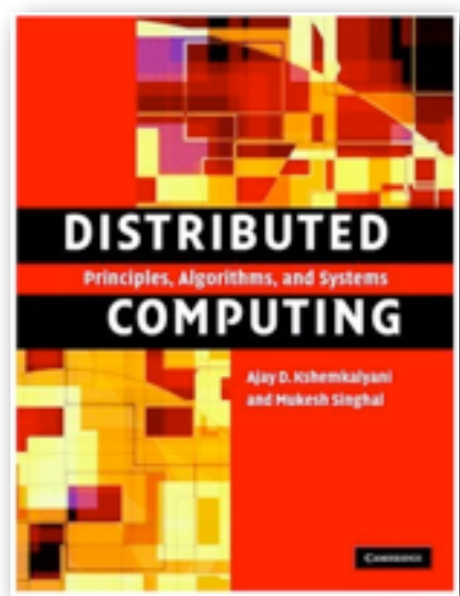
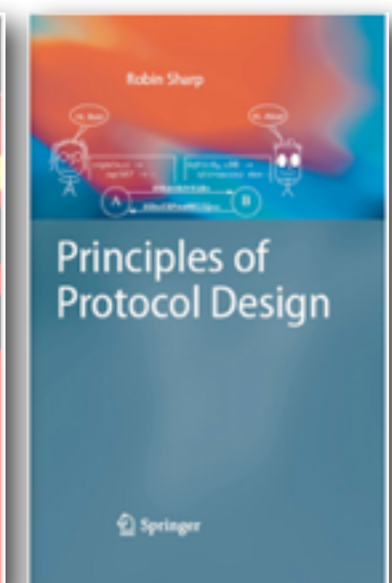
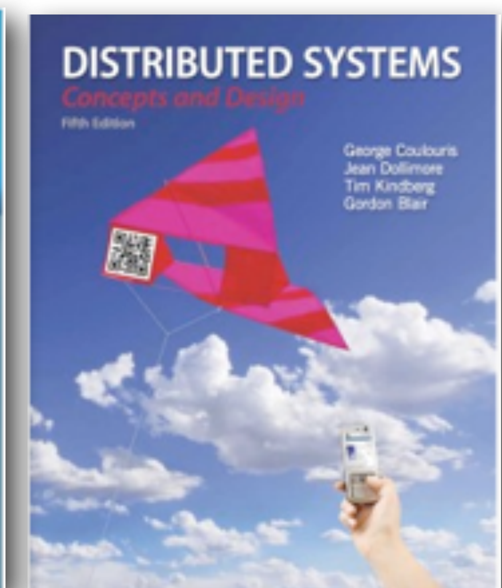
Logical Time

Nicola Dragoni
Embedded Systems Engineering
DTU Compute

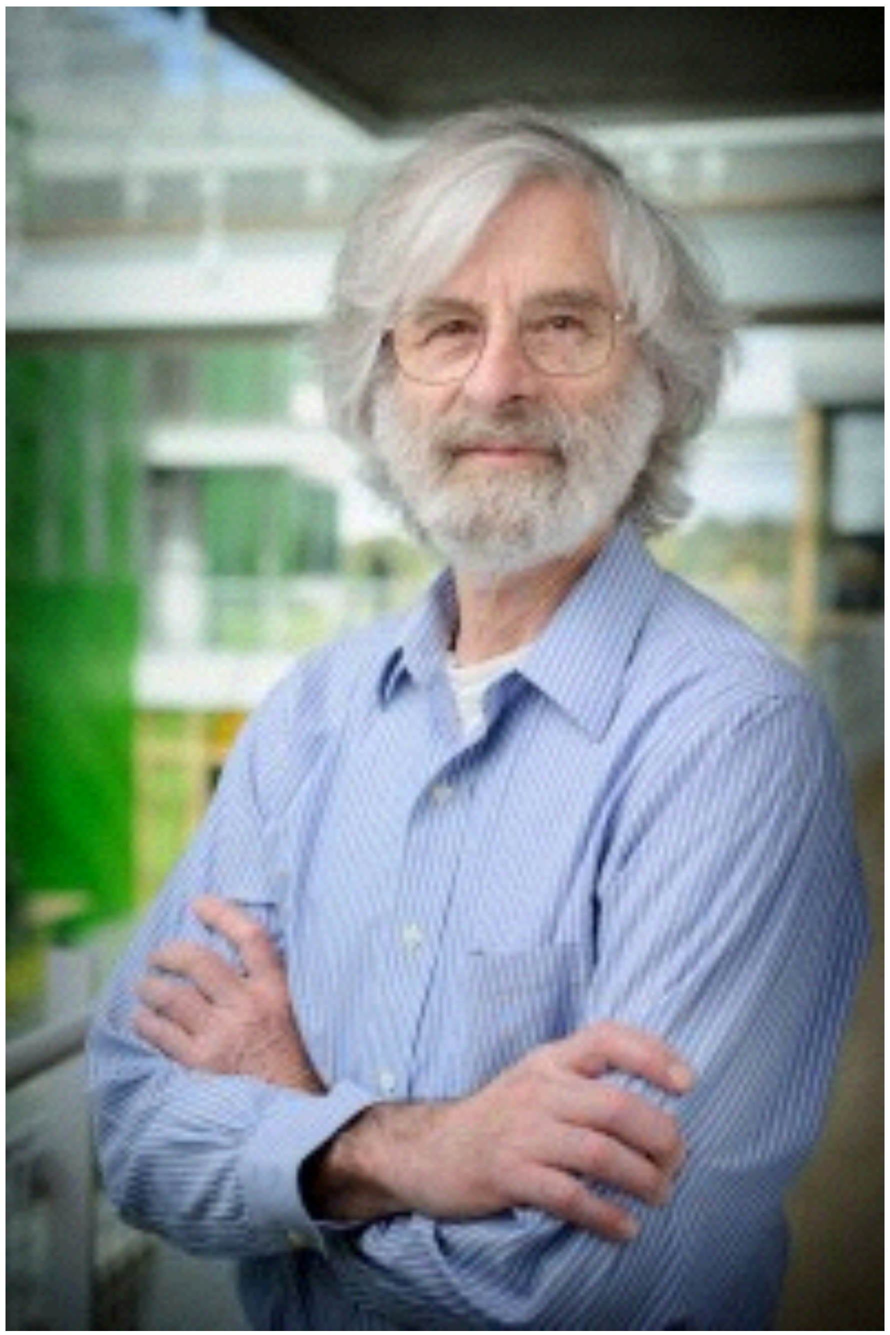
1. Introduction
2. Clock, Events and Process States
3. Logical Clocks
4. Efficient Implementation







2013 ACM Turing Award: **Leslie Lamport**



Award Citation

For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.

Background

Leslie Lamport is a Principal Researcher at Microsoft Research. He received the IEEE Emanuel R. Piore Award for his contributions to the theory and practice of concurrent programming and fault-tolerant computing. He was also awarded the Edsger W. Dijkstra Prize in Distributed Computing for his paper *"Reaching Agreement in the Presence of Faults"*. He won the IEEE John von Neumann Medal and was also elected to the U.S. National Academy of Engineering and the U.S. National Academy of Sciences.

Why Is Time Interesting?

- Ordering of events: what happened first?
  - **Storage of data** in memory, file, database, ...
  - **Requests for exclusive access** - who asked first?
  - **Interactive exchanges** - who answered first?
  - **Debugging** - what could have caused the fault?
- Causality is linked to temporal ordering:

if e_i causes e_j, then e_i must happen before e_j

(Causality, i.e. causal precedence relation, among events in a distributed system is a powerful concept in reasoning, analyzing and drawing inferences about a computation)

Computer Clocks and Timing Events

- Each computer has its own internal (physical) clock, which can be used by local processes to obtain a value of the current time

- Processes (on different computers) can associate timestamps with their events

Even if two processes read their clocks *<u>at the same time</u>, their* ***local clocks may supply different time values***

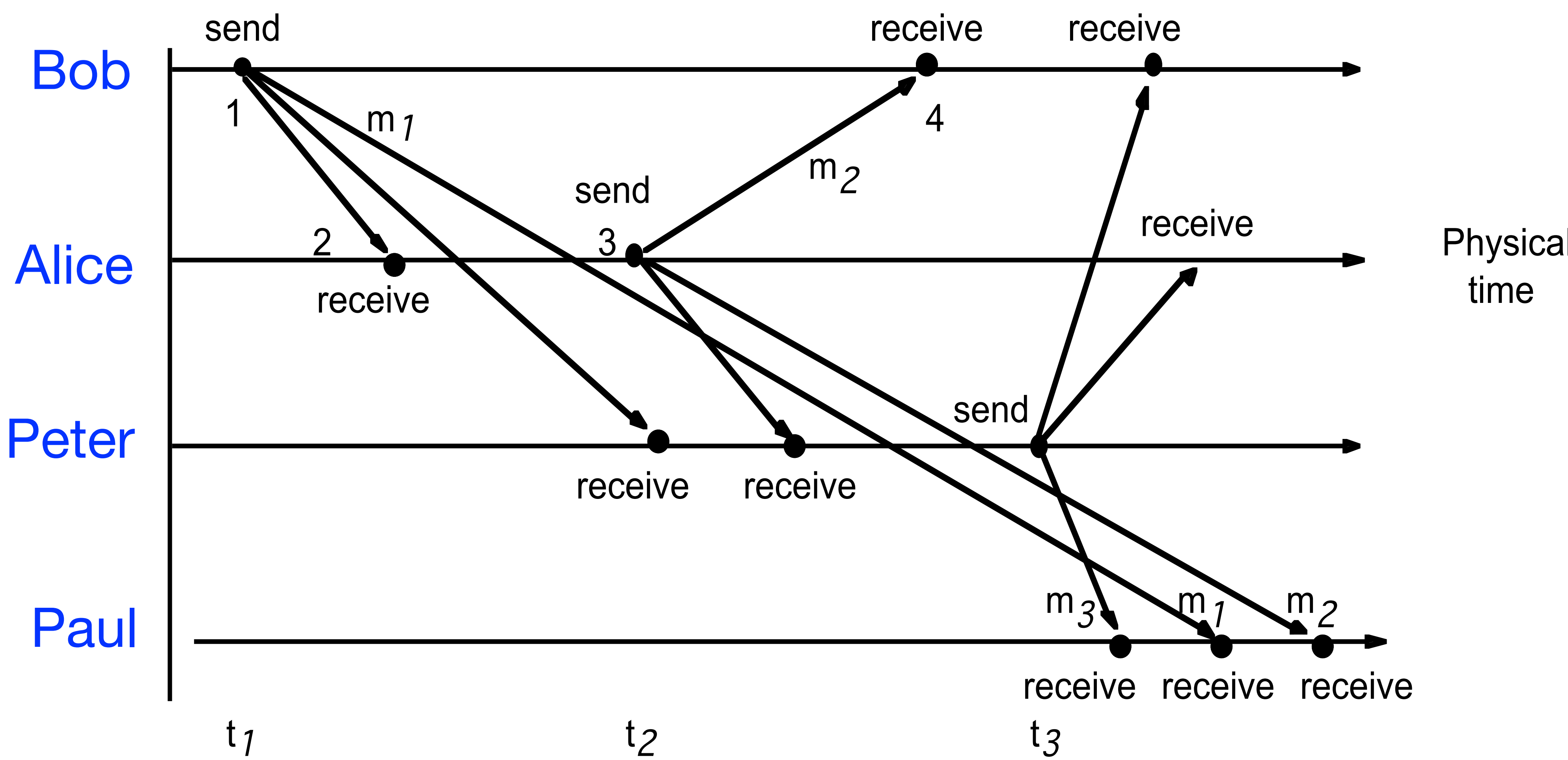
- This is because:
  - computer clocks drift from perfect time
  - their drift rates differ from one another

