CS 425 / ECE 428

Basics

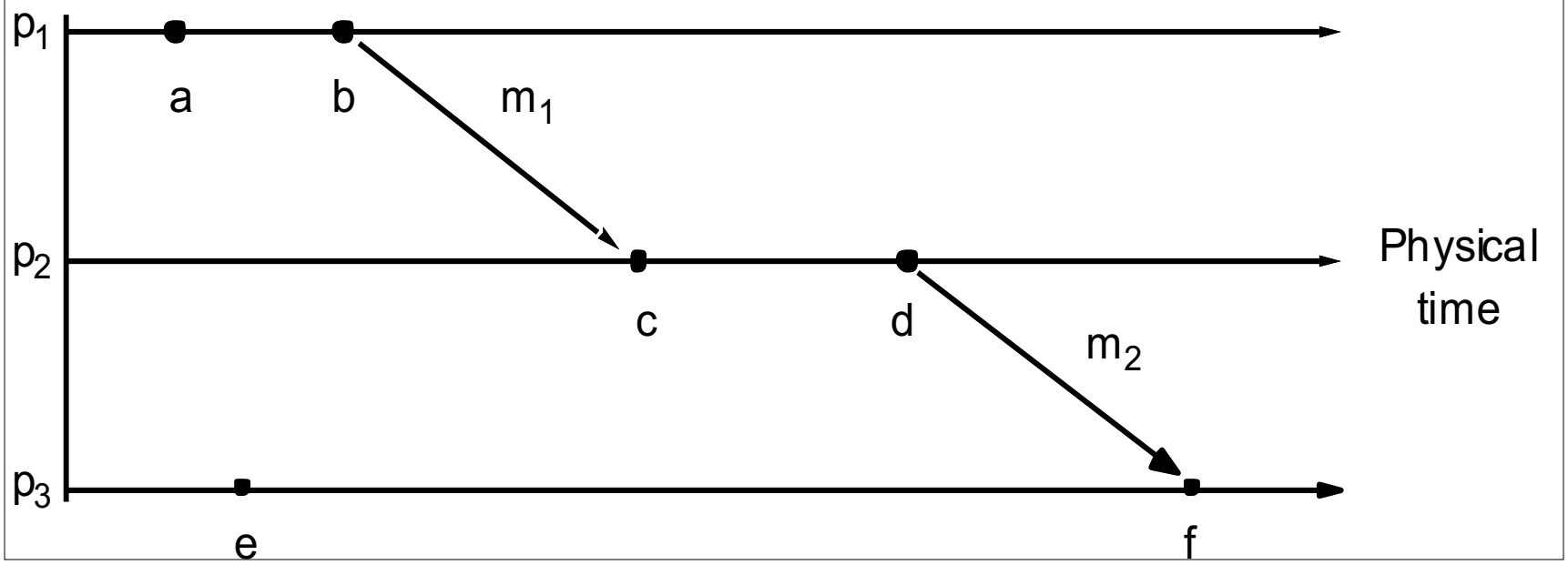
- A Distributed System (DS) consists of a number of *processes.*
- Each process has a state (values of variables).
- State of a may change as a result of a computation step or communication step
- Processes may communicate by message-passing or via shared memory
  - Message-passing: Send & Receive message
  - Shared memory: Write & Read shared memory

Basics

- An event is an abstraction to represent steps performed by a process
- Each process has a local clock, which may drift
- *Synchronous* systems: Each action is performed within bounded amount of time (e.g., a computation step, message delivery over the network, memory operation)
- *Asynchronous* systems: Unbounded delay may be incurred. The processes may perform computation at arbitrarily different "rates". Messages may be delayed for unbounded intervals.

Event types in message-passing systems

- **Compute event**
- **Send event**
- **Receive event**



Ordering of Events

- **Real time at which events occur can be used to determine the order of the events**
- **Can we infer event ordering in absence of real-time?**
- **Is real-time sufficient to infer causal dependency between events?**
  - **Is it possible for event a to have affected event b ?**

