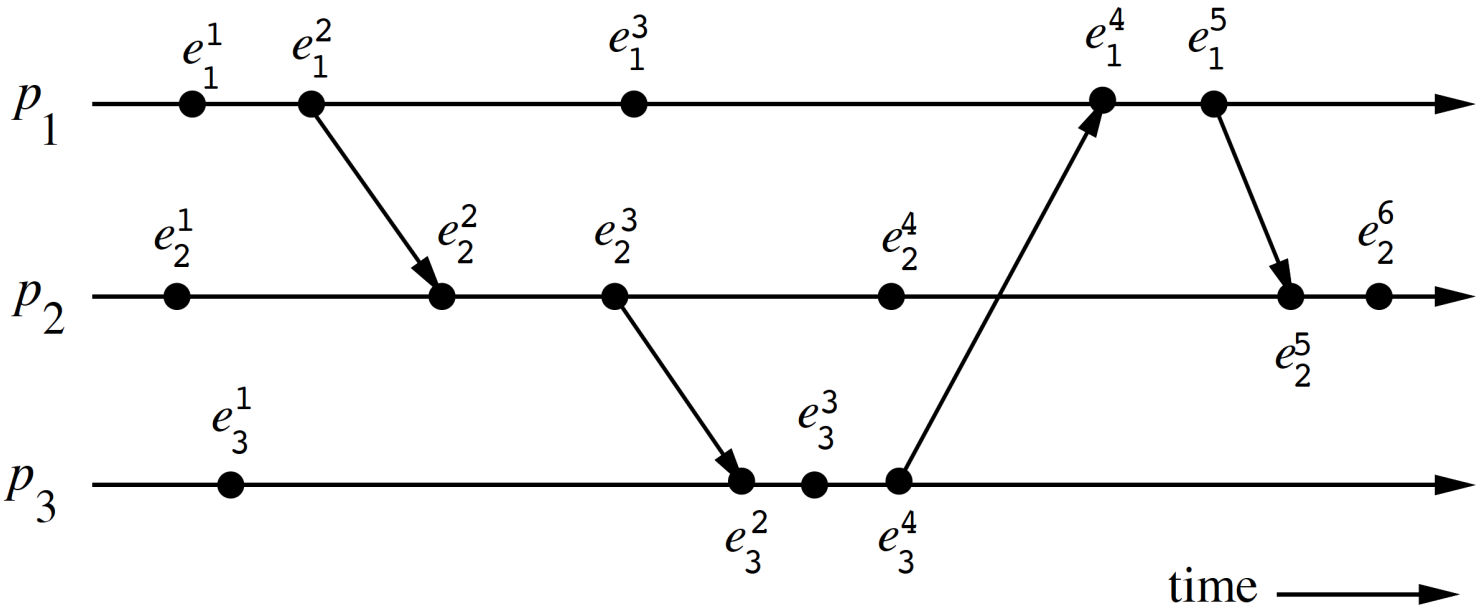
- The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process
- Define a relation $\rightarrow_{msg}$ that captures the causal dependency due to message exchanges as follows:

  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 Precedence Relation (contd)

- The relation $\rightarrow$ is as defined by Lamport

"happens before"

An event $e_1$ happens before the event $e_2$ and denoted by $e_1 \rightarrow e_2$ if the following holds true:

- $e_1$ occurs before $e_2$ on the same process OR
- $e_1$ is the send message and $e_2$ is the corresponding receive message OR
- There exists another event $e'$ such that $e_1$ happens before $e'$ and $e'$ happens before $e_2$

# Causal Precedence Relation (contd)

- For any two events $e_i$ and $e_j$, $e_i \not\rightarrow e_j$ denotes the fact that event $e_j$ does not directly or transitively dependent on event $e_i$

  That is, event $e_i$ does not causally affect event $e_j$
- In this case, event $e_j$ is not aware of the execution of $e_i$ or any event executed after $e_i$ on the same process.

Note the following two rules:

- For any two events $e_i$ and $e_j$

$$e_i \not\rightarrow e_j \text{ does not imply } e_j \not\rightarrow e_i$$

- For any two events $e_i$ and $e_j$

$$e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow ei\ .$$

# Concurrent Events

- For any two events $e_i$ and $e_j$:

  if $e_i \not\rightarrow e_j$ and $e_j \not\rightarrow e_i$,

  then events $e_i$ and $e_j$ are said to be concurrent (denoted as $e_i \parallel e_j$ )

Example:

$$e_3^3 \parallel e_2^4 \text{ and } e_2^4 \parallel e_1^5 \text{ but } e_3^3 \text{ not } \parallel e_1^5$$

- The relation $\parallel$ is not transitive;

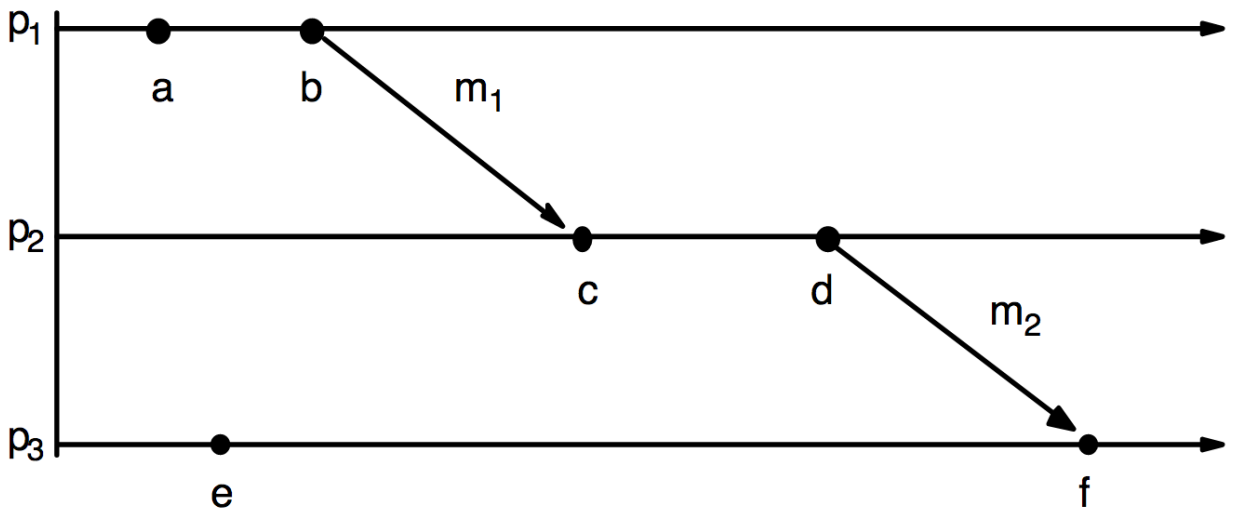
  that is,

  $(e_i \parallel e_j) \wedge (e_j \parallel e_k)$ does not imply $e_i \parallel e_k$

- For any two events $e_i$ and $e_j$ in a distributed execution,

  $e_i \rightarrow e_j$ OR $e_j \rightarrow e_i$, OR $e_i \parallel e_j$

# Concurrency – An Example




$a \rightarrow b$ (at $p1$) $c \rightarrow d$ (at $p2$)

$b \rightarrow c$ ($m1$)

also $d \rightarrow f$ ($m2$)

Not all events are related by $\rightarrow$, e.g., a $\not\rightarrow$ e and e $\not\rightarrow$ a
they are said to be concurrent; write as $a \parallel e$

# Logical vs. Physical concurrency

- Two events are logically concurrent if and only if they do not causally affect each other.
- In physical concurrency: events occur at the same instant in physical time.
- Two+ events may be logically concurrent even though they do not occur at same instant in physical time
- If processor speed and message delays would have been different, the execution of these events could have very well coincided in physical time.
- Whether a set of logically concurrent events coincide in the physical time or not, does not change the outcome of the computation.
- A set of logically concurrent events may not have occurred at the same instant in physical time, we can assume that these events occurred at the same instant in physical time.