Clock drift rate: rate at which a computer clock deviates from a perfect reference clock

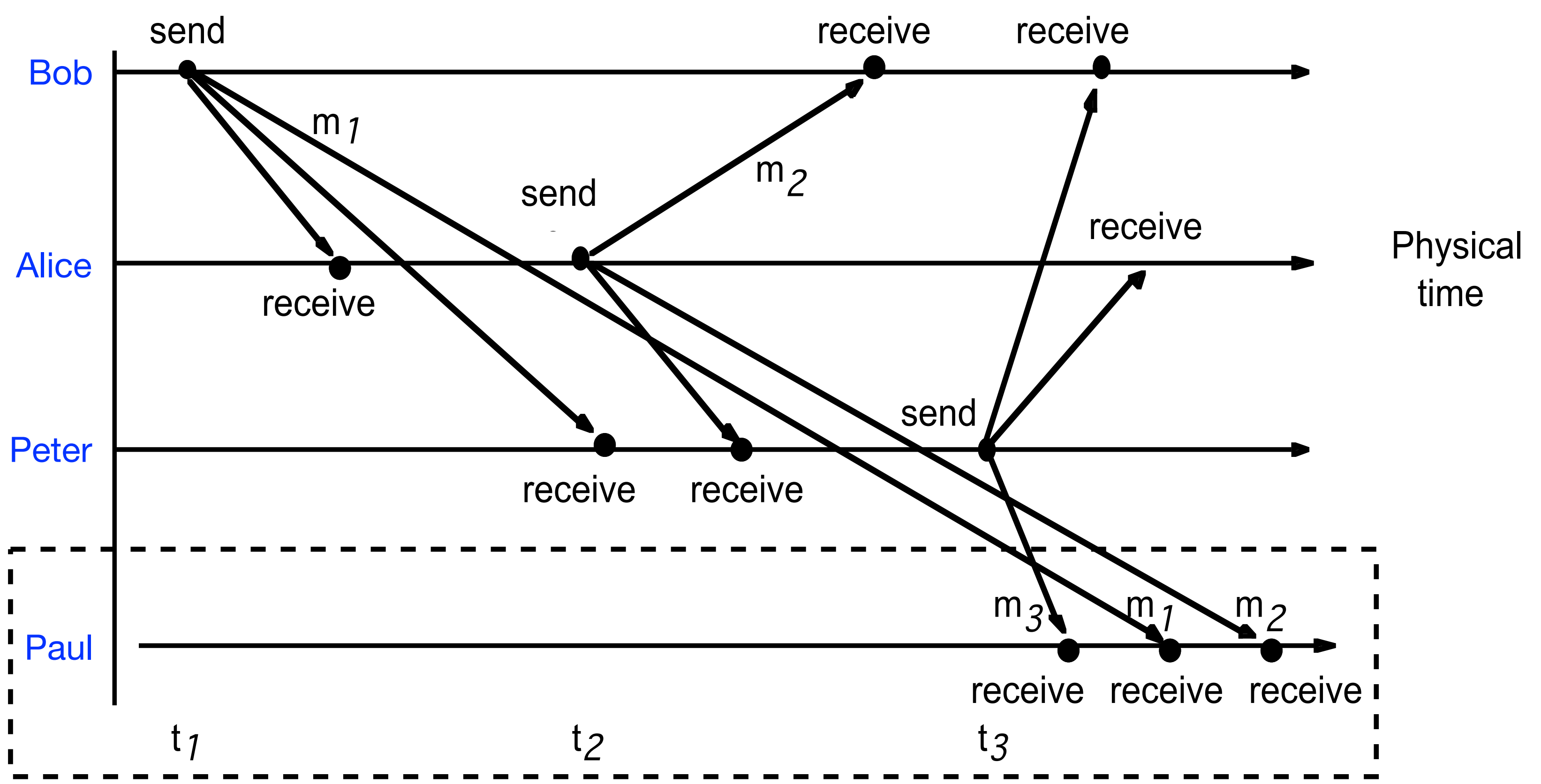
- Consequence ==> *if the physical clocks are not precisely synchronized, the causality relation between events may not be accurately captured*

Example: Real-Time Ordering of Events

- Consider the following set of exchanges between a group of email users Bob, Alice, Peter, and Paul on a mailing list:
  1. Bob sends a message with the subject *Meeting*
  2. Alice and Peter reply by sending a message with the subject *Re: Meeting*



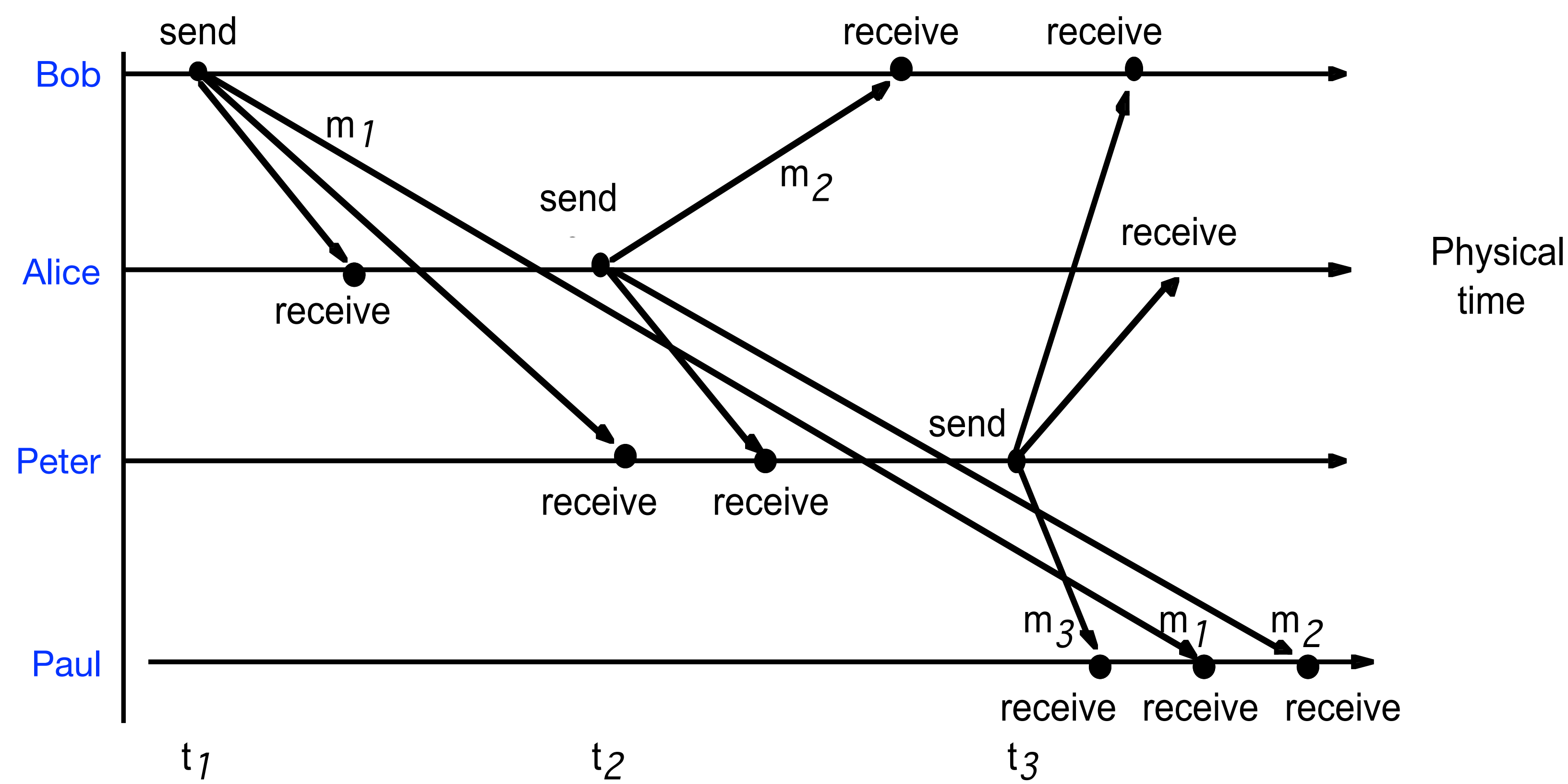
Example: Real-Time Ordering of Events (cont.)



- Due to the independent delays in message delivery, the messages may be delivered in the following order:

Paul's Inbox	
From	**Subject**
Peter	Re:Meeting
Bob	Meeting
Alice	Re: Meeting

Example: Real-Time Ordering of Events (cont.)



- If the clocks could be synchronized: messages $\mathbf{m_1}$, $\mathbf{m_2}$ and $\mathbf{m_3}$ would carry times $\mathbf{t_1}$, $\mathbf{t_2}$ and $\mathbf{t_3}$ where $\mathbf{t_1 < t_2 < t_3}$ (time ordering)

	Paul's Inbox	
	From	**Subject**
$\mathbf{t_1}$	Bob	Meeting
$\mathbf{t_2}$	Alice	Re:Meeting
$\mathbf{t_3}$	Peter	Re: Meeting

The Problem

- The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems

- **Usually** causality is tracked using physical time

- In distributed systems, it is not possible to have a global physical time!

What We Want…

- Capture the notion of **causality**: whether an event (sending or receiving a message) at one process occurred *before*, *after* or *concurrently* with another event at another process

- The execution of a system described in terms of events and their ordering *despite the lack of accurate clocks*

No Accurate Clocks… but Event Ordering!

Idea... Logical Time!

- Since clocks cannot be synchronized perfectly across a distributed system, **logical time** can be used **to provide an ordering among the events** (at processes running in different computers in a distributed system) **without recourse to clocks**

- Let us consider our email ordering problem.. what do we know ***logically***?