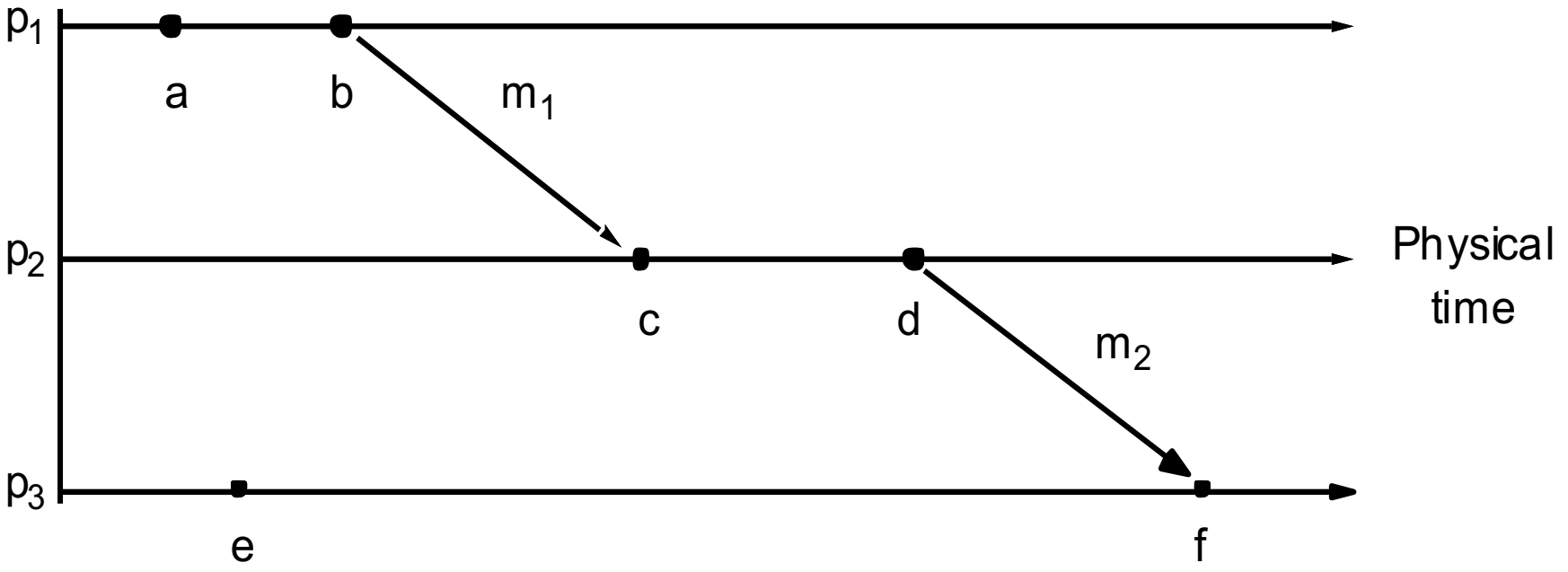
Happened-Before Relation

- **"happened-before" relation defined without using real-time**

- **Notation: a → b a happened-before b**

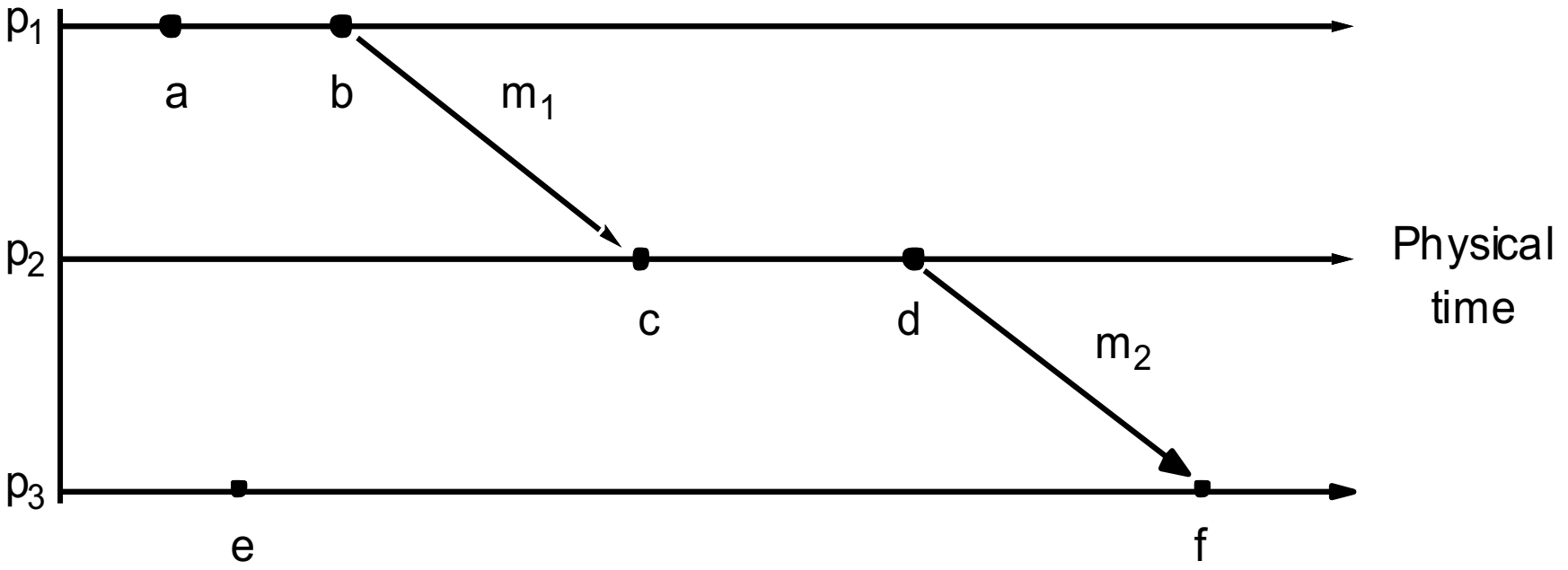
- **Three rules for *message-passing systems*:**
  - **Events on the same process: Events a and b occur at the SAME process, and a occurs before b, then a → b**
  - **If a = Send(m) event and b = Receive(m) events then a → b**
  - **Transitivity: if a → b and b → c then a → c**

Events Occurring at Three Processes



a → b Why?

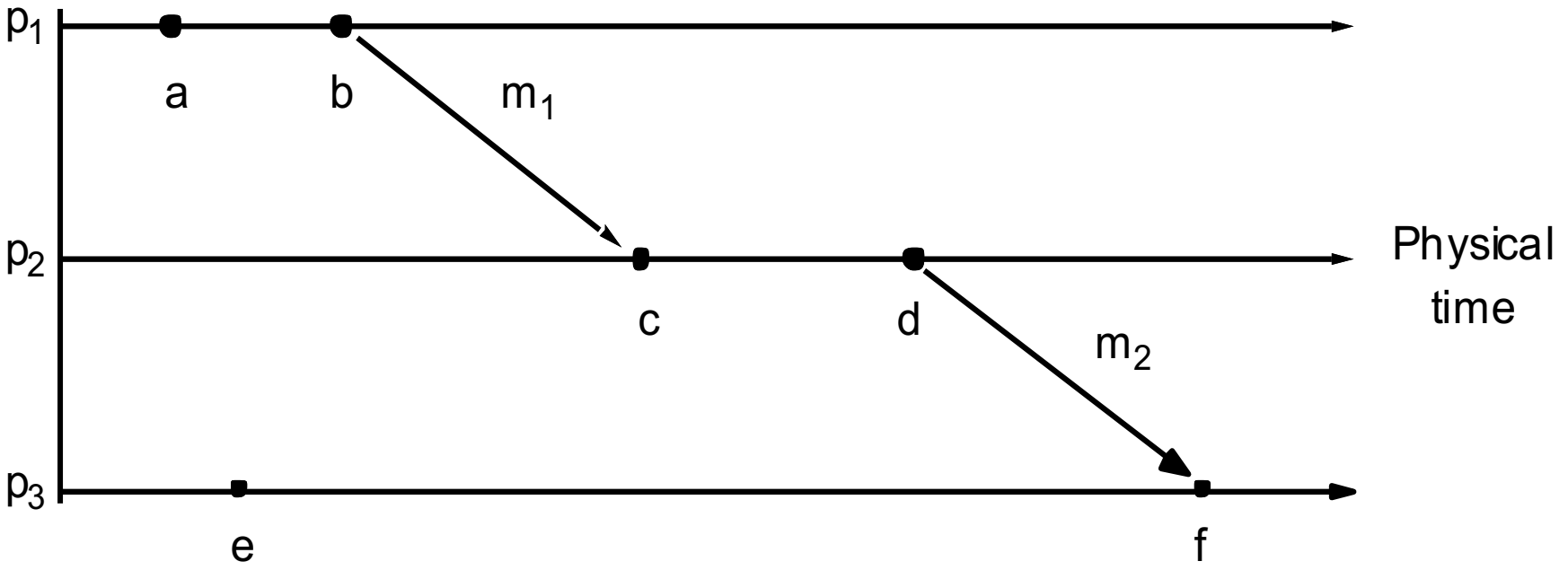
Events Occurring at Three Processes



a → b Why?

b → c Why?

