Distributed Systems

CS425/ECE428

Jan 30 2023

Instructor: Radhika Mittal

Logistics Related

- VM clusters have been assigned
  - *for those who filled out the forms and are enrolled in / planning to switch to 4 credits.*
  - you might have received (or will receive by tonight) some instructions over email.

- HW1 will be released on Wednesday.

- Late registration can be permitted *if there is capacity in the course*. Please talk to CS/ECE advising office.

Quick Recap: Clock Synchronization

- Synchronization in synchronous systems:
  - Synchronization bound = *(max – min)/2*

- Synchronization in asynchronous systems:
  - Cristian Algorithm: Synchronization between a client and a server.
    - Synchronization bound = $(T_{round} / 2) - min \leq T_{round} / 2$
  - Berkeley Algorithm: internal synchronization between clocks.
    - A central server picks the average time and disseminates offsets.
  - Network Time Protocol
    - Hierarchical time synchronization over the Internet.
    - Symmetric mode synchronization.

Today's agenda

- Logical Clocks and Timestamps
  - Chapter 14.4
- Global State (if time)
  - Chapter 14.5

Event Ordering

- A usecase of synchronized clocks:
  - Reasoning about order of events.
- Why is it useful?
  - Reconciling updates made to an object in a distributed datastore.
  - Rollback recovery during failures:
    1. *Checkpoint state of the system; 2. Log events (with timestamps); 3. Rollback to checkpoint and replay events in order if system crashes.*
  - …
- Can we reason about order of events without synchronized clocks?

Process, state, events

- Consider a system with **n** processes: $<p_1, p_2, p_3, \ldots, p_n>$
- Each process $\mathbf{p_i}$ is described by its *state* $\mathbf{s_i}$ that gets transformed over time.
  - State includes values of all local variables, affected files, etc.
- $\mathbf{s_i}$ gets transformed when an *event* occurs.
- Three types of events:
  - Local computation.
  - Sending a message.
  - Receiving a message.

Event Ordering

- Easy to order events within a single process $\mathbf{p_i}$, based on their time of occurrence.

- How do we reason about events across processes?
  - A message must be *sent* before it gets *received* at another process.

- These two notions help define *happened-before* (HB) relationship denoted by $\rightarrow$.
  - $\mathbf{e} \rightarrow \mathbf{e'}$ means **e** *happened before* **e'**.