✓ A message is received after it was sent

Bob sends $\mathbf{m_1}$ *before Alice receives* $\mathbf{m_1}$
Alice sends $\mathbf{m_2}$ *before Bob receives* $\mathbf{m_2}$

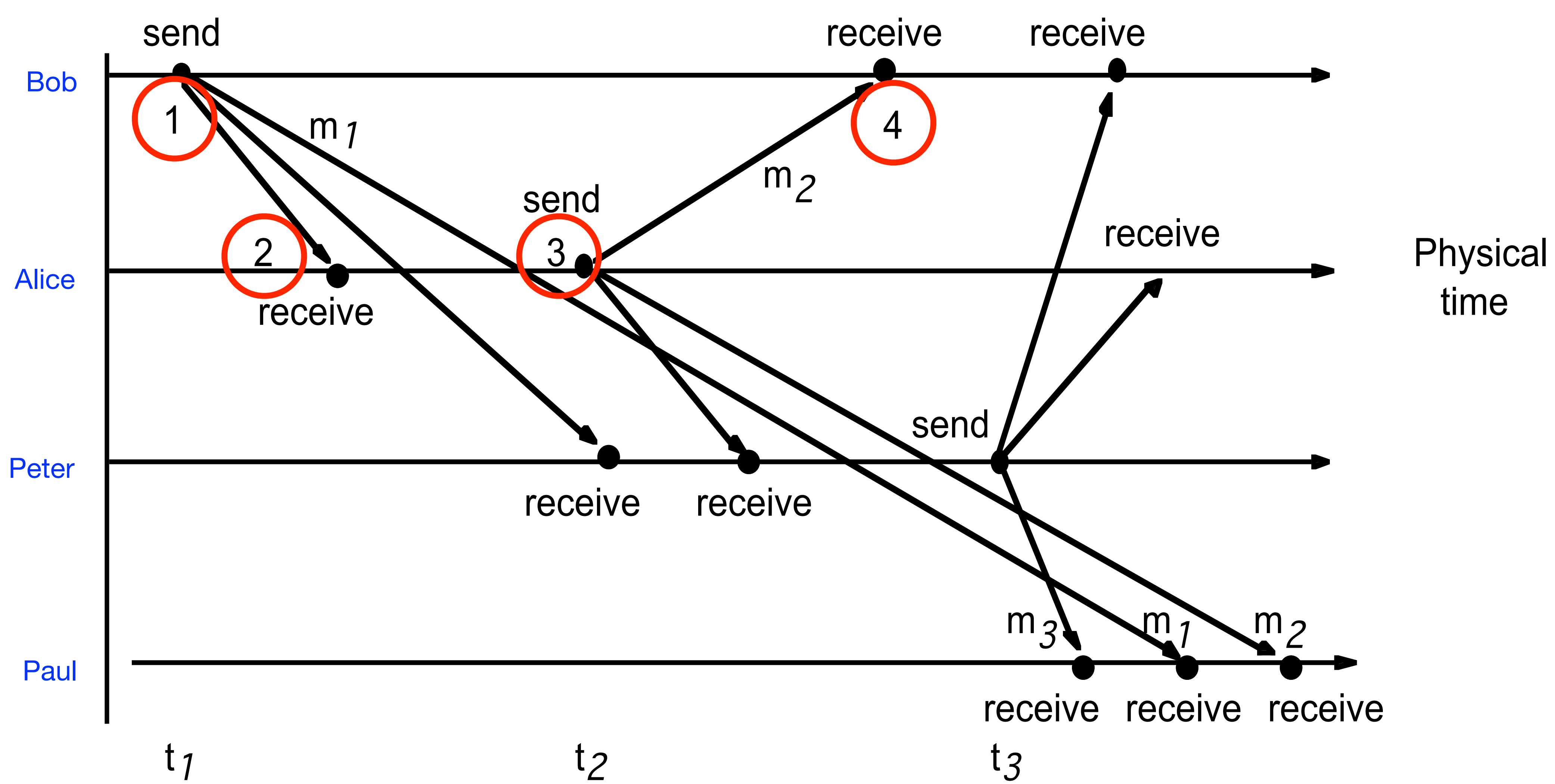
✓ Replies are sent after receiving messages

Alice receives $\mathbf{m_1}$ *before sending* $\mathbf{m_2}$

Example: Real-Time Ordering of Events (cont.)

- Logical time takes this idea further by assigning a number to each event corresponding to its logical ordering

- As a result, later events have higher numbers than earlier ones



The Idea... in 1 Slide

- Every process has a logical clock that is advanced using a set of rules

- Every event is assigned a timestamp

- Timestamps obey the fundamental monotonicity property:

  if an event **a** causally affects an event **b**,
  then the timestamp of **a** is smaller than the timestamp of **b**

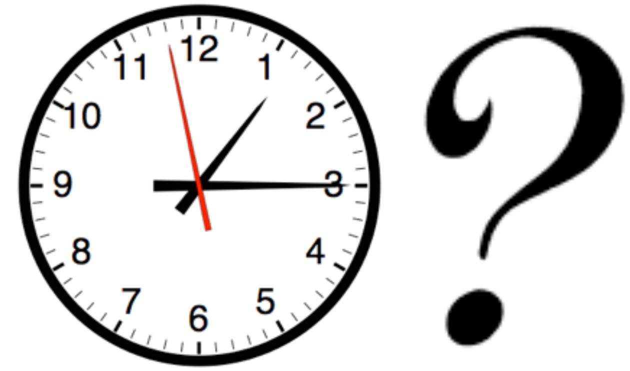
- WHAT WE WANT: causality between events can be generally inferred from their timestamps

...more formally...

Distributed System Model

- We consider the following asynchronous distributed system:
  - **n** processes **p_i, $i = 1, \ldots, n$**
  - each process executes on a single processor
  - processors do **not** share memory --> processes communicate only by message passing
  - Actions of a process **p_i**: communicating actions (Send or Receive) or state transforming actions (such as changing the value of a variable)
- Event: occurrence of a single action that a process carries out as it executes

What Do We Know About Time?



- We **cannot** synchronize clocks *perfectly* across a distributed system

  ➡ *We cannot in general use physical time to find out the order of any arbitrary pair of events occurring within a distributed system* **[Lamport, 1978]**

- The sequence of events *within a single process* $\mathbf{p_i}$ can be placed in a total ordering, denoted by the relation $\rightarrow_i$ ("occurs before") between the events

  $\mathbf{e} \rightarrow_i \mathbf{e'}$ if and only if the event **e** occurs before **e'** at $\mathbf{p_i}$

  In other words: if two events occurred at the same process $\mathbf{p_i}$, then they occurred in the order in which $\mathbf{p_i}$ observes them

- Whenever a message is sent between two processes, the event of sending the message occurred before the event of receiving the message

Happened-Before Relation (→)

- Lamport's **happened-before relation** → (or **causal ordering**):

  HB1: If ∃ process p_i : $e \rightarrow_i e'$, then $e \rightarrow e'$

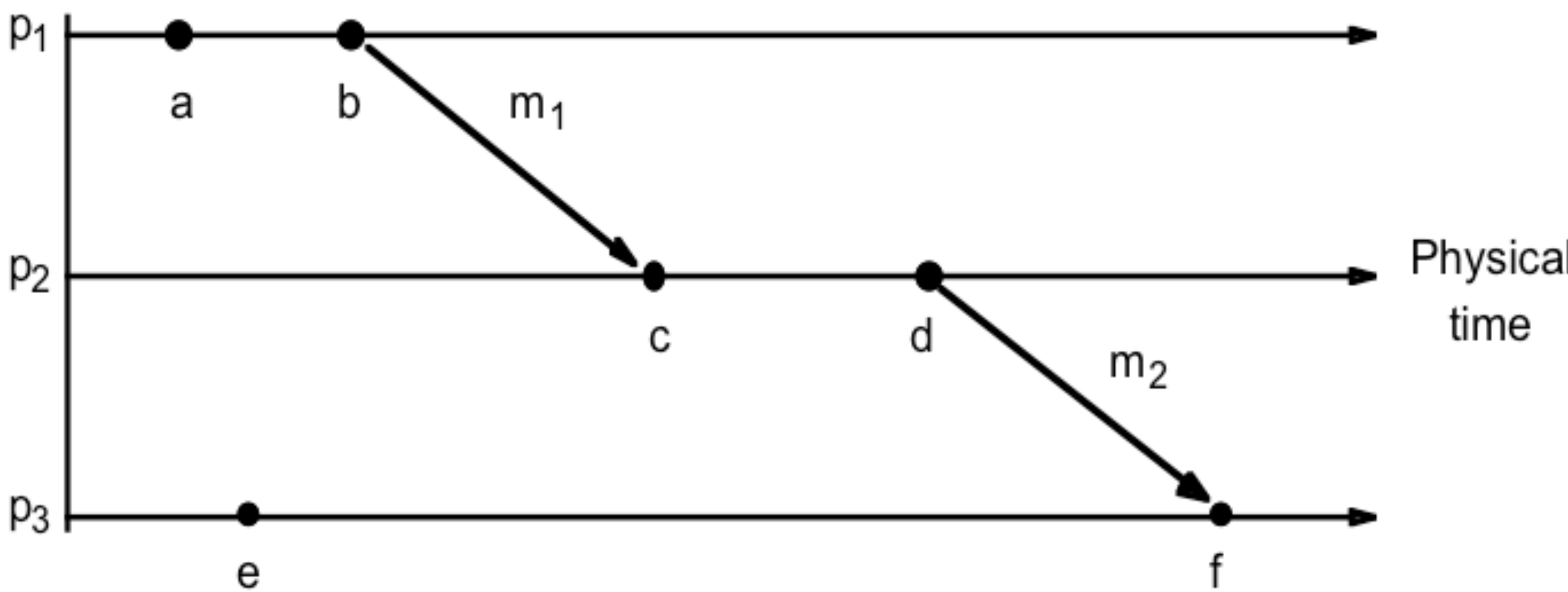
  HB2: For any message m, **send(m) → receive(m)**

  HB3: If e, e', e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$

- Thus, if $e \rightarrow e'$, then we can find a series of events $e_1, e_2, \ldots, e_n$ occurring at one or more processes such that
  - $e = e_1$
  - $e' = e_n$
  - for $i = 1, 2, \ldots, N-1$ either **HB1** or **HB2** applies between e_i and e_{i+1}

  In other words: either they occur in succession at the same process, or there is a message **m** such that e_i = **send(m)** and e_{i+1} = **receive(m)**