Events Occurring at Three Processes

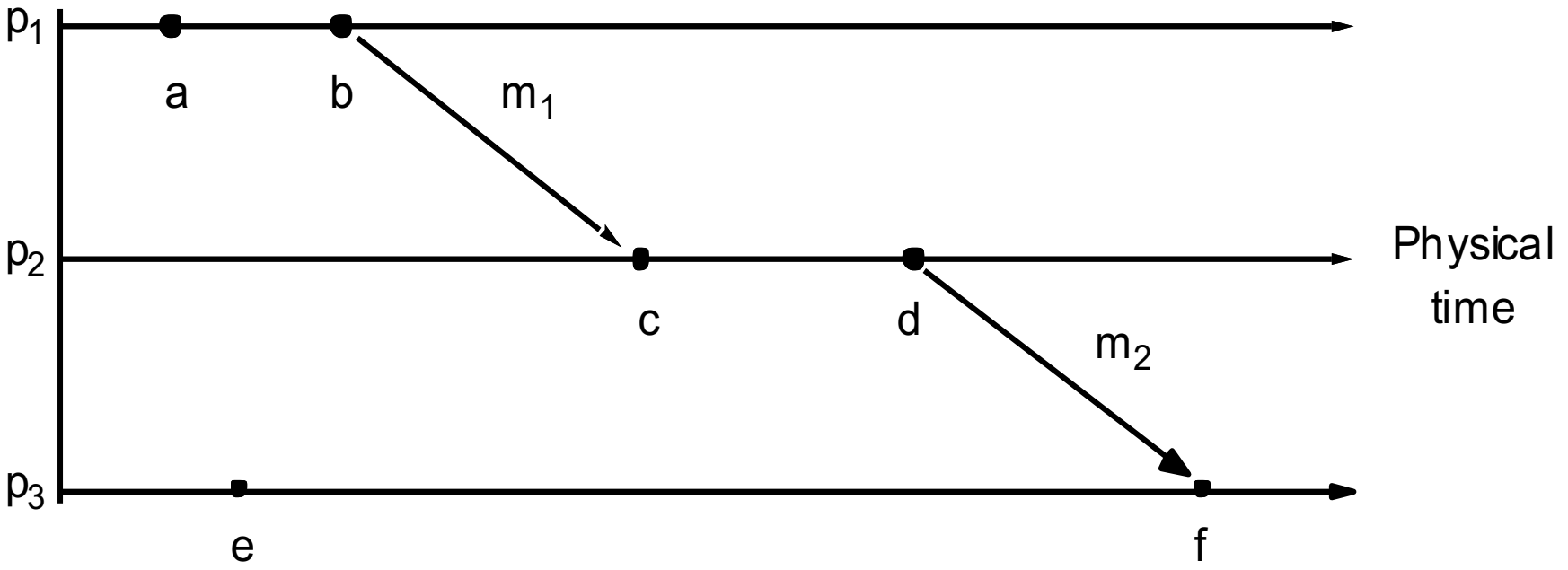


a → b Why?

b → c Why?

a → f Why?

Events Occurring at Three Processes



a → b Why?

b → c Why?

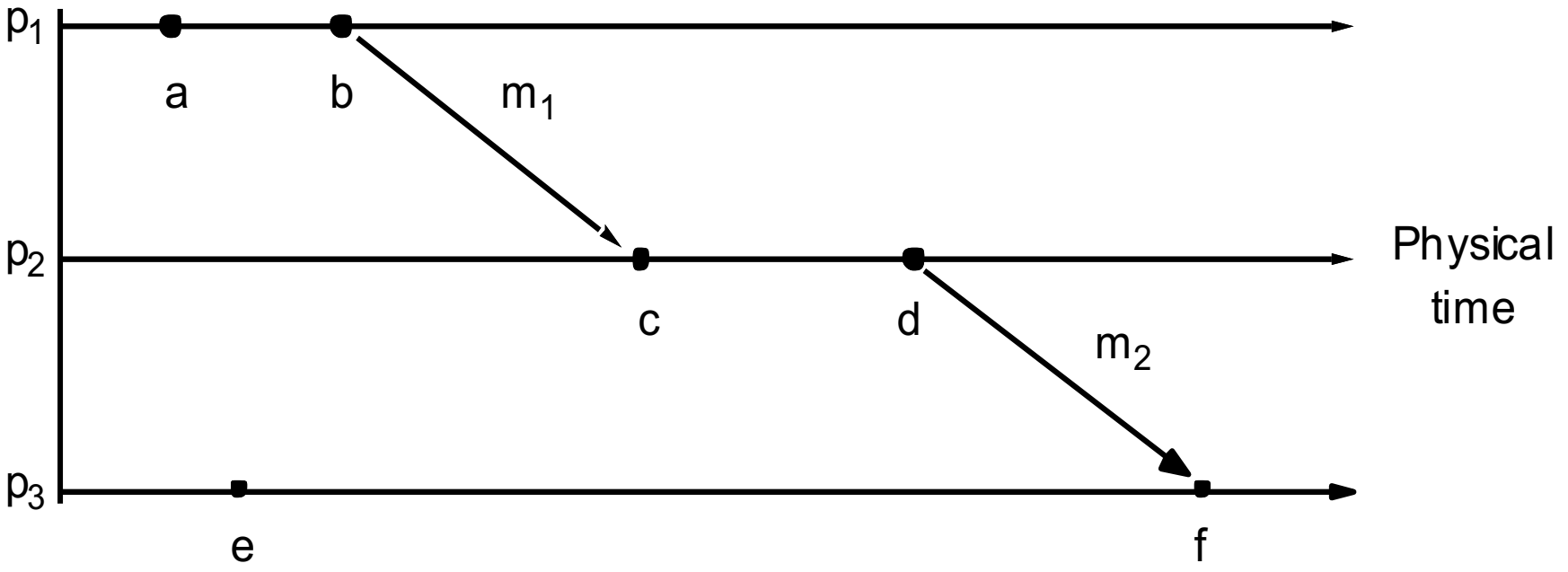
a → f Why?

a → e ????

Happened-Before

- If $a \not\rightarrow b$ and $b \not\rightarrow a$ then a and b are (logically) concurrent events
- Notation: $a \parallel b$
- Concurrent events are not causally related

Identify concurrent events



Ordering of events

- **Need some mechanism to track the ordering of events in the program without using real-time**
- **Logical timestamps useful for this**
- **Lamport introduced the logical clock mechanism**