# 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 $send(m_{ij}) \rightarrow send(m_{kj})$,

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

- 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.

# Global State

- A collection of the local states of its components:
  - The processes and the communication channels
- The state of a process is defined by the local contents of processor registers, stacks, local memory, etc
- The state of channel depends the set of messages in transit in the channel
- An internal event changes only state of the process
- A send event changes
  - state of the process that sends the message and
  - the state of the channel on which the message is sent.
- Similarly a receive event changes
  - the state of the process that receives the message and
  - the state of the channel on which the message is received

# Global State (contd)

## Notations

- $LS_i^x$ denotes the state of $p_i$ after occurrence of event $e_i^x$ and before the event $e_i^{x+1}$
- $LS_i^0$ denotes the initial state of process $p_i$
- $LS_i^x$ is a result of the execution of all the events executed by process pi till $e_i^x$
- Let $send(m) \leq LS_i^x$ denote the fact:

$$\exists y,\ 1 \leq y \leq x \ s.t\ e_i^y = send(m)$$

- Let $rec(m)$ $(not \leq)$ $LS_i^x$ denote the fact:

$$\forall y,\ 1 \leq y \leq x \ s.t\ e_i^y \ \ (not\ equal\ to)\ rec(m)$$

# Global State (contd)

- The global state of a distributed system is a collection of the local states of the processes and the channels.

A global state GS is defined as,

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

- For a global state to be meaningful, the states of all the components of the distributed system must be recorded at the same instant
- Two important situations (Impossible !!):
  - the local clocks at processes were perfectly synchronized
  - there were a global system clock that can be instantaneously read by the processes

# A Consistent Global State

**Basic idea:**

- A state should not violate causality – an effect should not be present without its cause
- A message cannot be received if it was not sent.
- Such states are called consistent global states and are meaningful global states.
- Inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state

# A Consistent Global State

**Definition:**

➔ A global state is a consistent global state iff

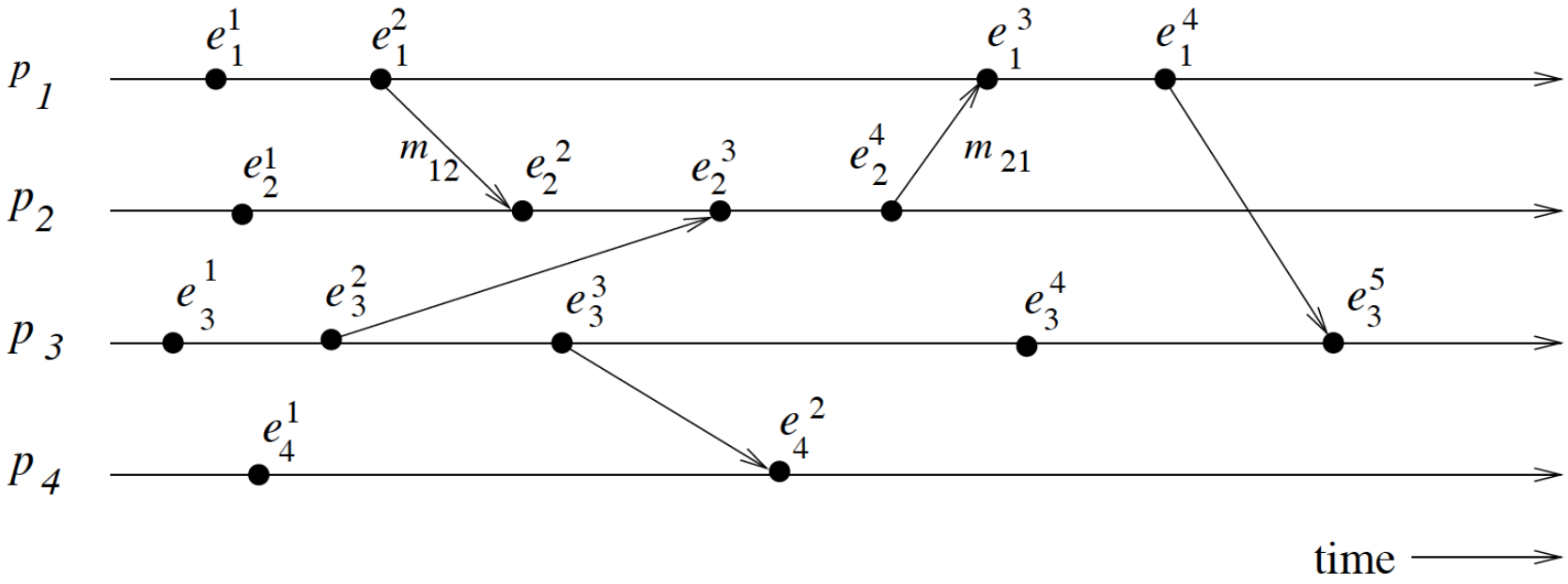
$$\forall m_{ij} : \; send(m_{ij}) \not\leq LS_i^{x_i} \;\Leftrightarrow\; m_{ij} \notin SC_{ij}^{x_i, y_j} \bigwedge rec(m_{ij}) \not\leq LS_j^{y_j}$$