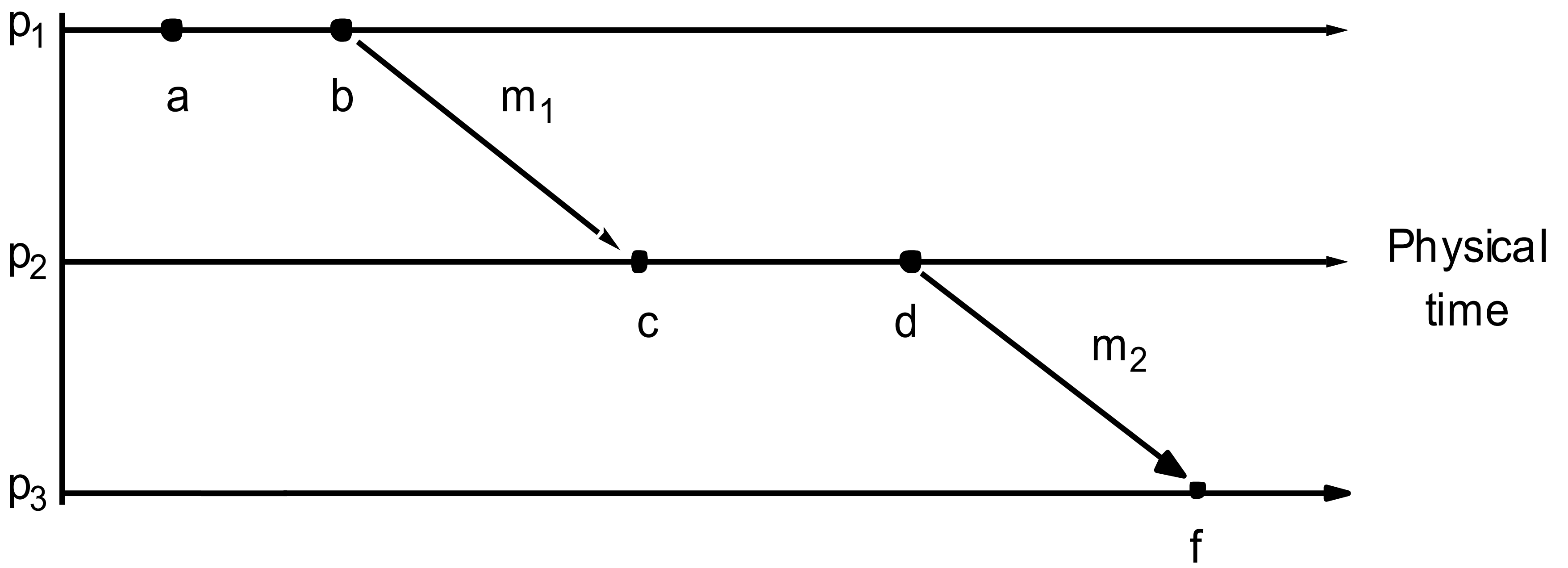
Happened-Before Relationship

- *Happened-before* (HB) relationship denoted by →.
  - **e** → **e'** means **e** *happened before* **e'**.
  - **e** $\rightarrow_i$ **e'** means **e** *happened before* **e'**, as observed by $\mathbf{p_i}$.

- HB rules:
  - If ∃ $\mathbf{p_i}$, **e** $\rightarrow_i$ **e'** then **e** → **e'**.
  - For any message m, **send(m)** → **receive(m)**
  - If **e** → **e'** and **e'** → **e''** then **e** → **e''**

- Also called *"causal" or "potentially causal"* ordering.

Event Ordering: Example

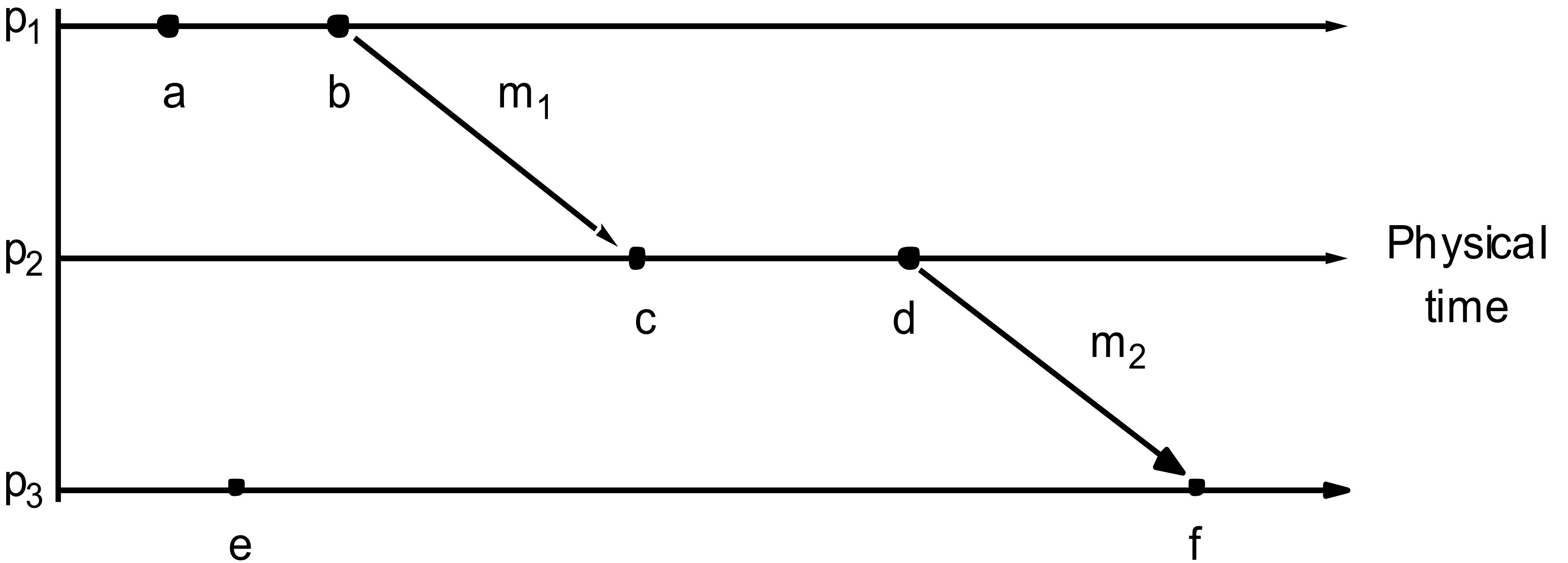


Which event happened first?

a → b and b → c and c → d and d → f

a → b and a → c and a → d and a → f

Event Ordering: Example



What can we say about **e**?