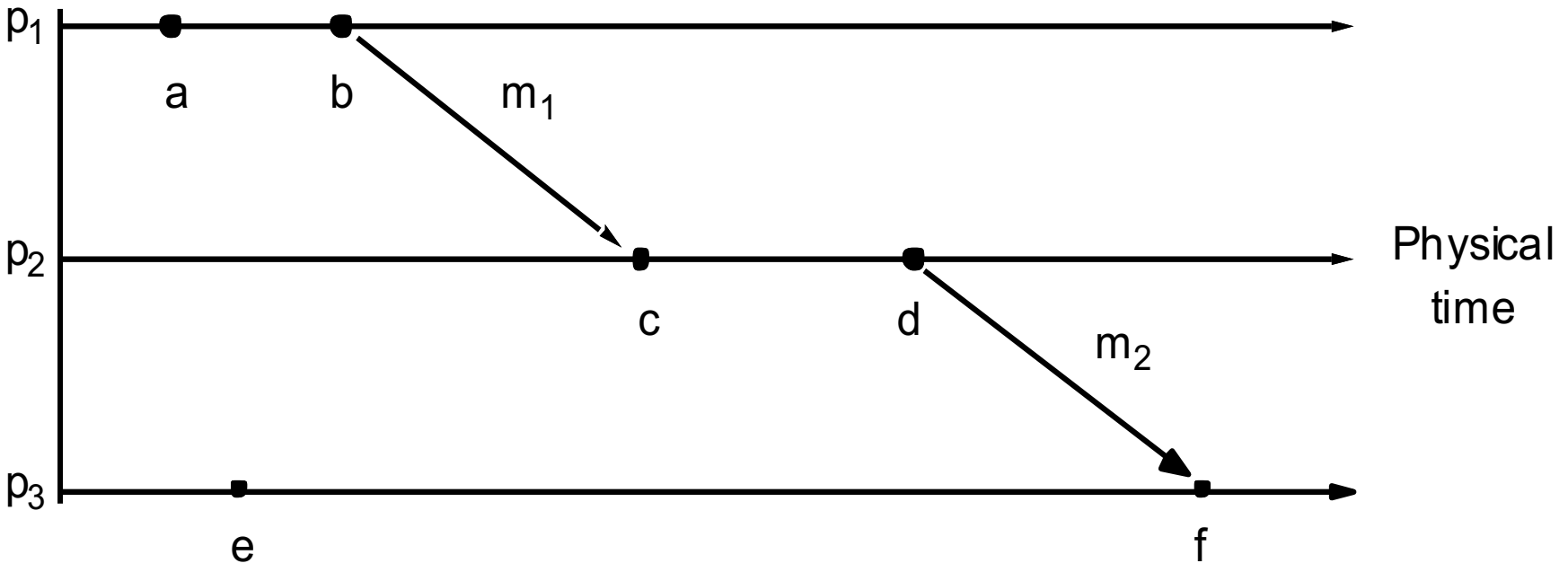
Logical Clock (Lamport Clock)

- **Each process Pi maintains a logical clock Li**
- **Intialialize Li = 0 at the start of the process**
- **We will describe how timestamps are updated for each type of event separately**

Logical Clock (Lamport Clock)

- **Compute event at process Pi:**
  - **Increment Li by 1**
  - **New value of Li is the timestamp of the compute event**

Lamport Timestamps

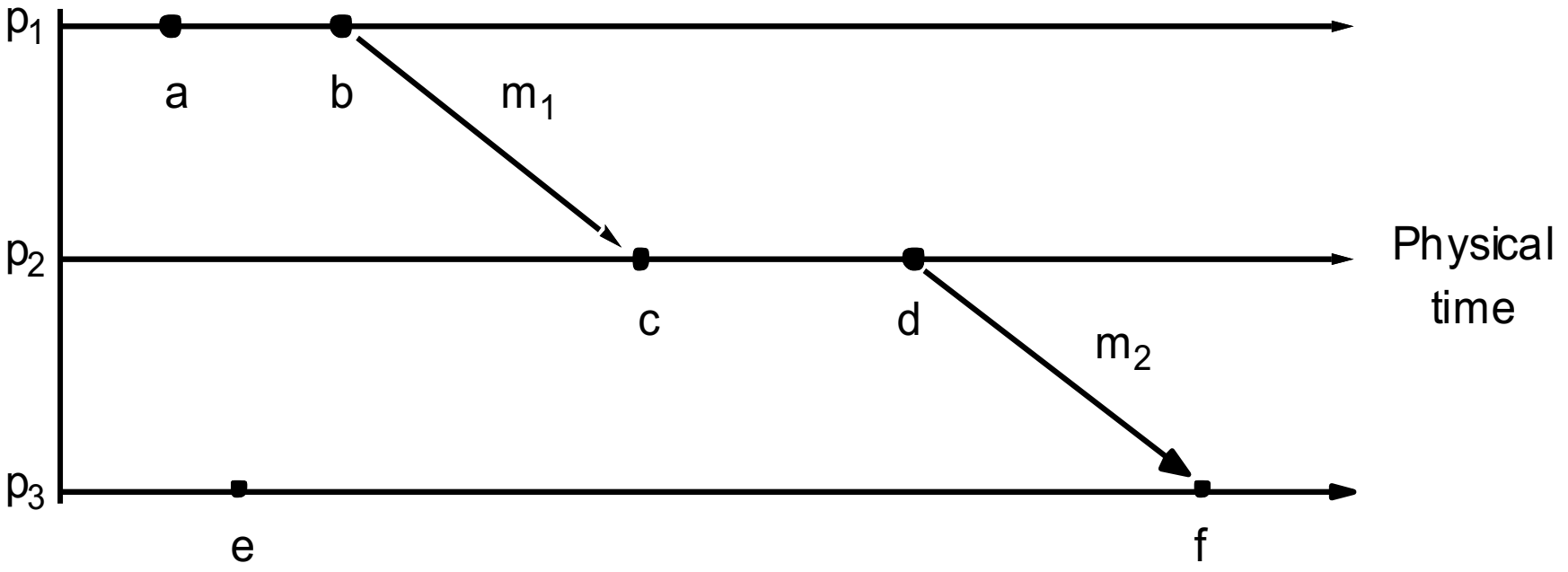


Each event is assigned a scalar timestamps

Logical Clock (Lamport Clock)

- **Compute event at process Pi:**
  - **Increment Li by 1**
  - **New value of Li is the timestamp of the compute event**
- **Send event at process Pi: Consider e = send(m)**
  - **Increment Li by 1**
  - **New value of Li is the timestamp of send event e**
  - **Piggyback the timestamp of e with message m**