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

# Consistent Global State – An Example

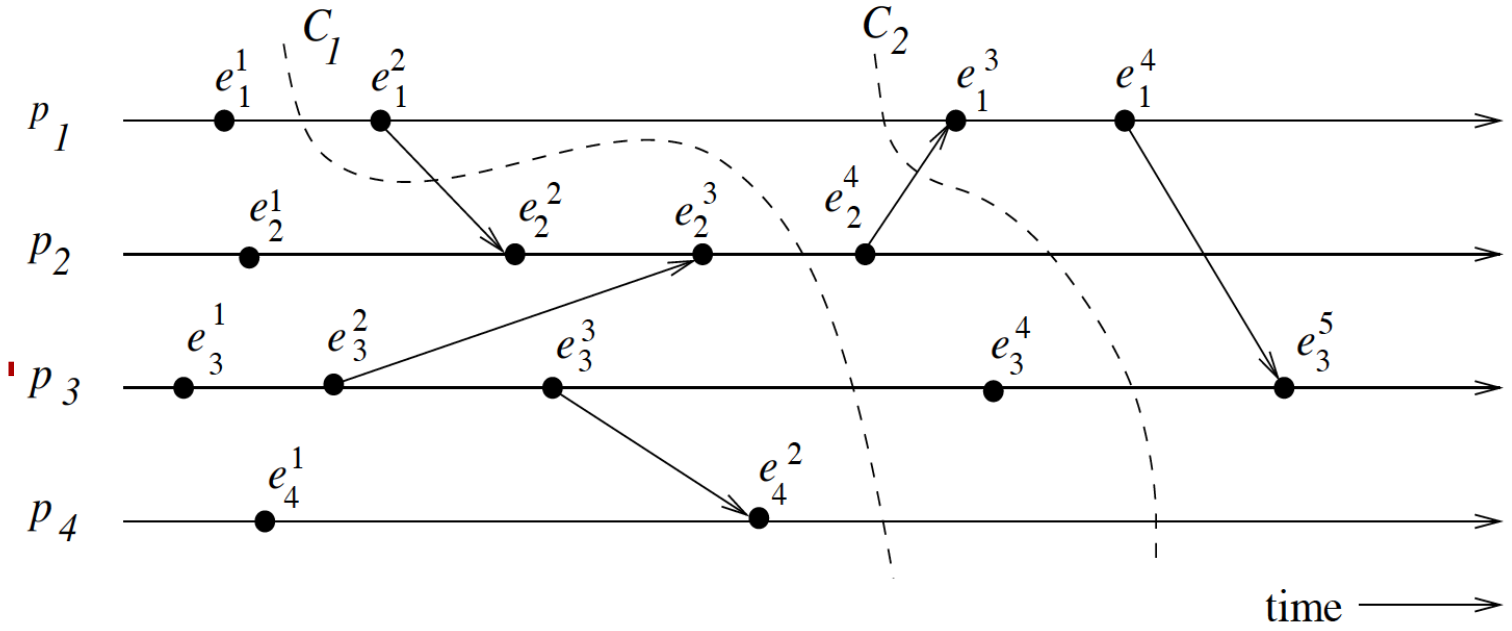


# Consistent Global State – Details

- A global state GS1 = {$LS_1^1$ , $LS_2^3$ , $LS_3^3$ , $LS_4^2$ } is inconsistent because
  - the state of $p_2$ has recorded the receipt of message $m_{12}$
  - The state of $p_1$ has not recorded its send

- A global state GS2 consisting of local states {$LS_1^2$ , $LS_2^4$ , $LS_3^4$ , $LS_4^2$ } is consistent;
- all the channels are empty except $C_{21}$ that contains message $m_{21}$.

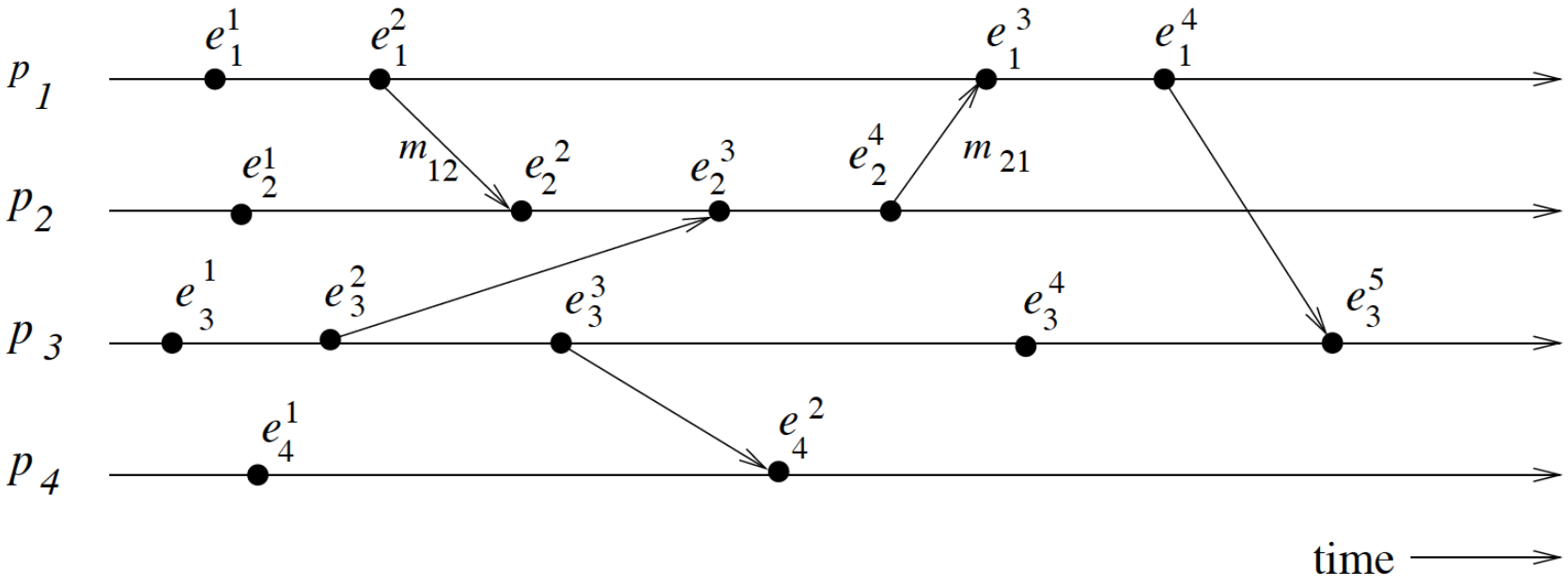
# Cuts of a Distributed Computation



# Cuts of a Distributed Computation

- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut
  - In previous figure, cut C2 is a consistent cut
- All messages that cross the cut from the PAST to the FUTURE are in transit in the corresponding consistent global state.
- A cut is inconsistent if a message crosses the cut from the FUTURE to the PAST
  - In previous figure cut C1 is an inconsistent cut