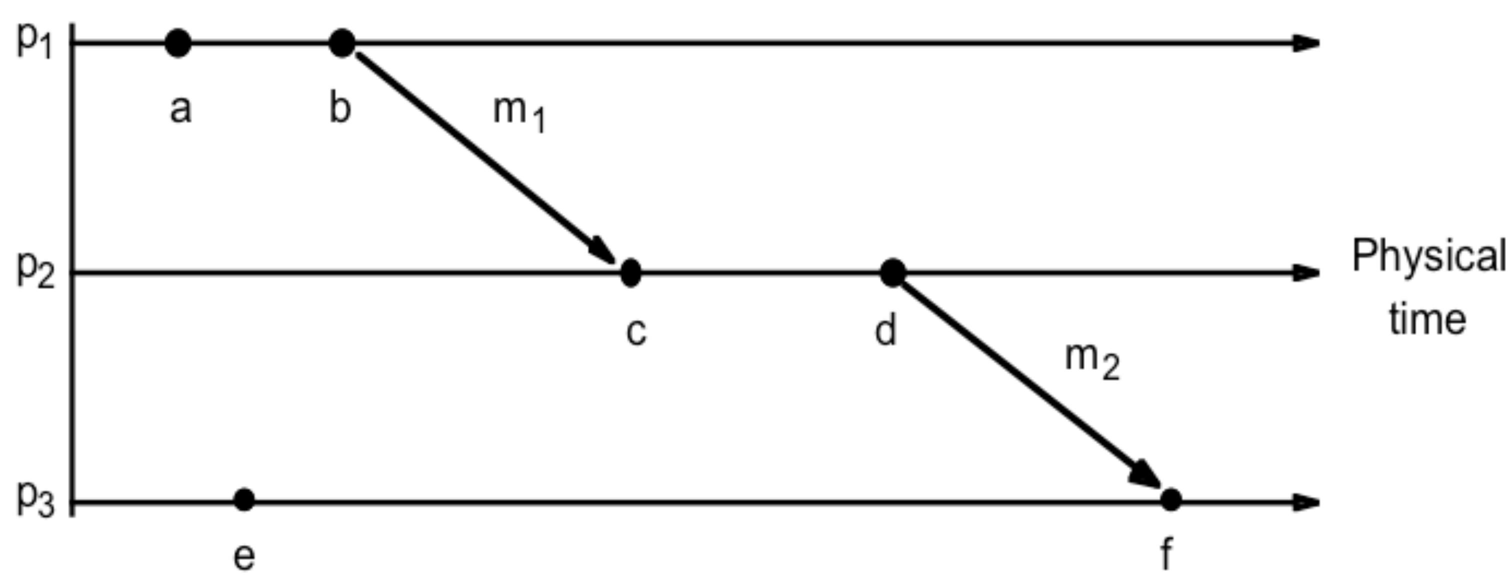
[Happened Before Relation] Example



- **a → b**, since the events occur in this order at process **p_1 (a $\rightarrow_1$ b)**
- **c → d**
- **b → c**, since these events are the sending and reception of message **m_1**
- **d → f**, similarly
- Combining these relations, we may also say that, for example, **a → f**

Happened-Before Relation (→)

- Note that the → relation is an IRREFLEXIVE PARTIAL ORDERING on the set of all events in the distributed system
  - Irreflexivity: **¬(a → a)**
  - Partial ordering: not all the events can be related by →



  - **¬(a → e)** and **¬(e → a)** since they occur at different processes and there is no chain of messages intervening between them
  - We say **a** and **e** are not ordered by →; **a** and **e** are concurrent **(a || e)**

Logical Clocks

- Each process p_i keeps its own logical clock, L_i, which it uses to apply so-called Lamport timestamps to events
- Logical clock: a MONOTONICALLY increasing software counter, which associates a value in an ORDERED domain with each event in a system

Definition [Logical Clock] A local logical clock **L** is a function that maps an event **e** ∈ **H** in a distributed system to an element in the time domain **T**, denoted as **L(e)** and called the **timestamp** of **e**, and is defined as follows:

$$\mathbf{L : H \rightarrow T}$$

such that the following monotonicity property (***clock consistency property***) is satisfied:

for two events e and e' ∈ H, e → e' ⇒ L(e) < L(e')

- N.B.: the values of a logical clock need bear no particular relationship to any physical clock

Logical Clocks Rules

$$e \rightarrow e' \Rightarrow L(e) < L(e')$$

- To match the definition of $\rightarrow$, we require the following clock rules:

CR1: If ∃ process p_i such that $e \rightarrow_i e'$, then $L_i(e) < L_i(e')$

CR2: If **a** is the sending of a message by p_i and **b** is the receipt of the same message by p_j, then $L_i(a) < L_j(b)$

CR3: If **e, e', e''** are 3 events : $L(e) < L(e')$ and $L(e') < L(e'')$ then $L(e) < L(e'')$

Ok, but how to use these rules *in practice*?

Logical Clocks... in Practice!

- To capture the → relation numerically: processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

LC1: L_i is incremented before each event is issued at process p_i: $L_i := L_i + 1$

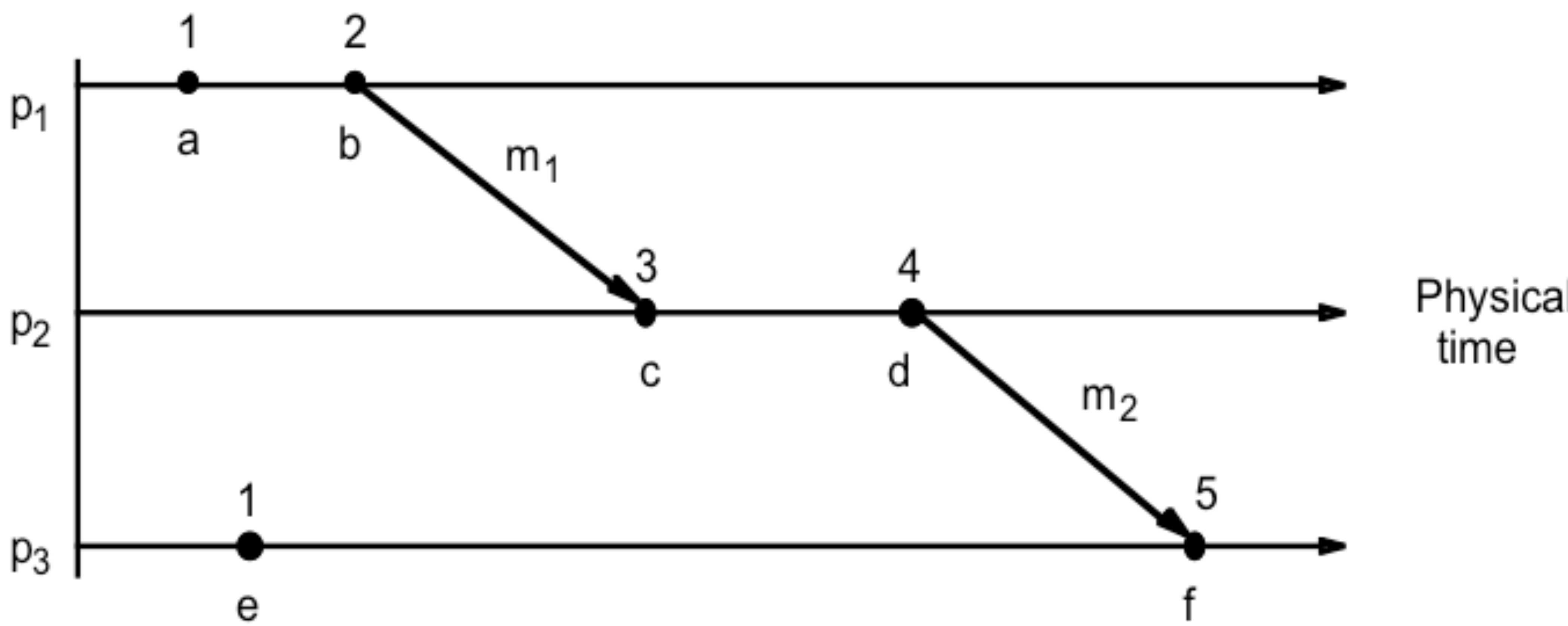
LC2: (a) When p_i sends a msg **m**, it piggybacks on **m** the value $t = L_i$

(b) On receiving **(m, t)**, a process p_j

- computes $L_j := \max(L_j, t)$
- applies **LC1**
- timestamp the event **receive(m)**

- Although we increment clocks by 1, we can consider any value **d > 0**
- Clocks which follow these rules are known as LAMPORT LOGICAL CLOCKS

[Lamport Clocks] Example 1

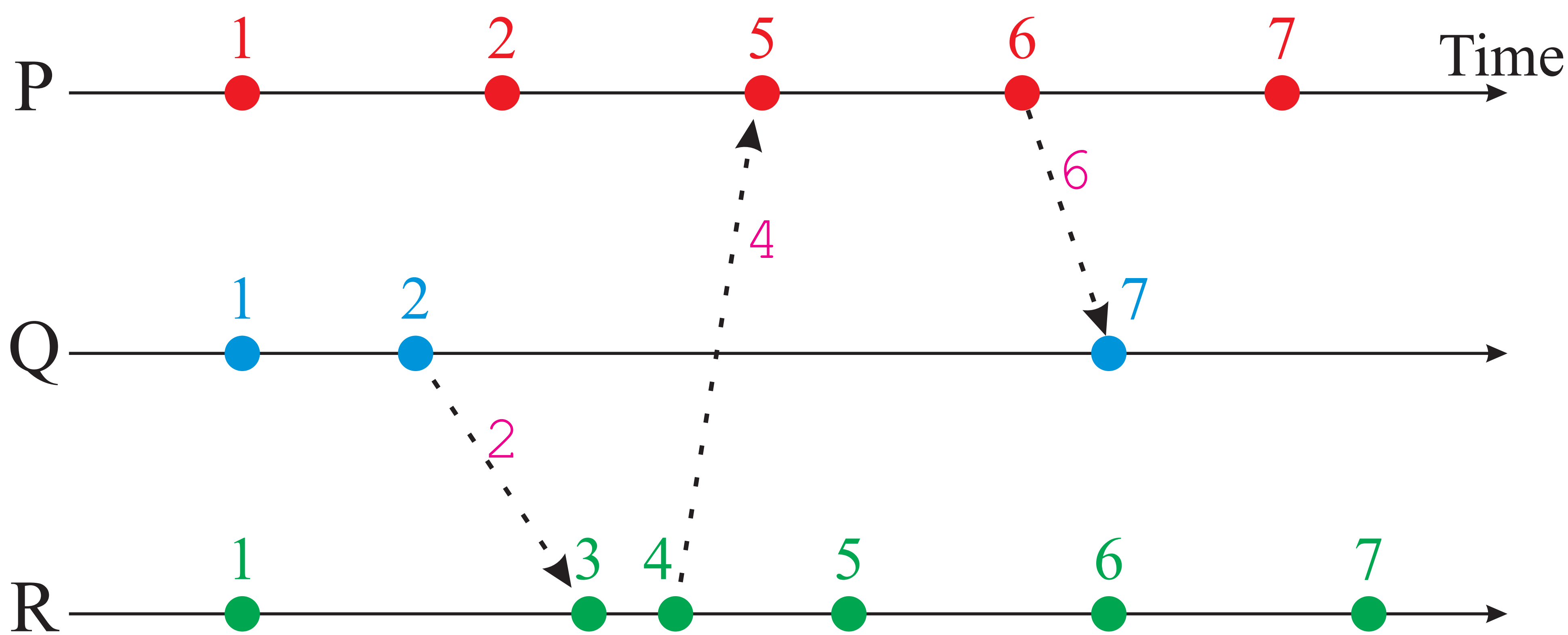


LC1: L_i is incremented before each event is issued at process p_i: $L_i := L_i + 1$