Lamport Timestamps

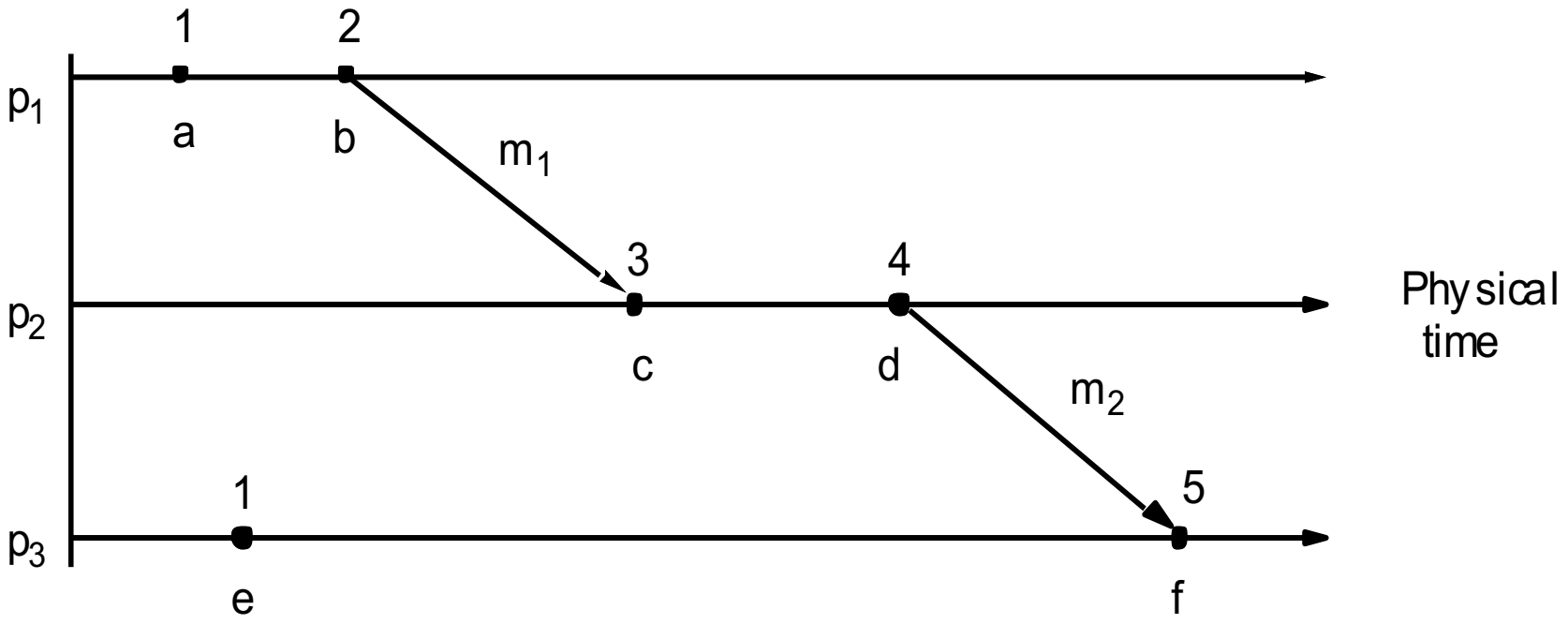


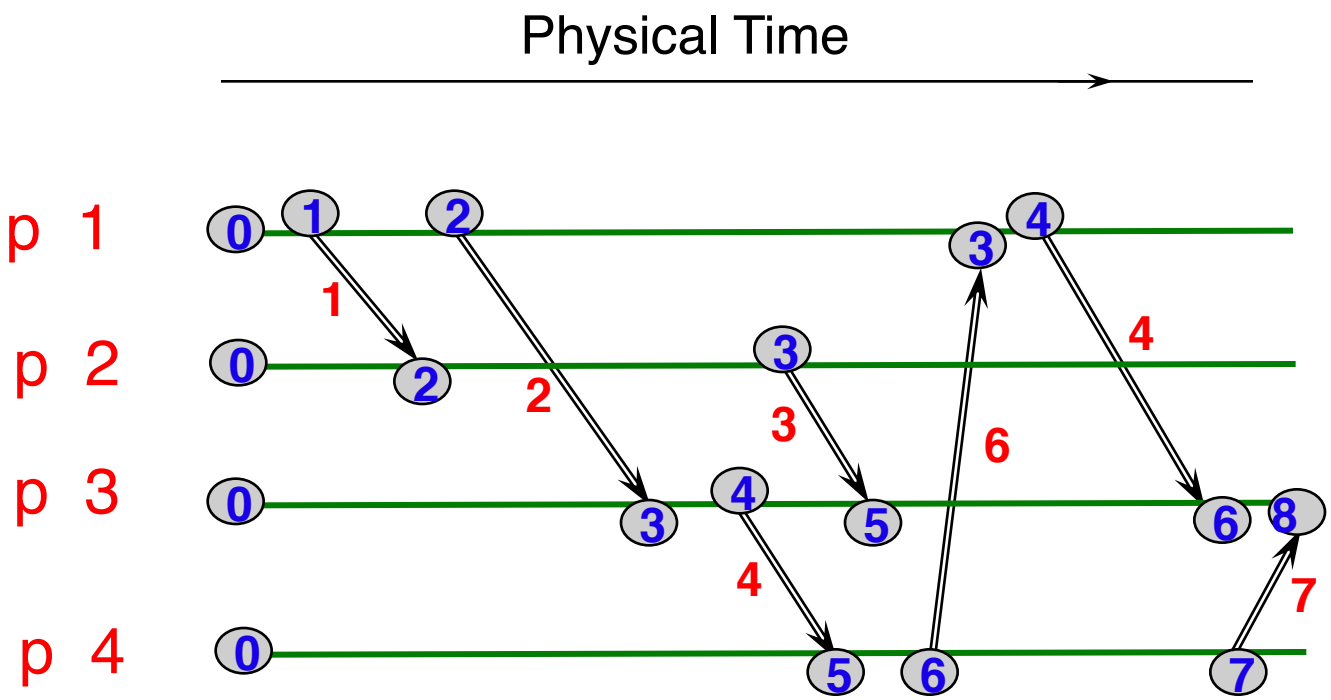
Each event is assigned a scalar timestamps

Logical Clock (Lamport Clock)

- **Compute event at process Pi:**
  - **Increment Li by 1**
  - **New value of Li is the timestamp of the compute event**
- **Send event at process Pi: Consider e = send(m)**
  - **Increment Li by 1**
  - **New value of Li is the timestamp of send event e**
  - **Piggyback the timestamp of e with message m**
- **Receive event at process Pi: Suppose (m,t) where m is a message, and t is the piggybacked timestamp, is received at event e at Pi**
  - **Update Li as Li := max(Li, t)+1**