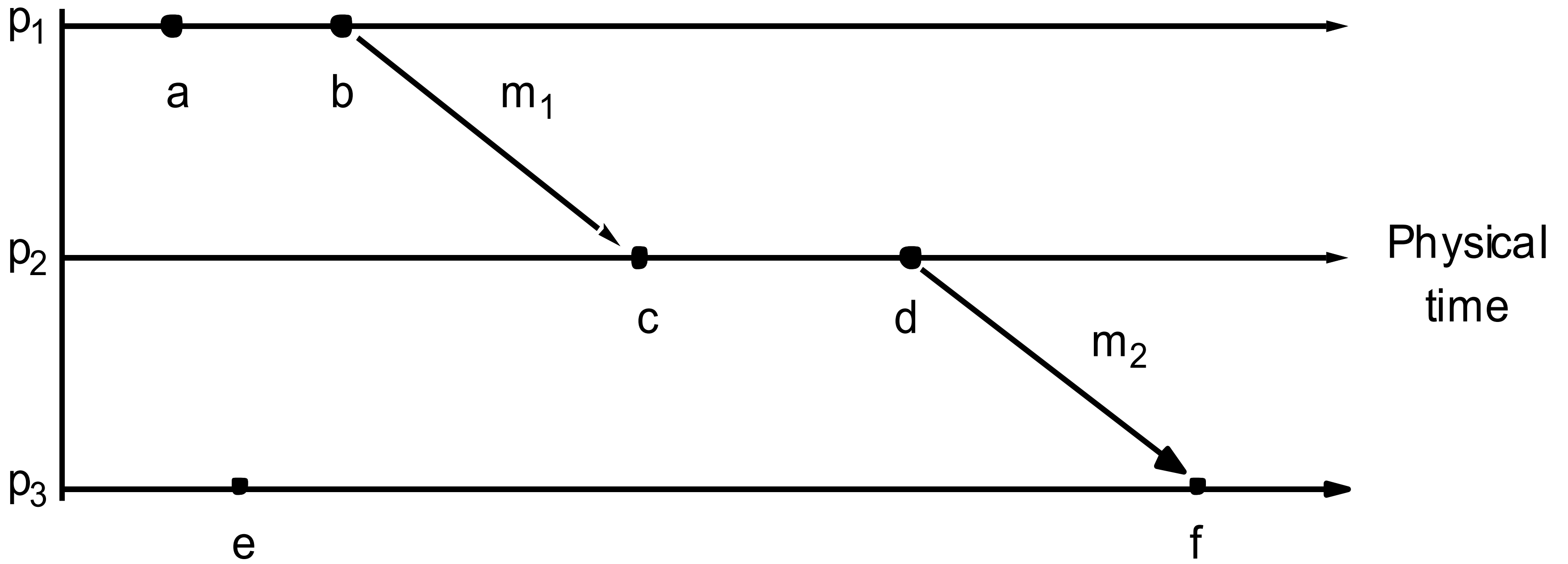
e → **f**

a $\not\rightarrow$ **e** and **e** $\not\rightarrow$ **a**

a || **e**

a and **e** are *concurrent*.