LC2: (a) When p_i sends a msg **m**, it piggybacks on **m** the value $t = L_i$

(b) On receiving **(m, t)**, a process p_j computes $L_j := \max(L_j, t)$ and then applies **LC1** before timestamping the event **receive(m)**

[Lamport Clocks] Example 2

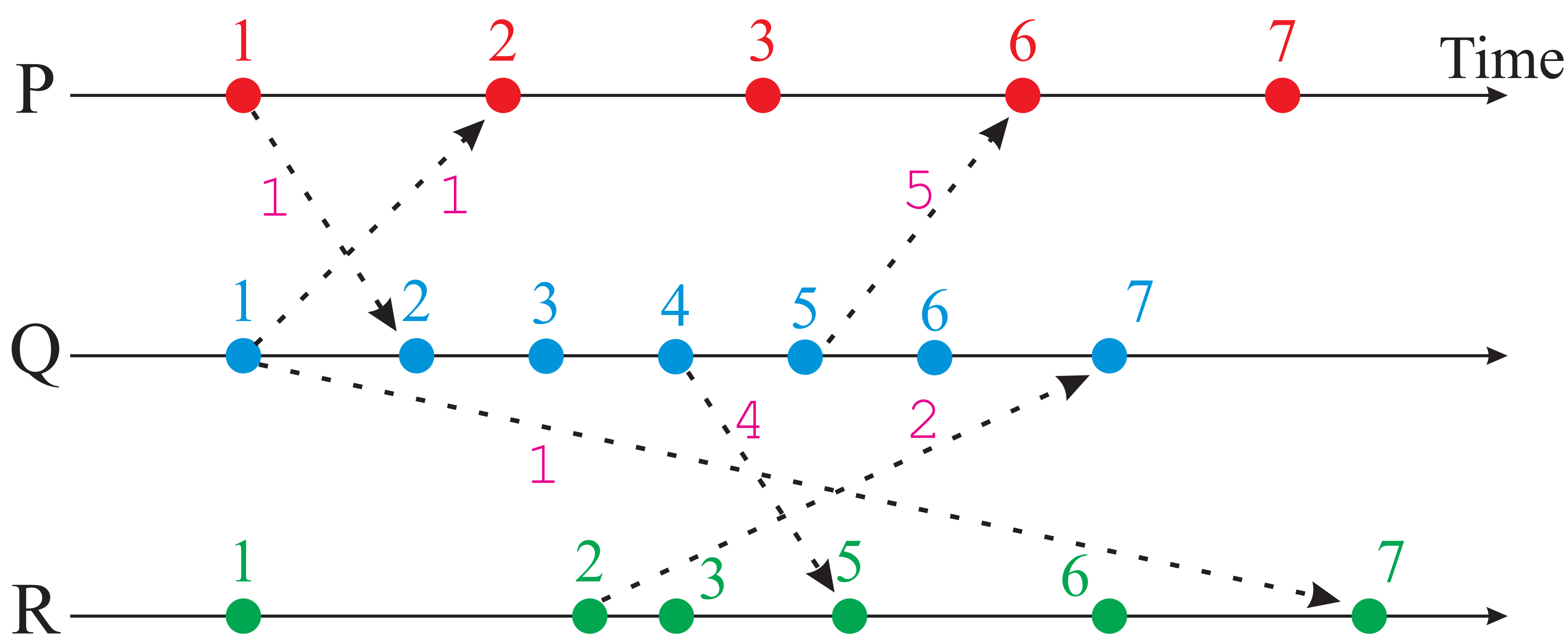


LC1: L_i is incremented before each event is issued at process p_i: $L_i := L_i + 1$

LC2: (a) When p_i sends a msg m, it piggybacks on m the value $t = L_i$

(b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies **LC1** before timestamping the event **receive(m)**

[Lamport Clocks] Example 3



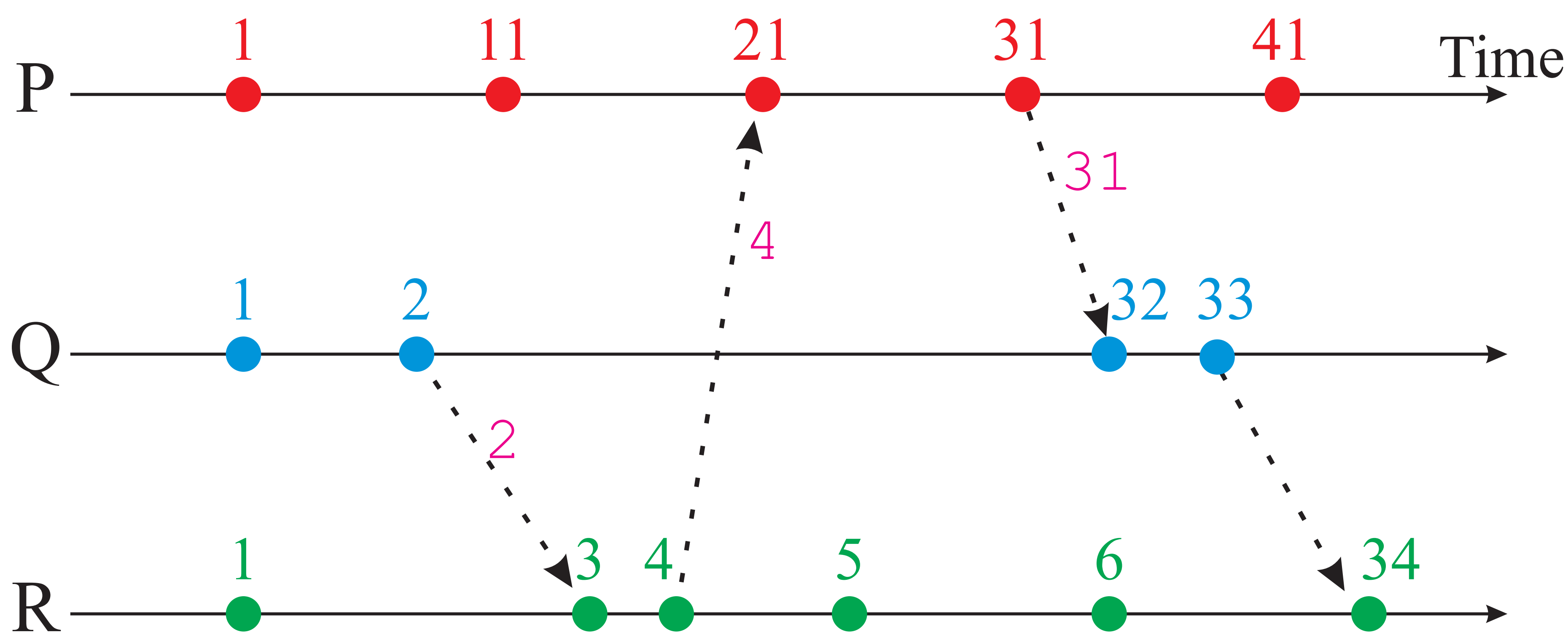
LC1: L_i is incremented before each event is issued at process p_i: $L_i := L_i + 1$

LC2: (a) When p_i sends a msg **m**, it piggybacks on **m** the value $t = L_i$

(b) On receiving **(m, t)**, a process p_j computes $L_j := \max(L_j, t)$ and then applies **LC1** before timestamping the event **receive(m)**

[Lamport Clocks] Example 4

LOCAL CLOCKS TEND TO RUN AS FAST AS THE FASTEST OF THEM



LC1: L_i is incremented before each event is issued at process p_i: $L_i := L_i + 1$

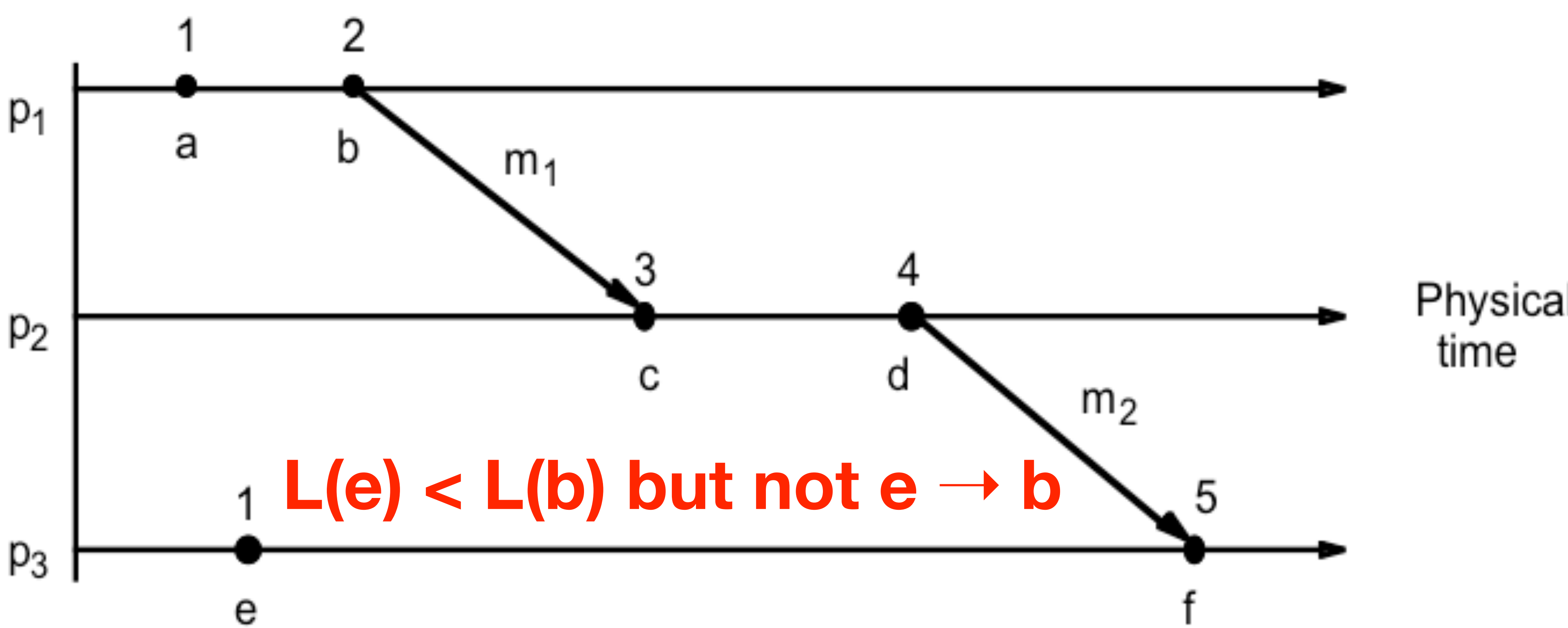
LC2: (a) When p_i sends a msg **m**, it piggybacks on **m** the value $t = L_i$

(b) On receiving **(m, t)**, a process p_j computes $L_j := \max(L_j, t)$ and then applies **LC1** before timestamping the event **receive(m)**

Shortcoming of Lamport Clocks (1)