# Consistent Global State – An Example

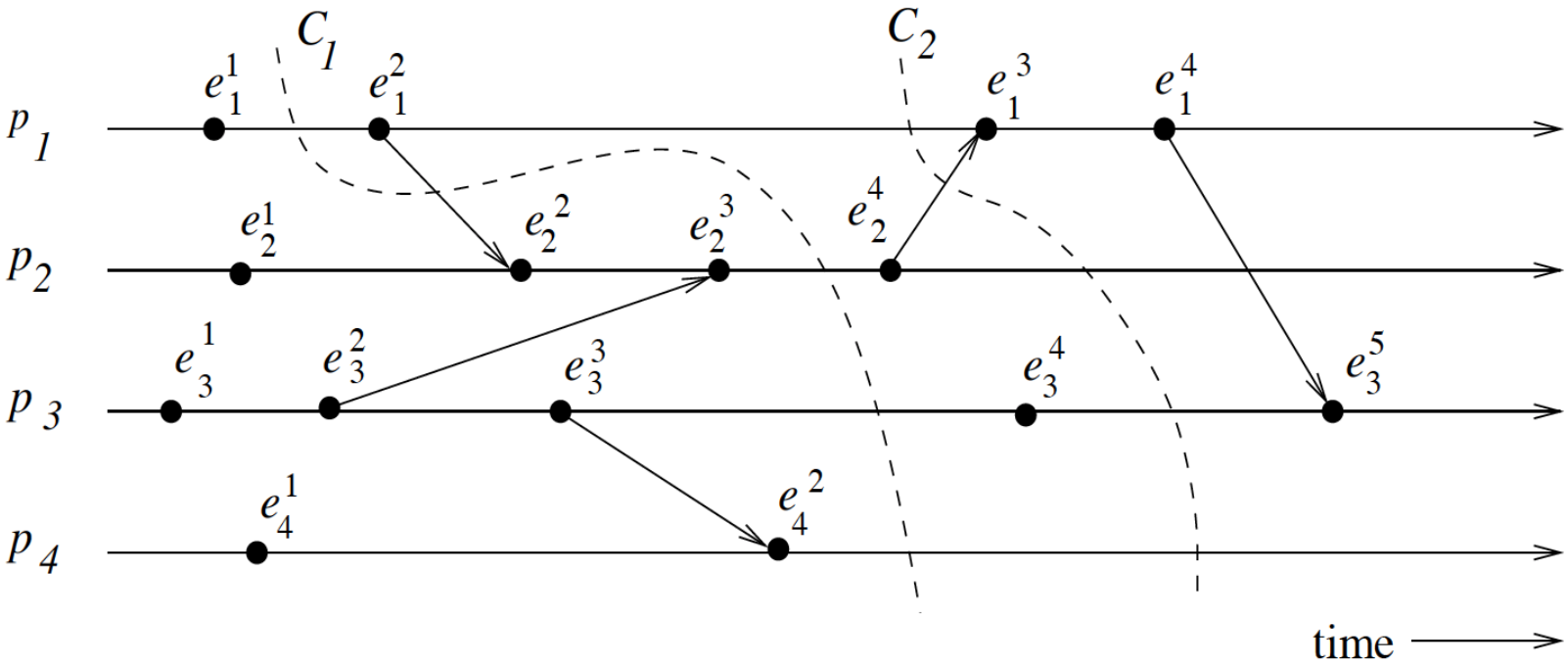


# Consistent Global State – Details

- A global state $GS_1 = \{LS_1^1 , LS_2^3 , LS_3^3 , LS_4^2 \}$ is inconsistent because
  - the state of $p_2$ has recorded the receipt of message $m_{12}$
  - The state of $p_1$ has not recorded its send message

- A global state $GS_2 = \{LS_1^2 , LS_2^4 , LS_3^4 , LS_4^2 \}$ is consistent;
  - all the channels are empty except $C_{21}$ that contains message $m_{21}$

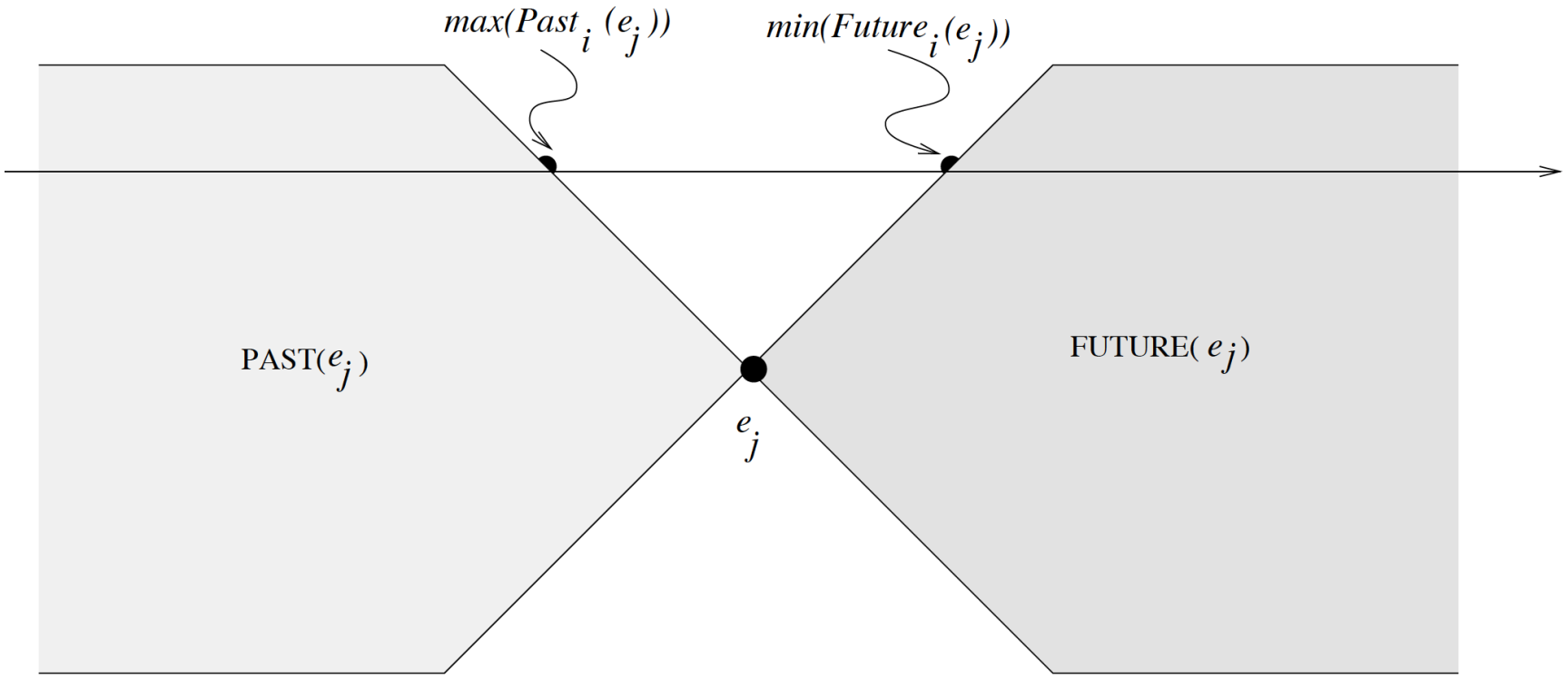
# Cuts of a Distributed Computation



# Cuts of a Distributed Computation

- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut
  - In previous figure, cut C2 is a consistent cut
- All messages that cross the cut from the PAST to the FUTURE are in transit in the corresponding consistent global state.
- A cut is inconsistent if a message crosses the cut from the FUTURE to the PAST
  - In previous figure cut C1 is an inconsistent cut