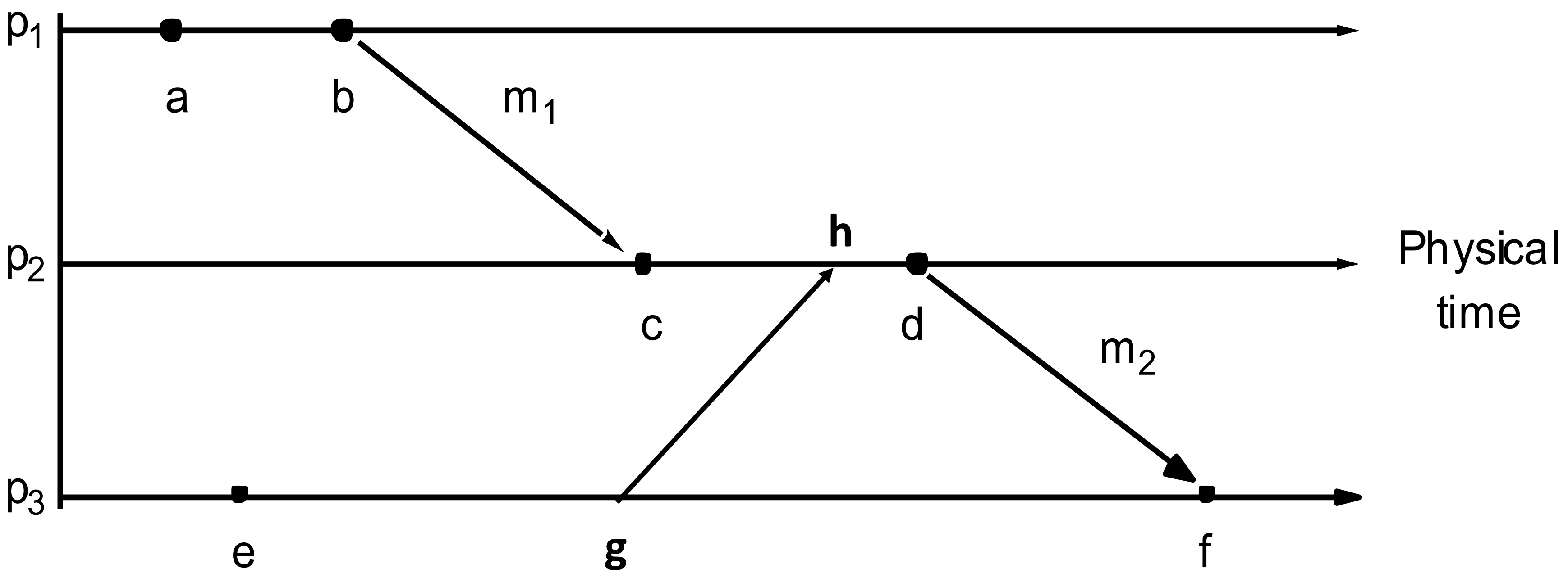
Event Ordering: Example



What can we say about **e** and **d**?

e || d

Logical Timestamps: Example



What can we say about **e** and **d**?