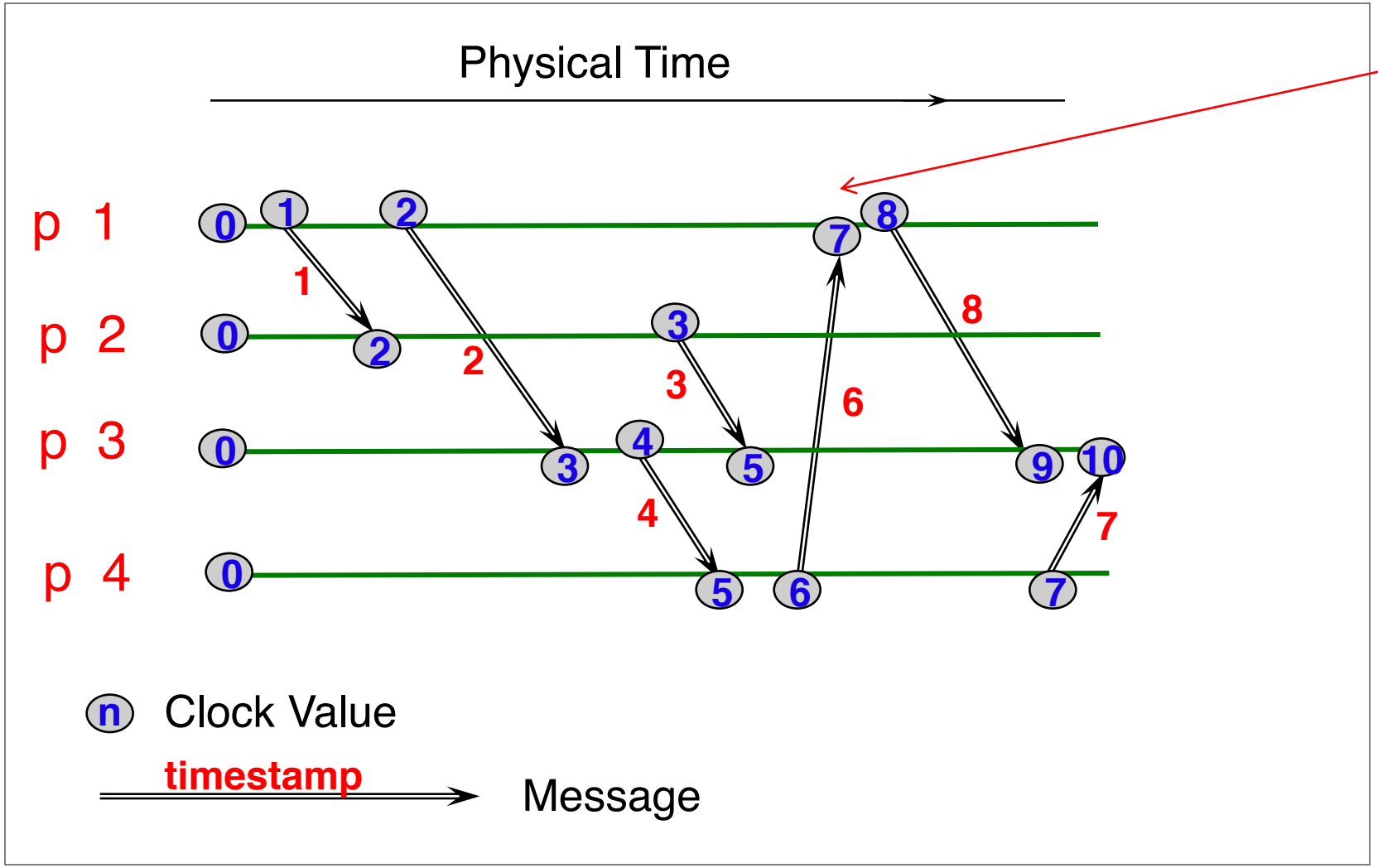
Lamport Timestamps





Find the Mistake

Corrected Example: Lamport Logical Time

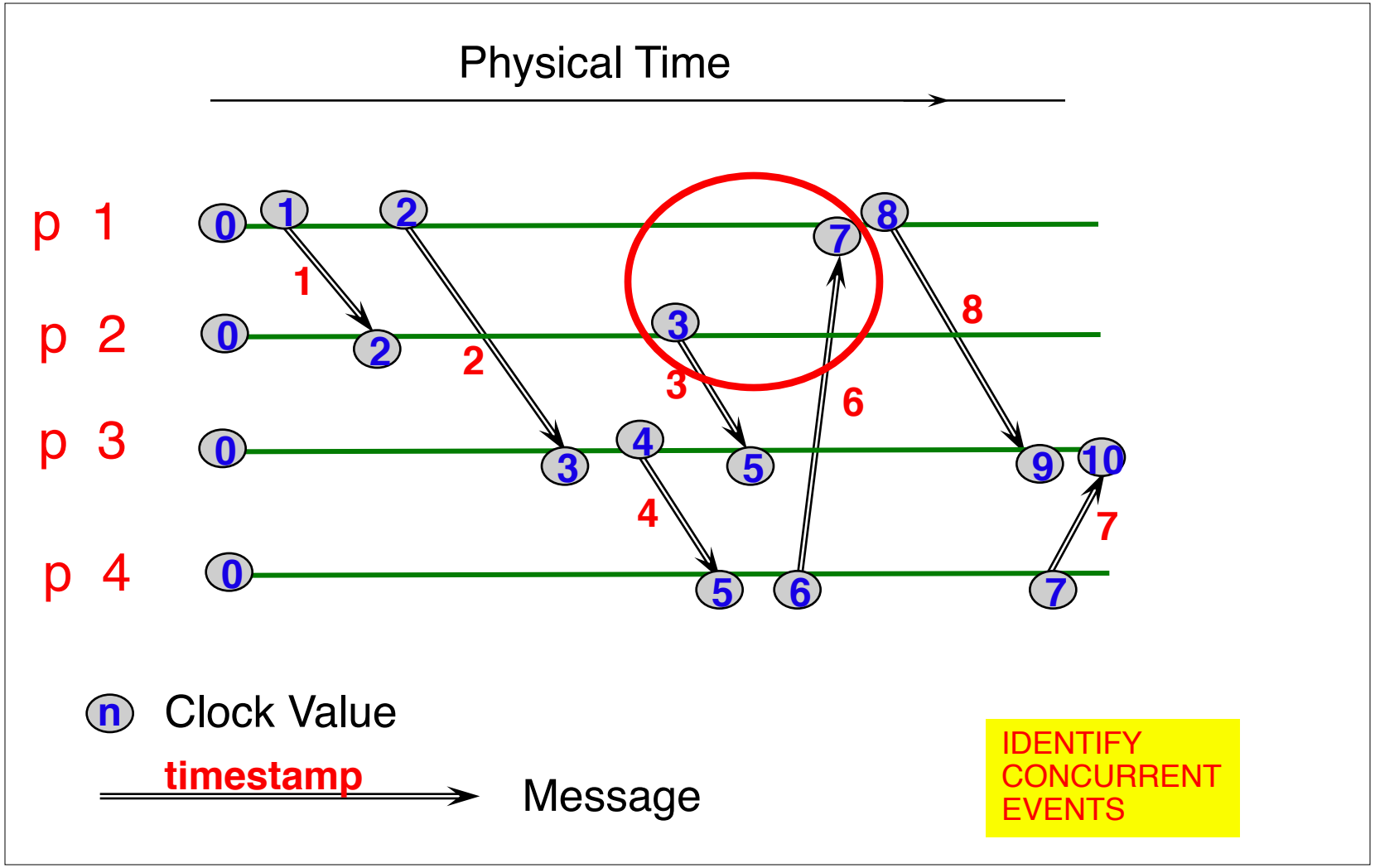


Properties of Lamport Clock

L(a) denotes Lamport or Logical timestamp of event a

- **If a → b then L(a) < L(b)**
- **However, L(b) > L(a) does not imply that a → b**

Properties of Lamport Clock



Vector Logical Clocks

- **With Lamport Logical Timestamp**

  e → f ⇒ timestamp(e) < timestamp (f), but

  timestamp(e) < timestamp (f) ⇒ {e → f} OR {e and f concurrent}

- **Vector Logical time addresses this issue:**

Vector Logical Clocks

- With Lamport Logical Timestamp

  e → f ⇒ timestamp(e) < timestamp (f), but

  timestamp(e) < timestamp (f) ⇒ {e → f} OR {e and f concurrent}

- Vector Logical time addresses this issue:
  - Each process maintains a vector clock,
    length = number of processes