# Past and Future Cones of an event



# Physical vs Logical clocks?

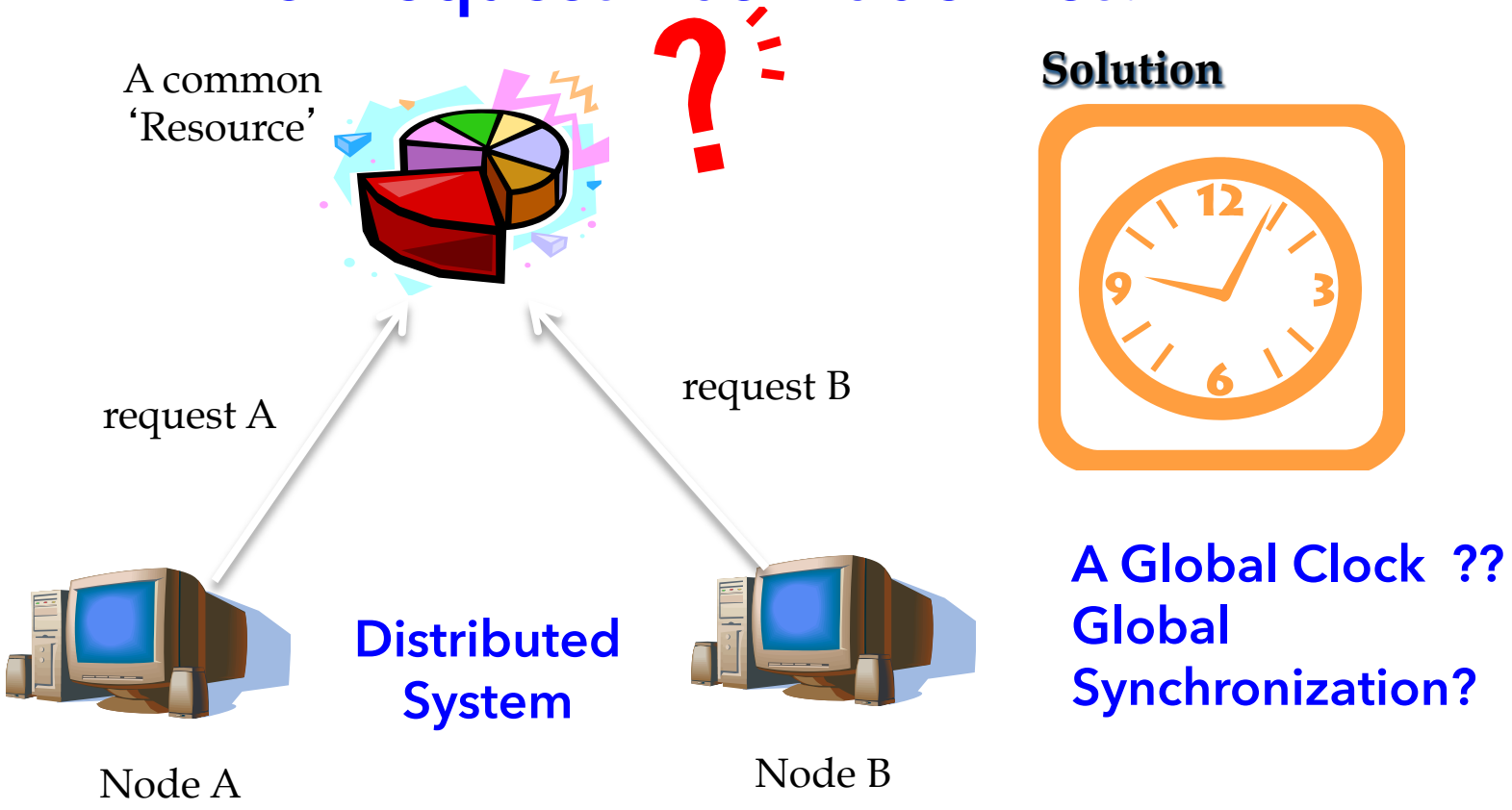
- Logical Clocks
  - Design and Implementation
- Three Different Ways
  - Scalar Time
  - Vector Time
  - Matrix Time
- Virtual Clocks
  - Time Wrap Mechanism
- Clock Synchronization
  - NTP Synchoronization Protocol

# 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

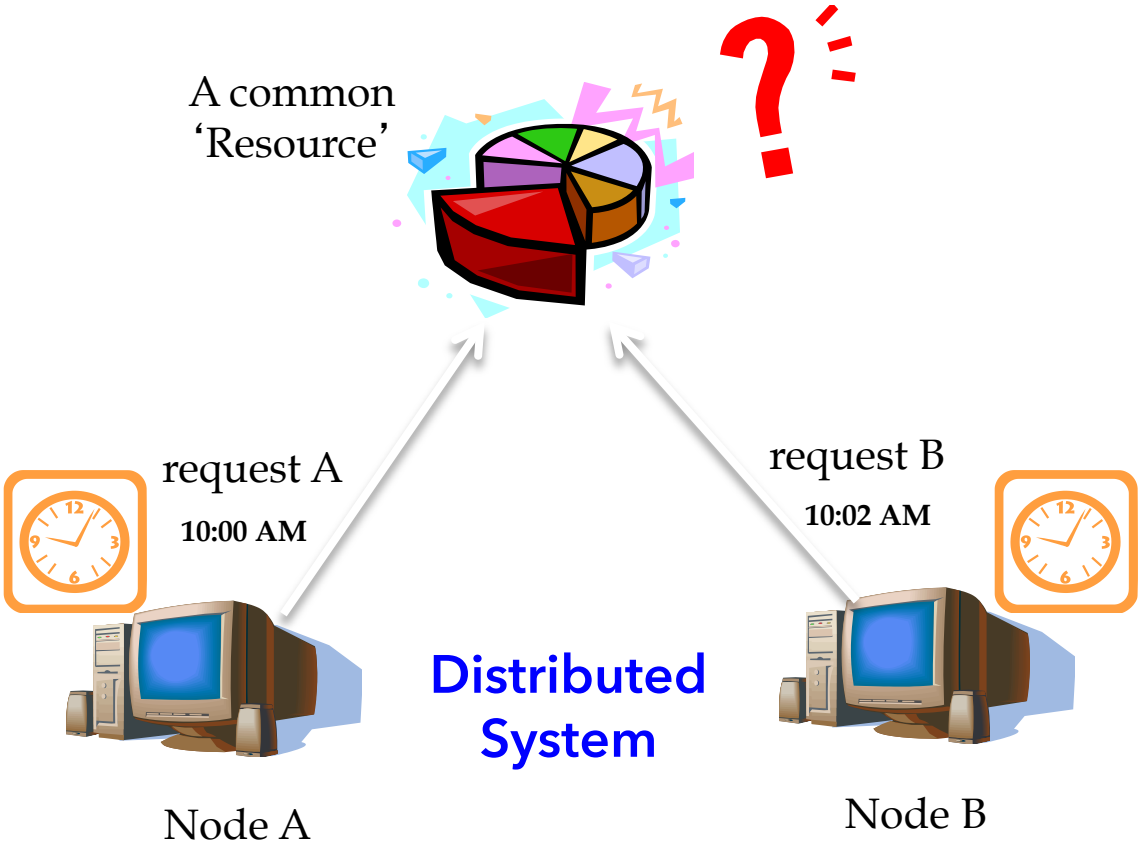
# Events Ordering – An Example

➔ Which request was made first?