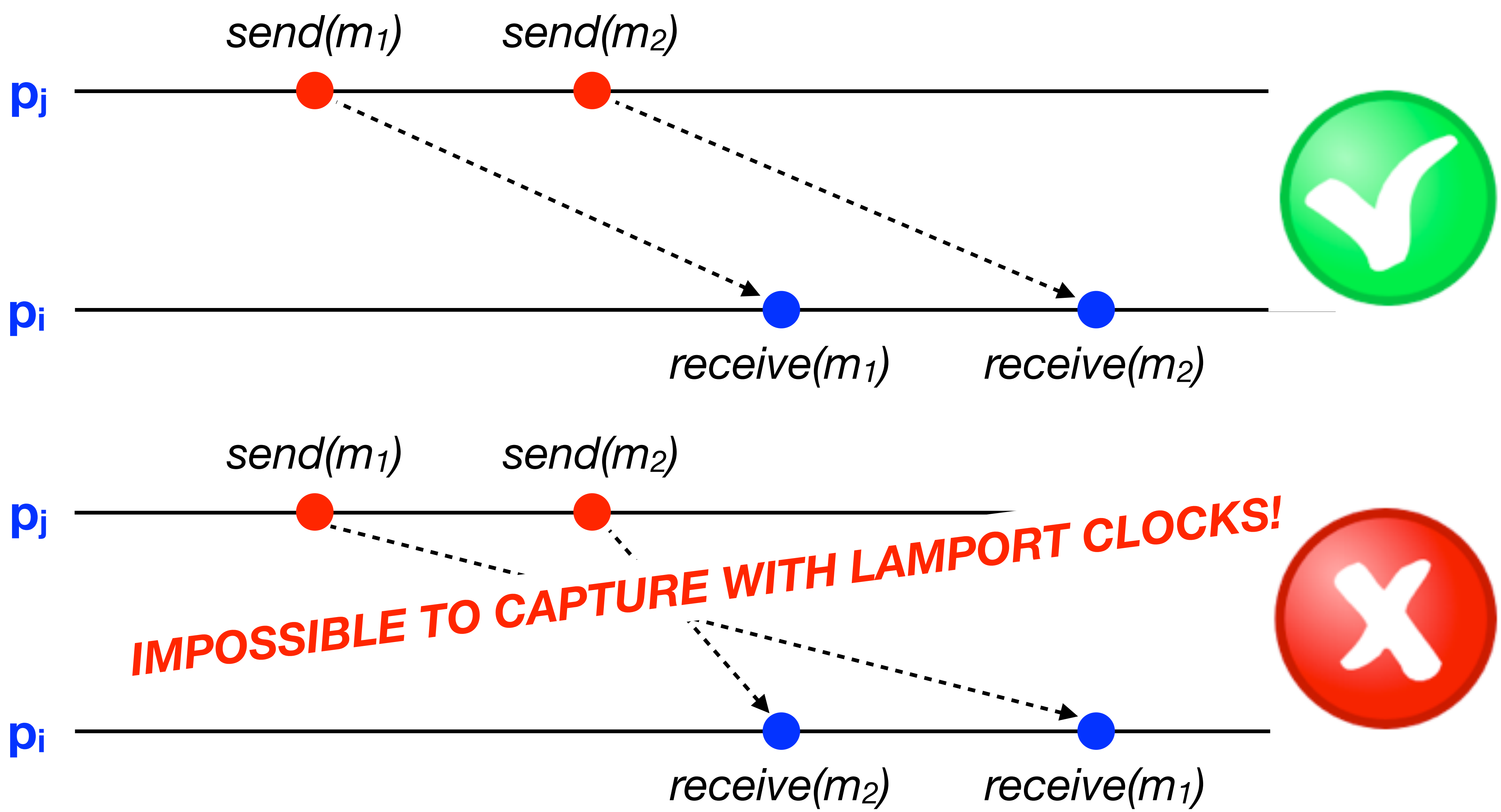
Clock consistency property: $\boxed{e \rightarrow e' \Rightarrow L(e) < L(e')}$

A significant problem with Lamport clocks is that if **$L(e) < L(e')$**, then we **cannot** infer that **$e \rightarrow e'$**

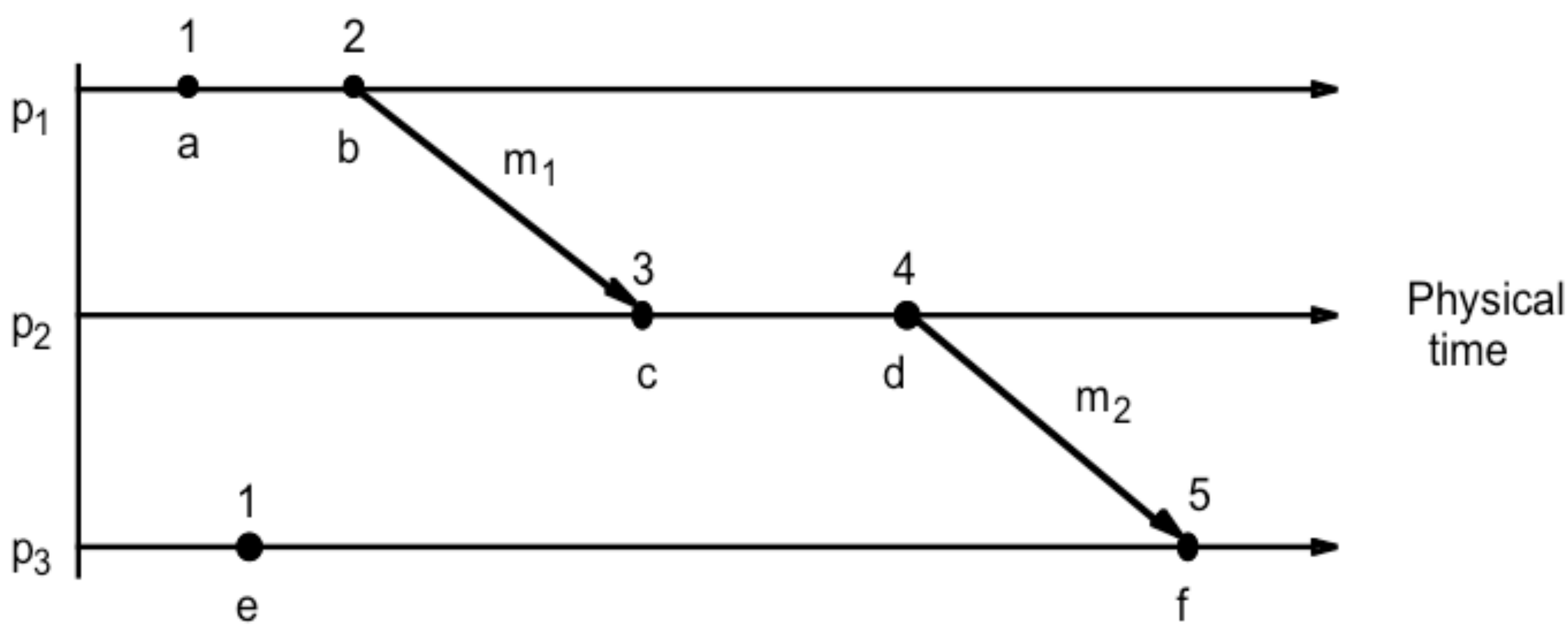


Shortcoming of Lamport Clocks (2)

- **Causal ordering of messages**: if **send(m_1) $\rightarrow$ send(m_2)** and **receive(m_1)** and **receive(m_2)** are on the same process **p_i**, then **receive(m_1) $\rightarrow_i$ receive(m_2)**



So... What Else Do We Need?



- Problem: *Lamport clocks describes global time by a single number, which is not enough and "hides" essential information.*
- Idea: *processes keep information on what they know about the other clocks in the system and use this information when sending a message*

Mattern and Fidge Vector Clocks

- Overcome the shortcoming of Lamport clocks

- Lamport clocks:

$$e \rightarrow f \textbf{ then } L(e) < L(f)$$

Clock consistency

- Vector clocks:

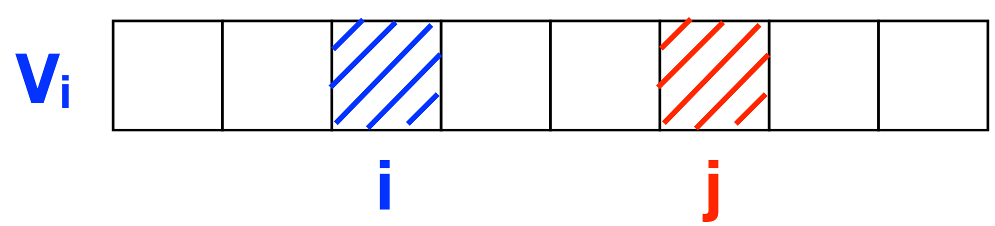
$$e \rightarrow f \textbf{ iff } V(e) < V(f)$$

Strong consistency

Vector Clocks

- A vector clock for a system of N processes: array of N integers

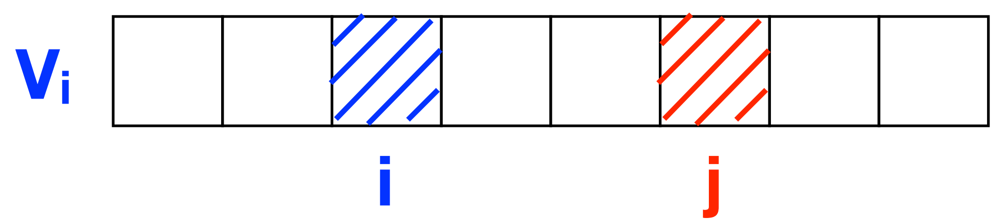
- Each process p_i keeps its own vector clock V_i, which it uses to timestamp local events



- Then $V_i[j]$ describes p_i's KNOWLEDGE of p_j's LOCAL LOGICAL CLOCK

- Example: if an event of p_2 is timestamped with (1, 1, 0) then p_2 knows that the value of the logical clocks are: 1 for p_1, 1 for p_2, 0 for p_3

Note that...