Vector Logical Clocks

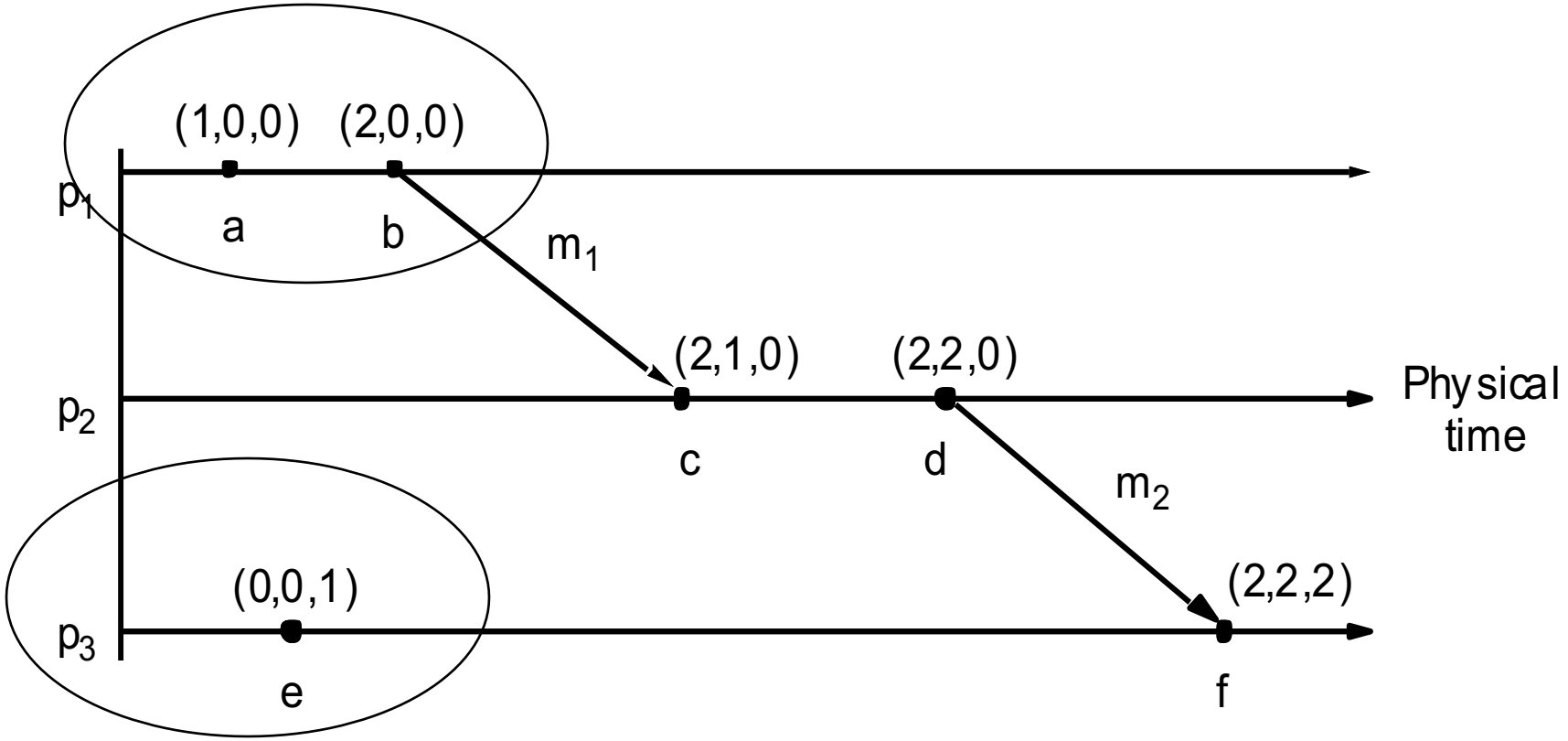
- With Lamport Logical Timestamp

  e → f ⇒ timestamp(e) < timestamp (f), but

  timestamp(e) < timestamp (f) ⇒ {e → f} OR {e and f concurrent}

- Vector Logical time addresses this issue:
  - Each process maintains a vector clock V_i, length = number of processes
  - At each event, process i increments i^{th} element of vector V_i
    - ➔ The new V_i is the timestamp of the event

Vector Timestamps



Vector clocks initialized to all-0 vector at each process

Initial value: (0,0,0)

Vector Logical Clocks

- With Lamport Logical Timestamp

  e → f ⇒ timestamp(e) < timestamp (f), but

  timestamp(e) < timestamp (f) ⇒ {e → f} OR {e and f concurrent}

- Vector Logical time addresses this issue:
  - Each process maintains a vector clock,
    length = number of processes
  - At each event, process i increments i^{th} element of vector V_i

    → The new V_i is the timestamp of the event
  - A message carries the Send event's vector timestamp

Vector Logical Clocks

- With Lamport Logical Timestamp

  e → f ⇒ timestamp(e) < timestamp (f), but

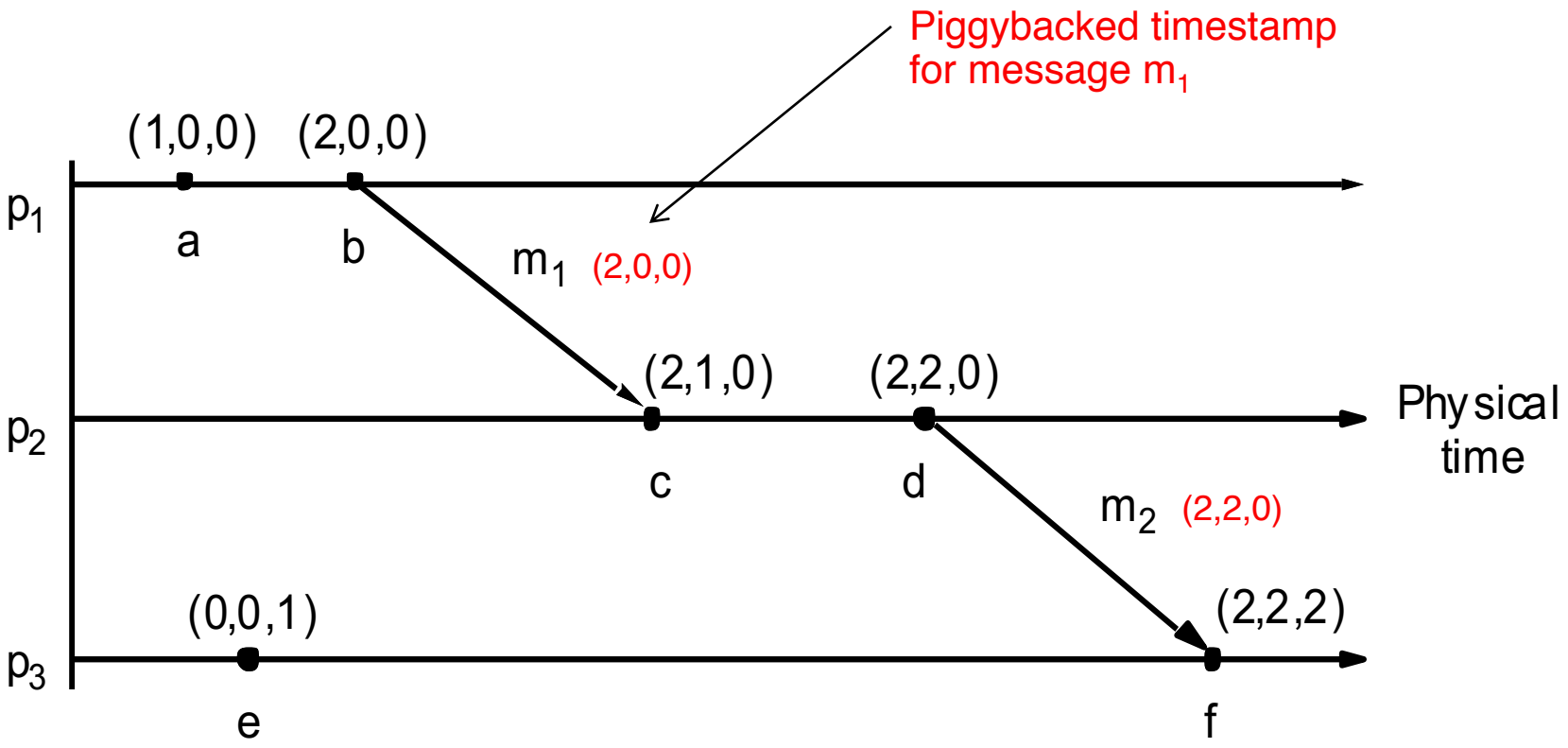
  timestamp(e) < timestamp (f) ⇒ {e → f} OR {e and f concurrent}