# 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

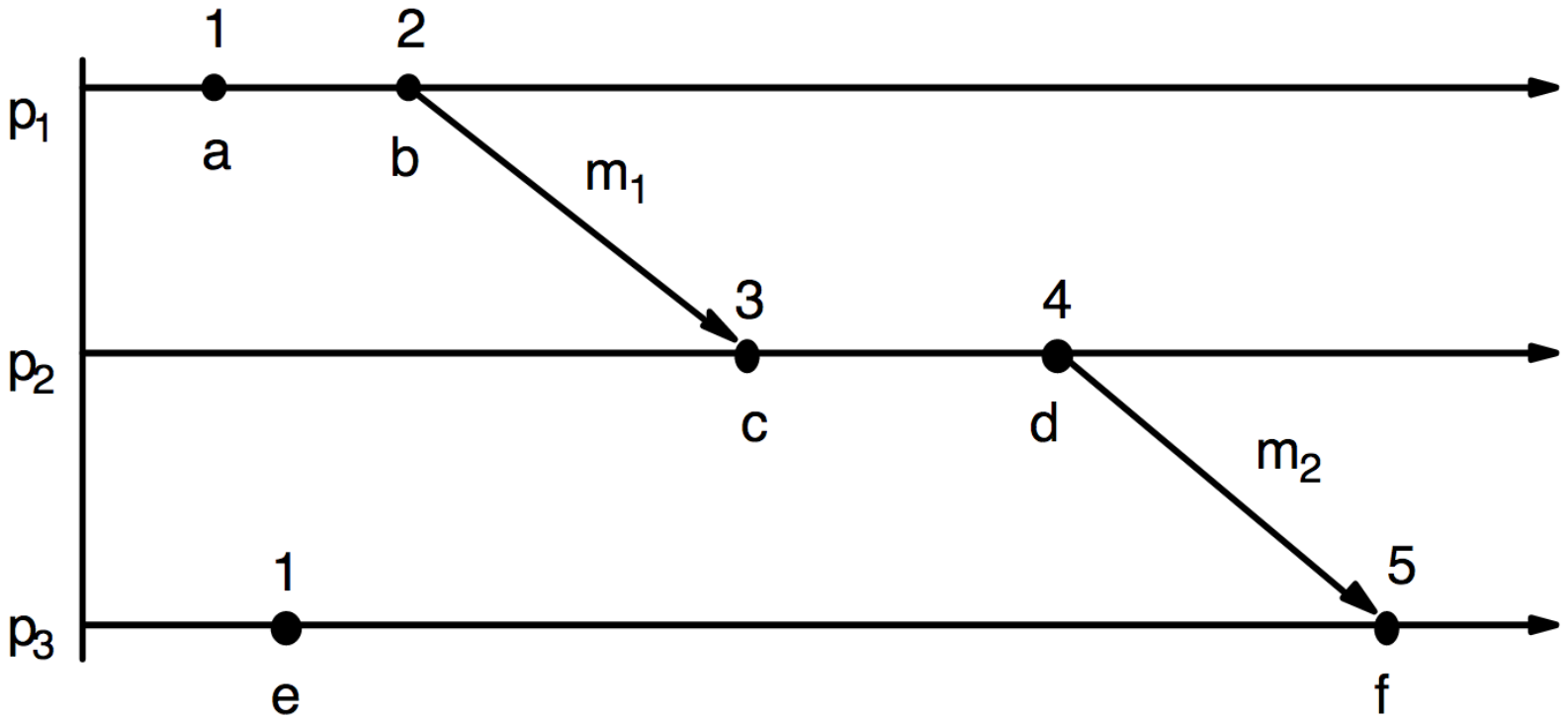
- A monotonically increasing software counter
- It does (need) not relate to a physical clock
- Each process $p_i$ has a logical clock $L_i$
- $LC_1$: $L_i$ is incremented by 1 before each event at process $p_i$
- $LC_2$:

  A) when process pi 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

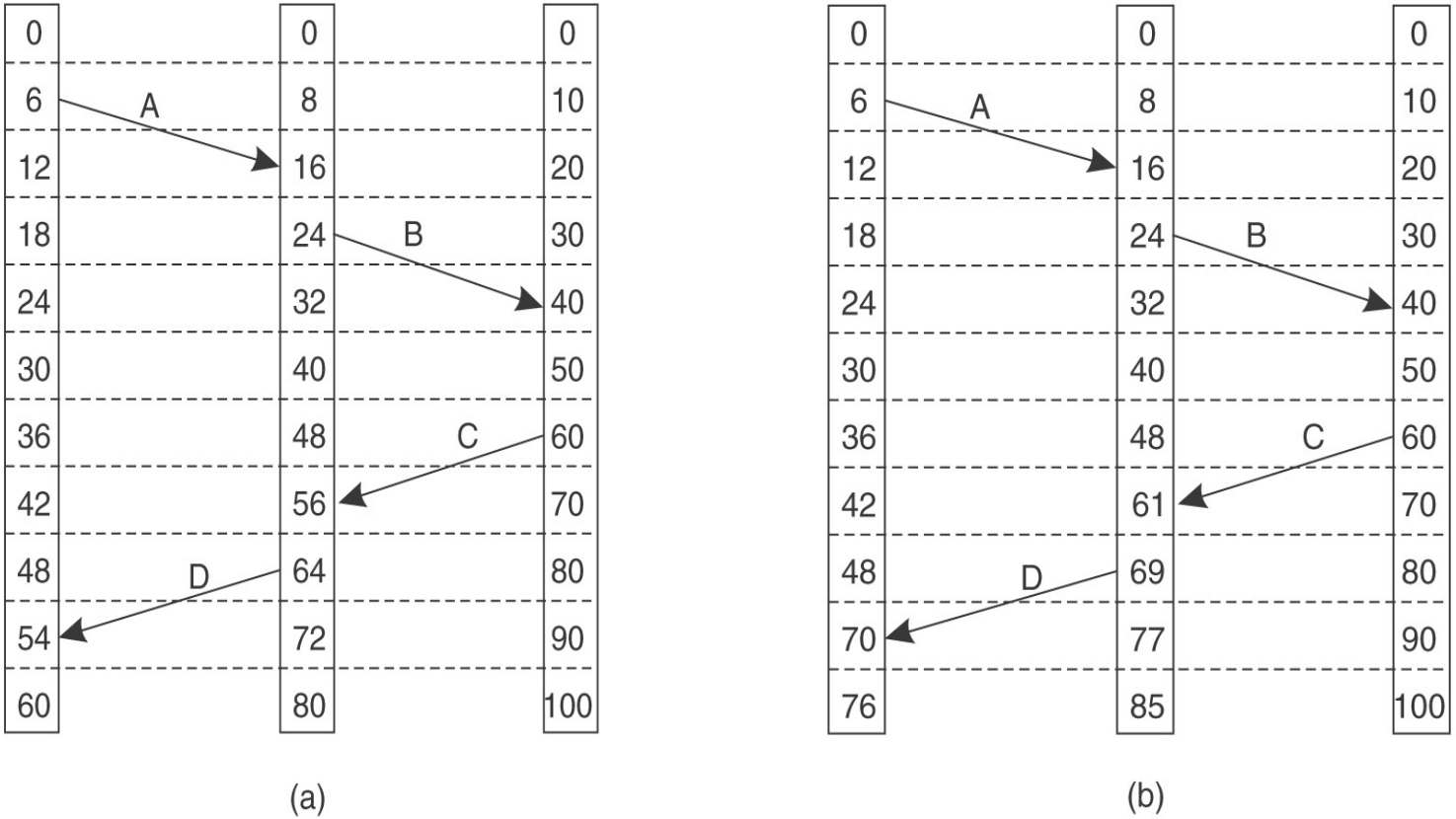



- e → e′ ⇒ L(e) < L(e′) but not vice versa
- Example: event b and event e

# 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

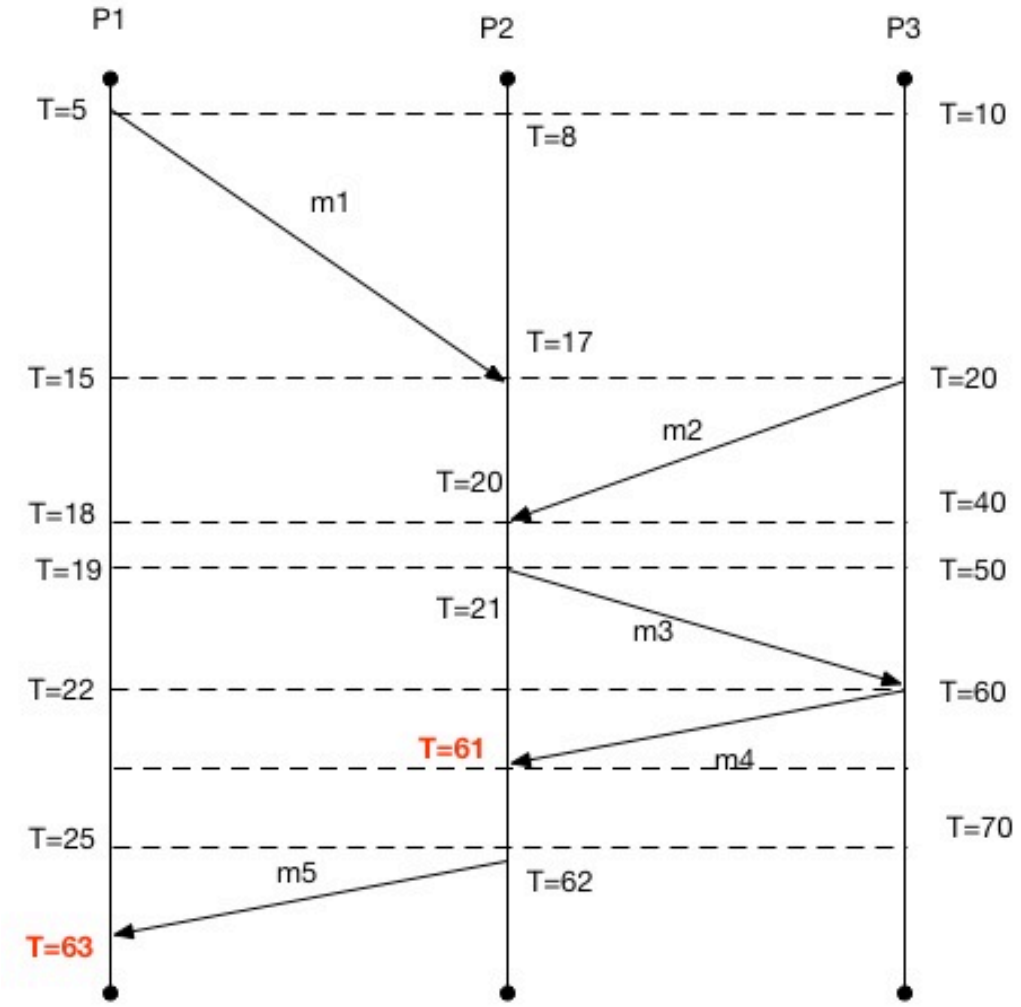


# An Illustration

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 $P_j$

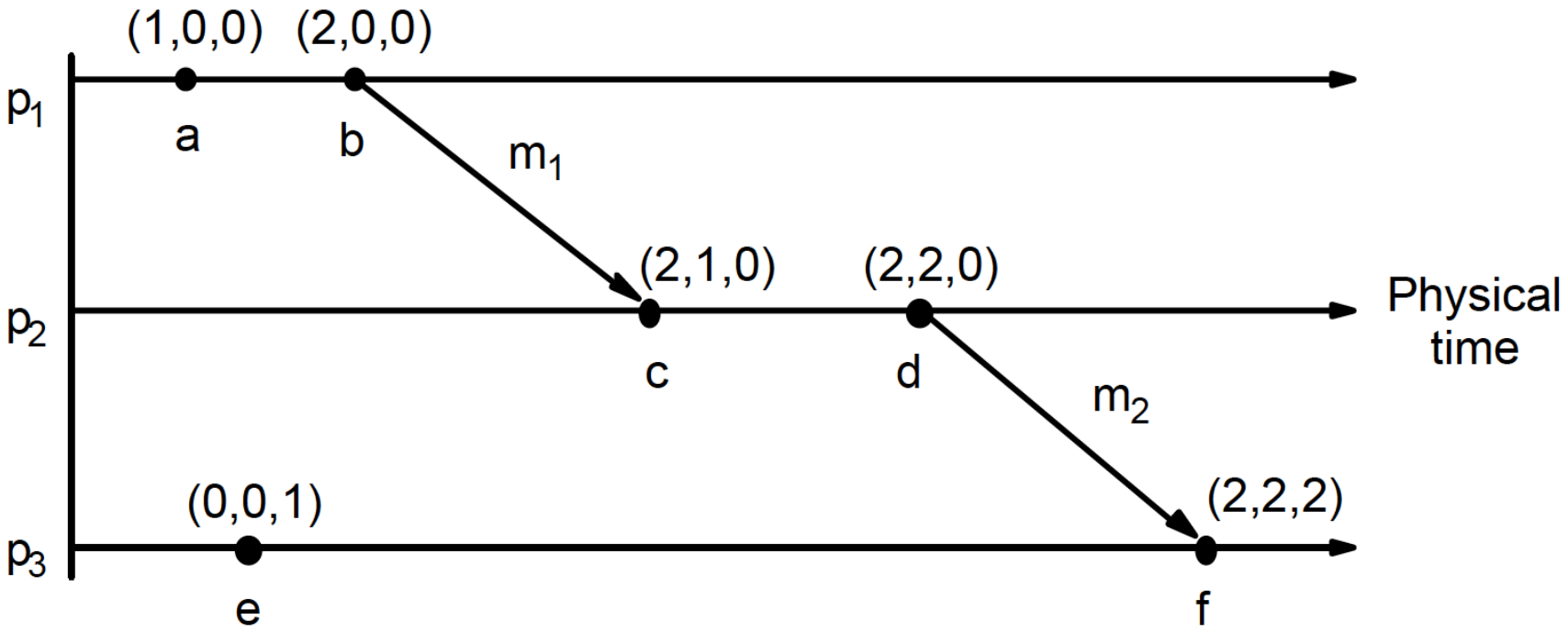
# Implementing Vector Clocks

- 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 $k$