- $V_i[j]$ ($j \neq i$):
  - Latest clock value received by p_i from process p_j
  - Number of events that have occurred at p_j that p_i has potentially been affected by
    - *Process p_j may have timestamped more events by this point, but no information has flowed to p_i about them in messages yet!*

[Vector Clocks] Implementation Rules

VC1: Initially, $V_i[j] := 0$, for i, j = 1, 2, ..., N

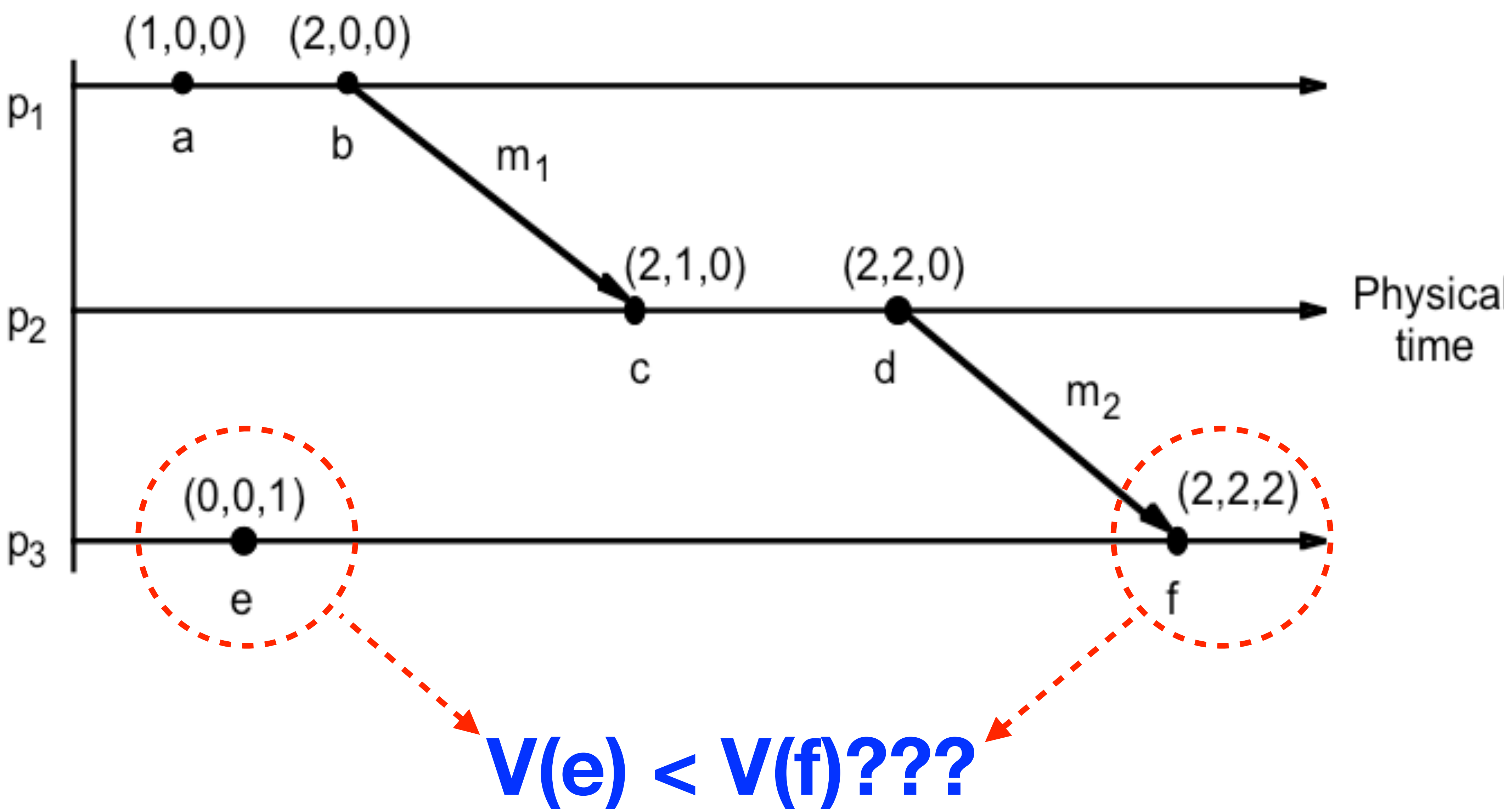
VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$

VC3: p_i includes the value $t = V_i$ in every message it sends

VC4: When p_i receives a timestamp t in a message

- p_i sets $V_i[j] := \max(V_i[j], t[j])$ for j = 1, 2, ..., N
- applies **VC2**
- timestamp the event **receive(m)**

[Vector Clocks] Example



Ordering on Vectors

- For vector clocks using rules VC1-4, it follows that

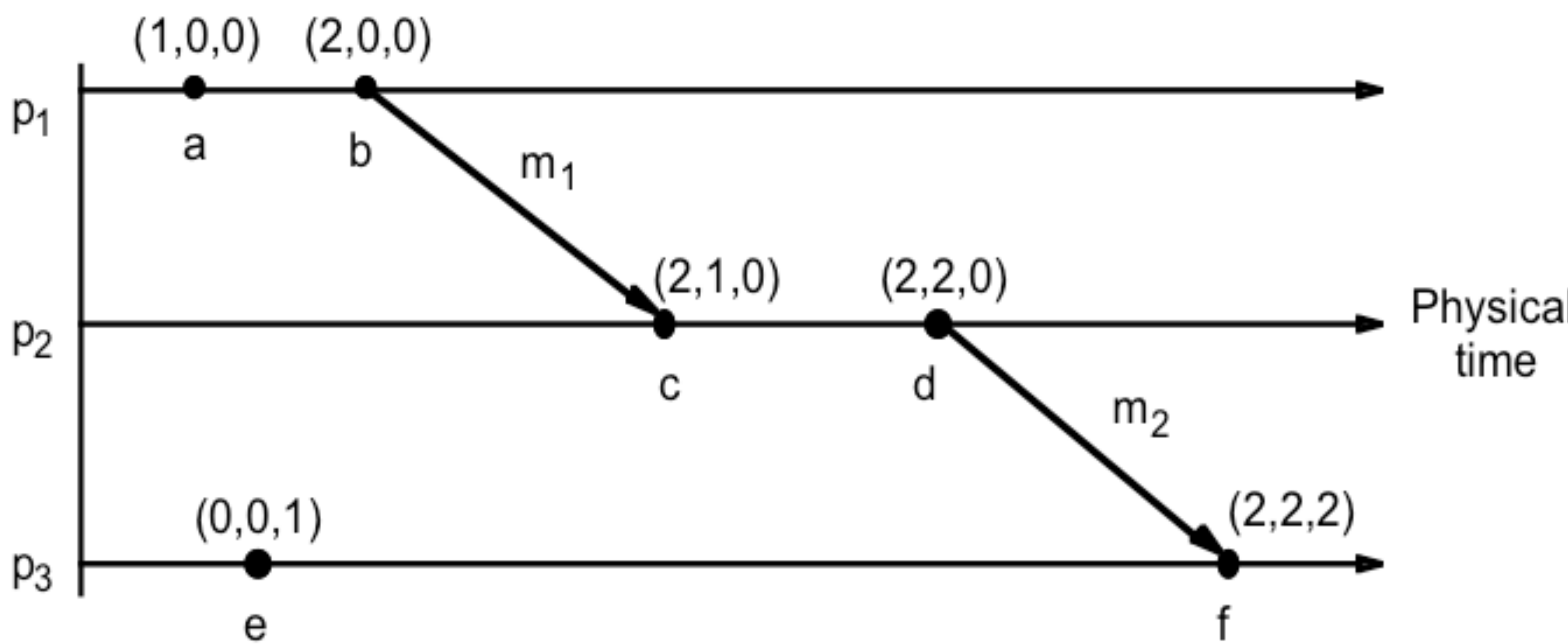
$$e \rightarrow e' \Leftrightarrow V(e) < V(e')$$

- Ordering relation (≤) on vectors:

$$V \leq V' \Leftrightarrow V[j] \leq V'[j] \text{ for } j = 1, 2, \ldots, N$$

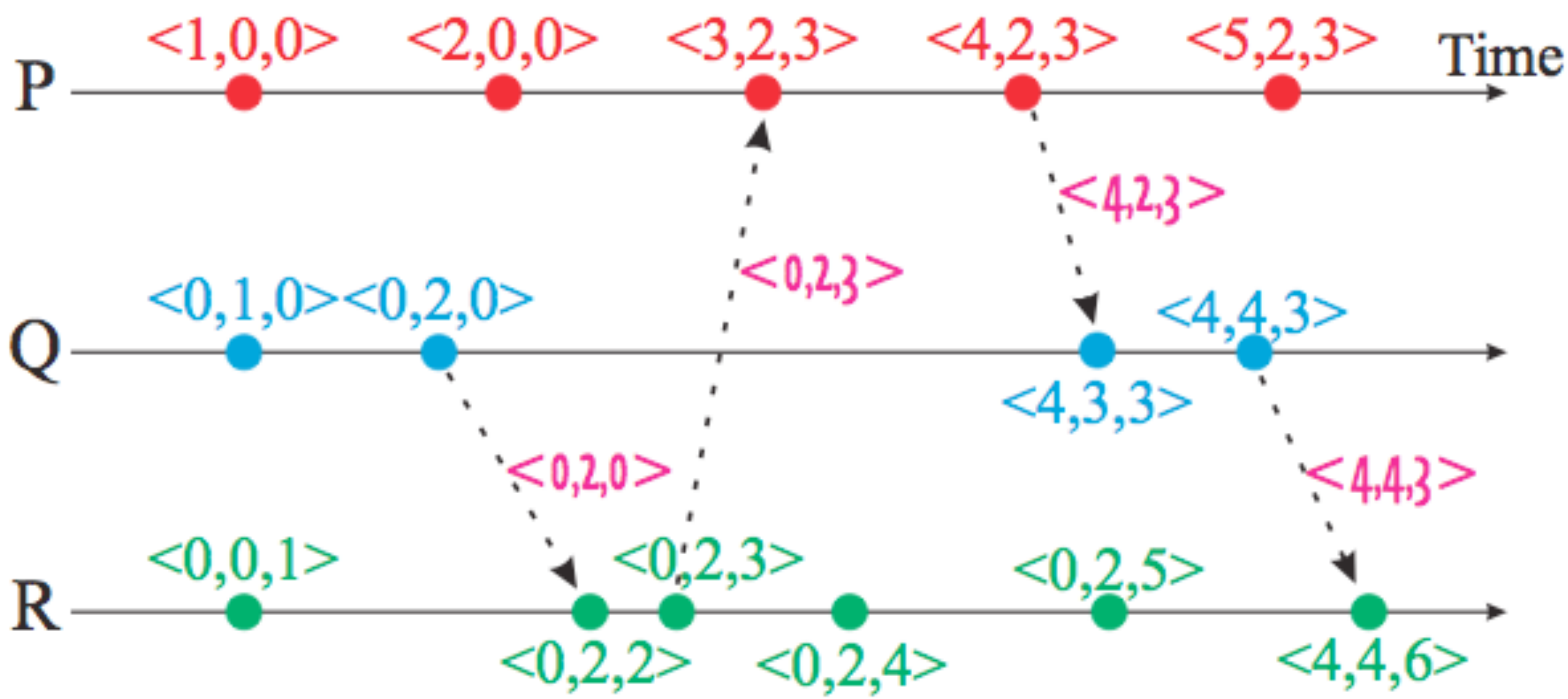
- In particular:
  - $V = V' \Leftrightarrow V[j] = V'[j] \text{ for } j = 1, 2, \ldots, N$
  - $V < V' \Leftrightarrow V \leq V' \wedge V \neq V'$
  - $V \parallel V' \Leftrightarrow \neg(V < V') \wedge \neg(V' < V)$