- Vector Logical time addresses this issue:
  - Each process maintains a vector clock, length = number of processes
  - At each event, process i increments i^{th} element of vector V_i

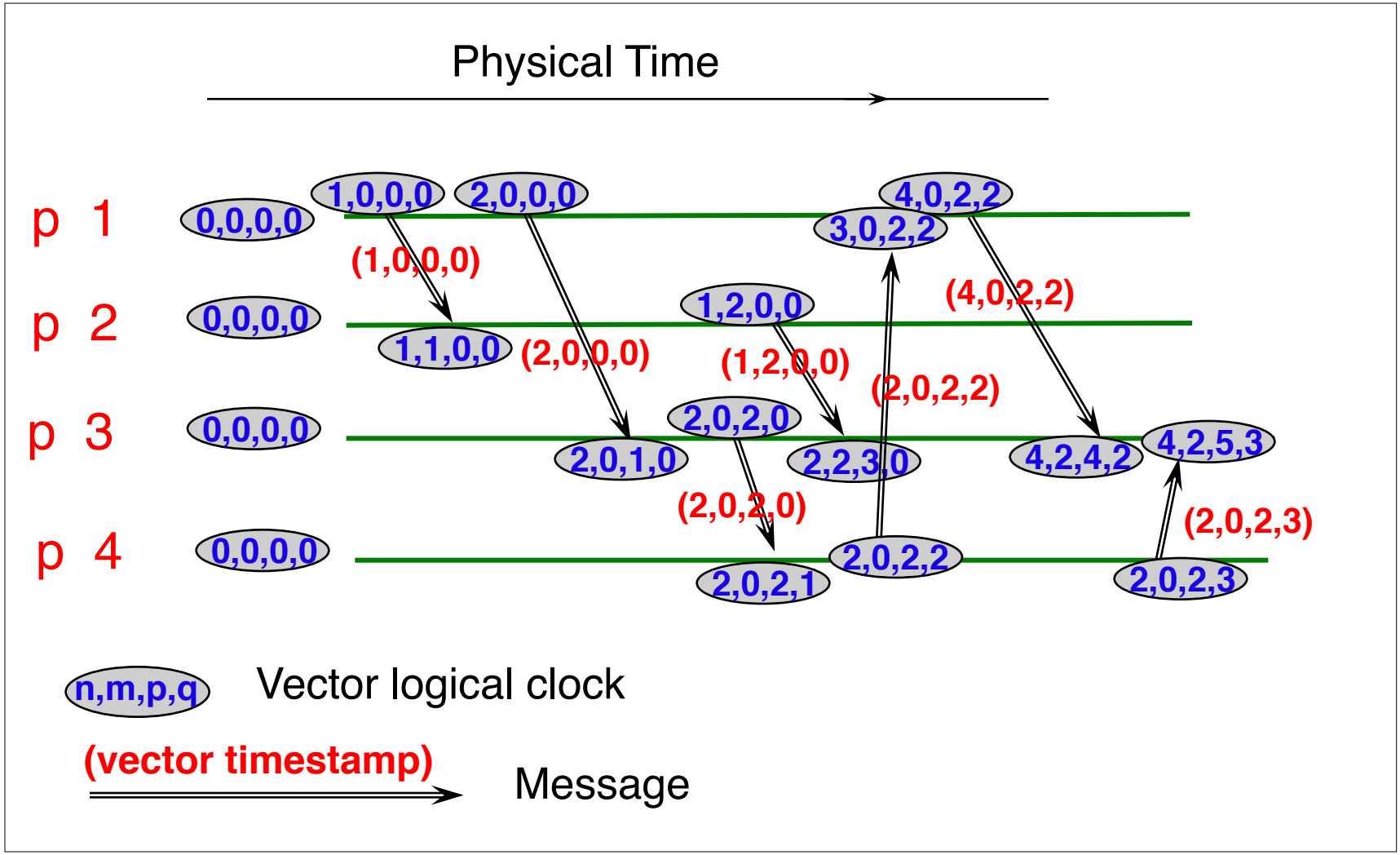
    → The new V_i is the timestamp of the event
  - A message carries the Send event's vector timestamp
  - For a receive(message) event at process k … let $V_{message}$ denote vector timestamp received with the message

$$V_k[j] = \begin{cases} \text{Max}(V_k[j] , V_{message}[j]), & \text{if } j \text{ is not } k \\ V_k[j] + 1 & j = k \end{cases}$$

Vector Timestamps



Example: Vector Timestamps



Comparing Vector Timestamps

- $VT_1 = VT_2$,

  iff $VT_1[i] = VT_2[i]$, for all $i = 1, \ldots, n$

- $VT_1 \leqslant VT_2$,

  iff $VT_1[i] \leqslant VT_2[i]$, for all $i = 1, \ldots, n$

- $VT_1 < VT_2$,

  iff $VT_1 \leqslant VT_2$ &

  $\exists\, j\ (1 \leqslant j \leqslant n\ \&\ VT_1[j] < VT_2[j])$

- **Then:** VT_1 is concurrent with VT_2

  iff (not $VT_1 < VT_2$ AND not $VT_2 < VT_1$)

The vector clock length equals number of processes.

Can shorter vectors suffice?

In general, for vector clocks, the number of elements must equal the number of processes.

Causality

- If a happened before b, then b
  "potentially causally depends" on b

- Potential causality

Summary

- Physical and Logical clocks
- Lamport (Logical) & vector timestamps
- Happened-before relation
- Causality