# An Example



# 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′

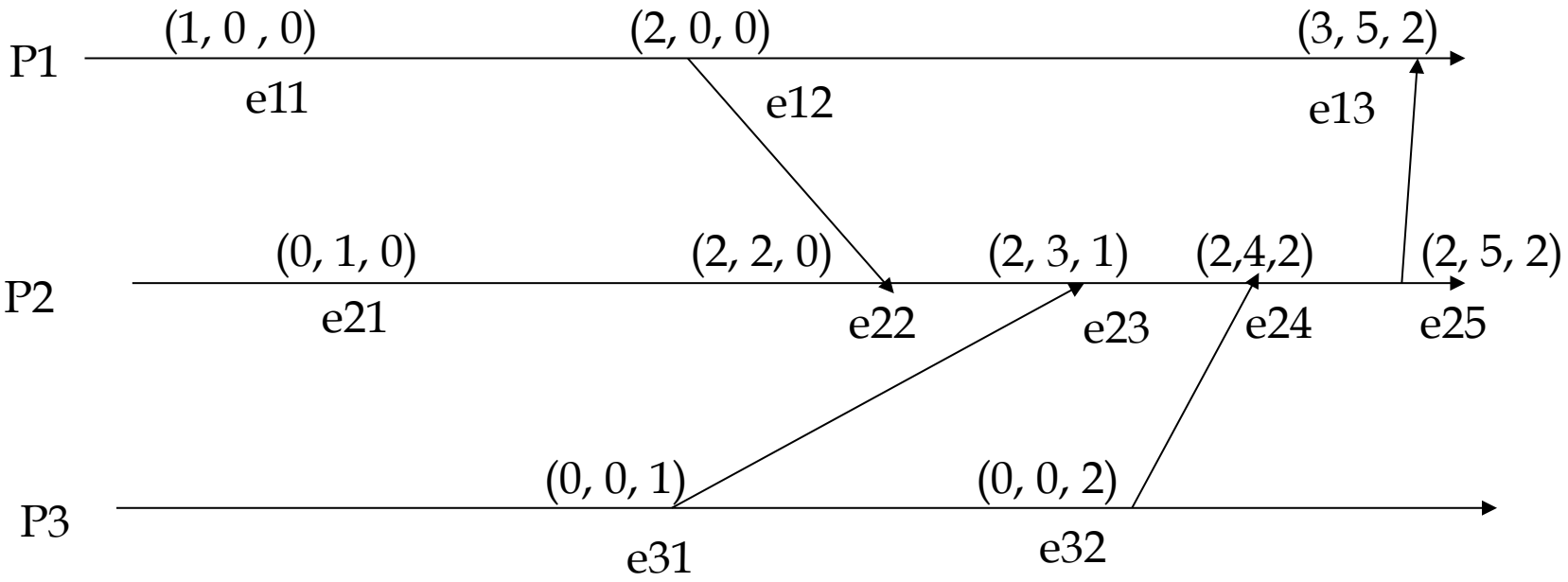
Examples:

(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** $i$

  **(3,3,5) ≤ (3,4,5)**

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

  **This implies** $e11 \rightarrow e22$

# 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

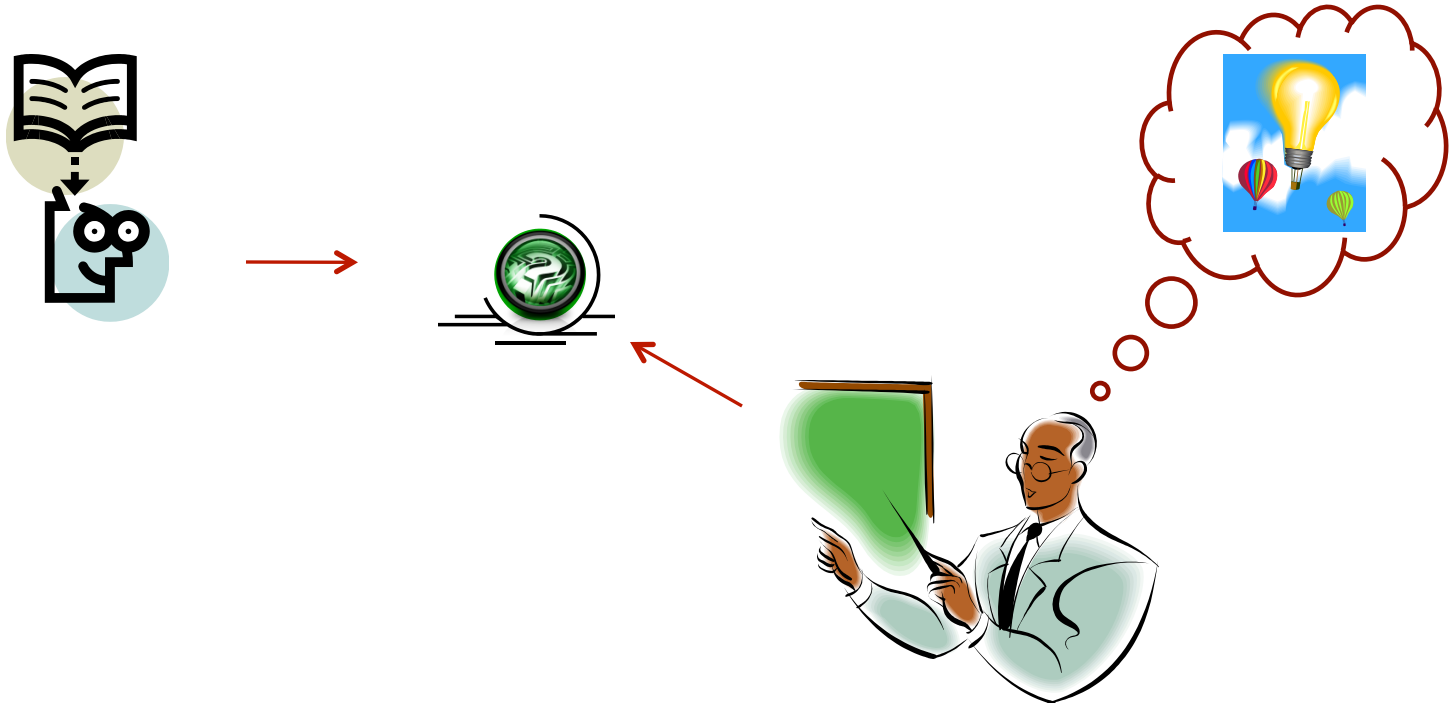
# Help among Yourselves?

- **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 ???