[Vector Clocks Ordering] Example



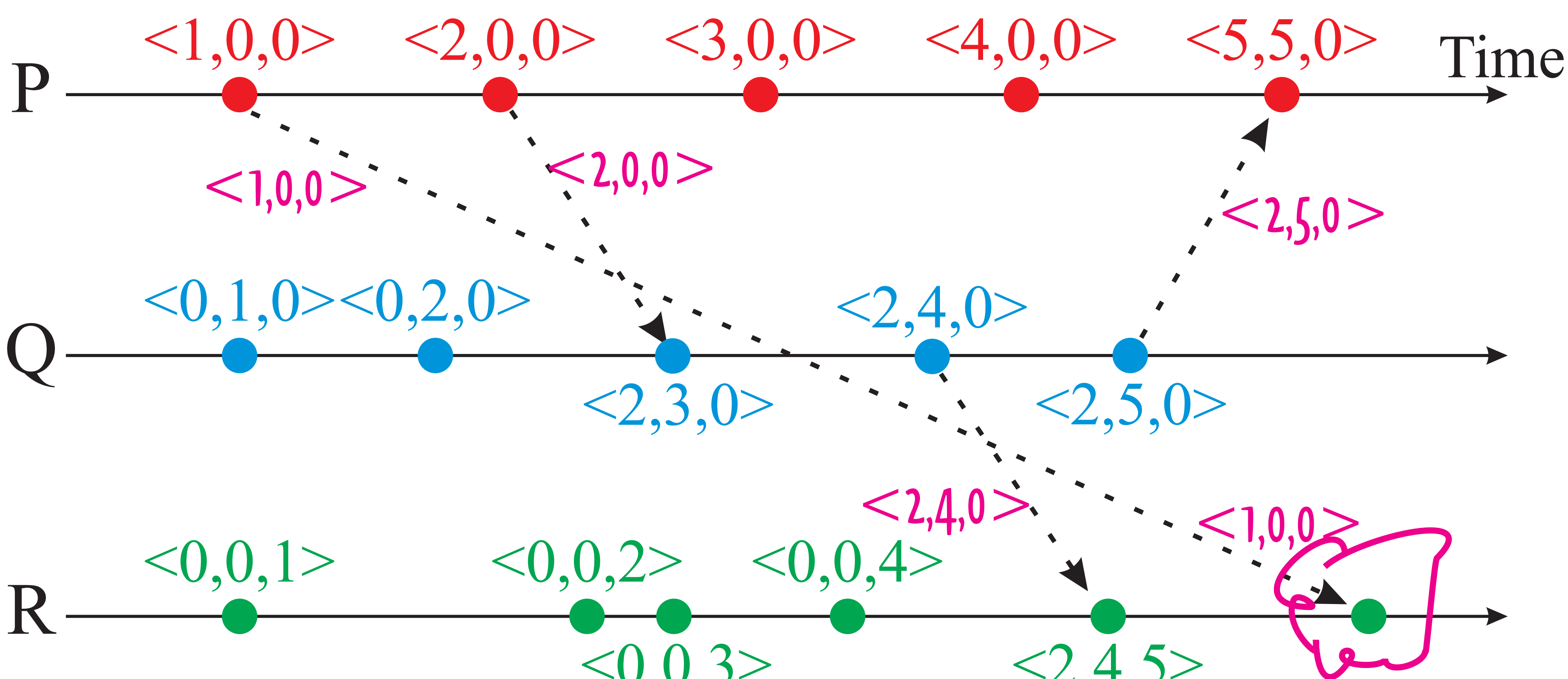
- **V(a) < V(f)**, reflecting the fact that **a → f**
- **c || e** because neither **V(c) ≤ V(e)** nor **V(e) ≤ V(c)**

[Vector Clocks] Example



[Vector Clocks] Violation of Causal Ordering

- Violation of causal ordering of messages occurs if msg M arrives with $V_M < V_i$.



- Here: $V_M[1] < V_R[1]$

Drawback of Vector Clocks

- The message overhead grows linearly with the number of processes in the system!!

- B. Charron-Bost. **Concerning the size of logical clocks in distributed systems**. *Information Processing Letters*, 39, pp. 11-16, 1991

  ==> Showed that if vector clocks have to satisfy the strong consistency property, then in general the vector timestamps must be at least of size **n**, the total number of processes

- Therefore, in general **the size of a vector timestamp is the number of processes involved in a distribute computation**

Efficient Implementation of Vector Clocks

Singhal-Kshemkalyani's Differential Technique

- M. Singhal and A. Kshemkalyani. **An efficient implementation of vector clocks**. *Information Processing Letters*, 43, pp. 47-52, 1992

- Observation
  When the number of processes is large and only few of them interact, then between successive msg sends to the same processes, only a few entries of the vector clock at the sender process are likely to change

- Solution
  When a process p_i sends a message to a process p_j, it piggybacks only those entries of its vector clock that differ since the last message sent to p_j

- Assumption
  Communication channels follow FIFO discipline for message delivery

Singhal-Kshemkalyani's Differential Technique

- The technique works as follows:
  - if entries $i_1, i_2, \ldots, i_m$, $m \le n$, of the vector clock at p_i have changed to $v_1, v_2, \ldots, v_m$, respectively, since the last message sent to p_j

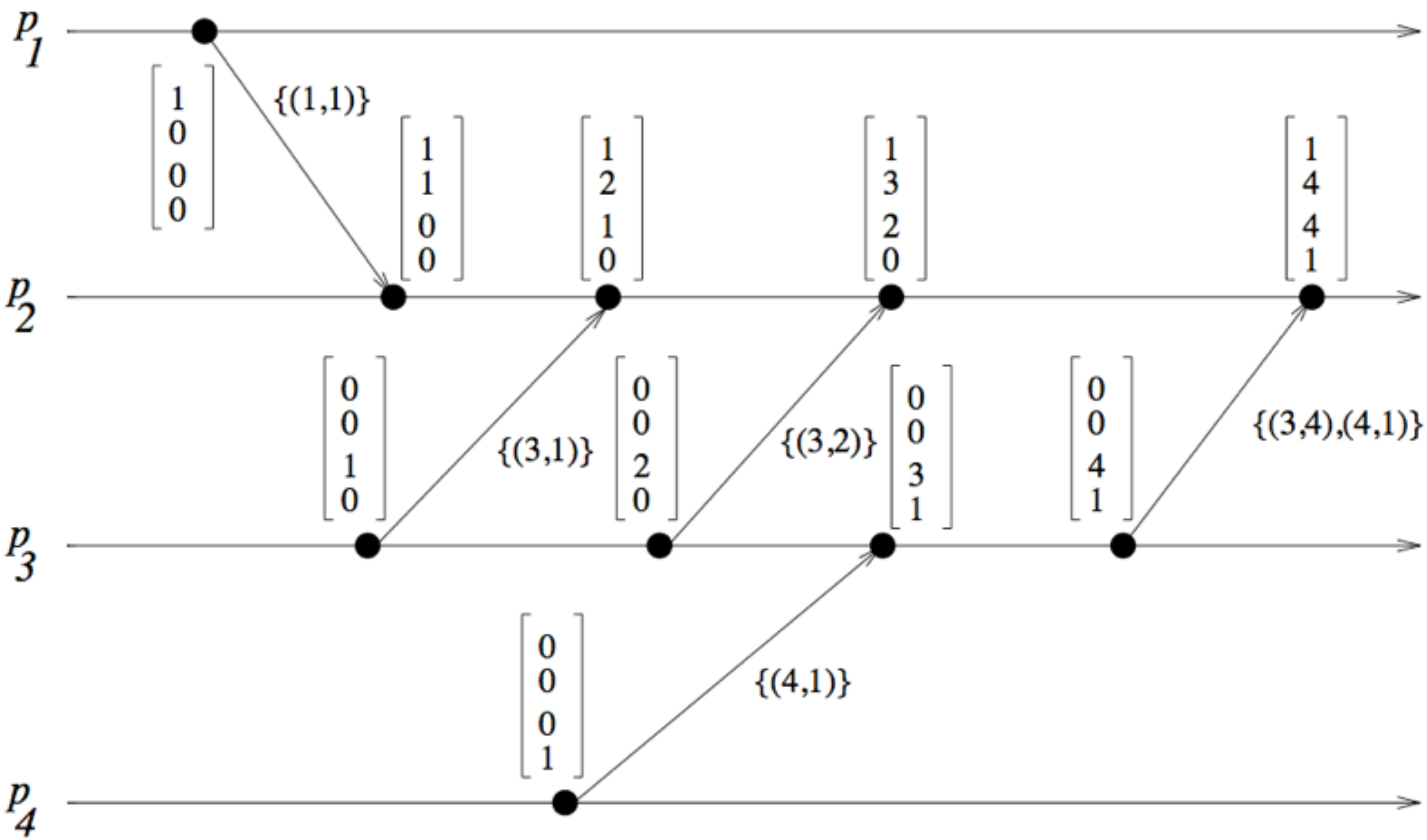
    then process p_i piggybacks a timestamp of the form

    $$\{(i_1, v_1), (i_2, v_2), \ldots, (i_m, v_m)\}$$

    to the next message to p_j
  - when p_j receives this message, it updates its vector clock V_j as follows:

    $$V_j[i_k] = \max(V_j[i_k], v_k) \text{ for } k = 1, 2, \ldots, m$$

Example: Vector Clocks Progress in S-K Technique



Analysis

- Worst case (**m = n**): every element of the vector clock has been updated at p_i since the last message to p_j

  ==> next msg from p_i to p_j will need to carry the entire vector of size **n**

- Average case (**m < n**): the size of the timestamp on a msg will be less than **n**

- Direct implementation: requires each process to remember the vector timestamp (of size at most **n**) in the message last sent to every other process

  ==> implementation will result in **O(n^2) storage overhead at each process**

Can we do better?

How to Cut Down the Storage Overhead?

Implementation of Singhal-Kshemkalyani's Idea

- Process p_i maintains the following **two additional vectors**:
  - **$LS_i[1 \ldots n]$** ("Last Sent")
    $LS_i[j]$: the value of **$V_i[i]$** when process p_i last sent a message to p_j
  - **$LU_i[1 \ldots n]$** ("Last Update")
    $LU_i[j]$: the value of **$V_i[i]$** when process p_i last updated the entry **$V_i[j]$**
- N.B.:
  - **$LU_i[i]$** = **$V_i[i]$** at all times
  - **$LS_i[j]$** needs to be updated only when p_i sends a message to p_j
  - **$LU_i[j]$** needs to be updated only when the receipt of a message causes p_i to update entry **$V_i[j]$**

Implementation of Singhal-Kshemkalyani's Idea

- Key condition: $LS_i[j] < LU_i[k] \quad k = 1, ..., n$

- When p_i sends a message to p_j, it sends only a set of tuples

$$\{(k, V_i[k]) \mid LS_i[j] < LU_i[k]\} \quad k = 1, ..., n$$

as the vector timestamp to p_j (instead of sending a vector of **n** entries in a message)

Exercise

- Singhal and Kshemkalyani's technique cuts down the storage overhead at each process from **$O(n^2)$** to ...

- Explain why.