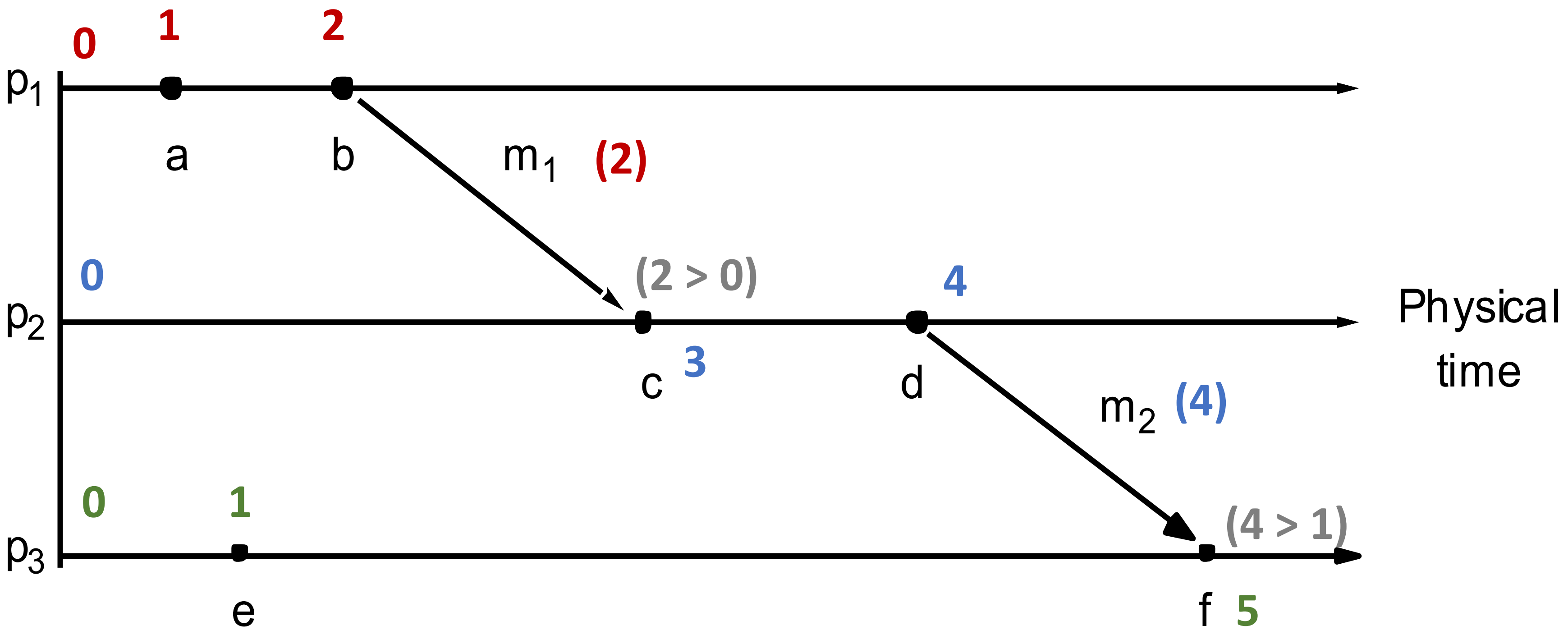
e → d

Lamport's Logical Clock

- Logical timestamp for each event that captures the *happened-before* relationship.

- *Algorithm:* Each process $\mathbf{p_i}$
  1. initializes local clock $\mathbf{L_i = 0}$.
  2. increments $\mathbf{L_i}$ before timestamping each event.
  3. piggybacks $\mathbf{L_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{L_i = max(t, L_i)}$
     - increments $\mathbf{L_i}$ before timestamping the receive event (as per step 2).

Logical Timestamps: Example

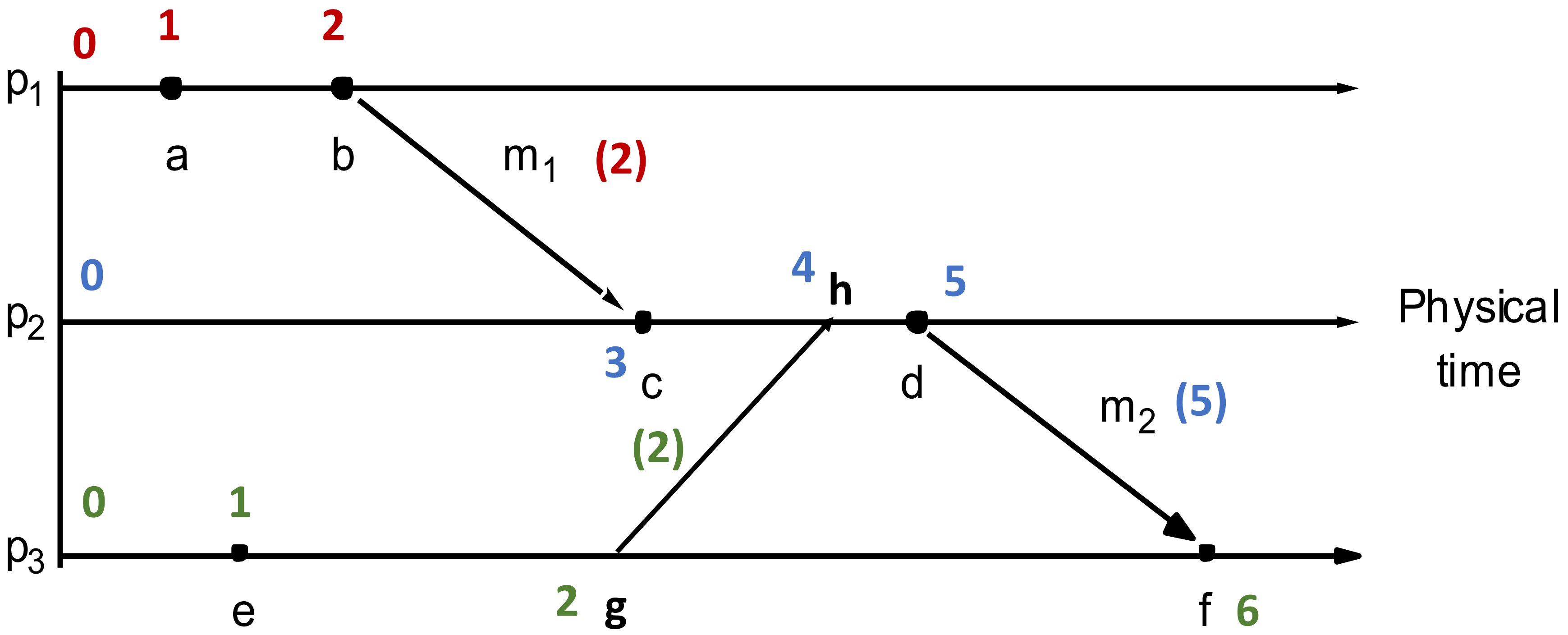


Lamport's Logical Clock

- Logical timestamp for each event that captures the *happened-before* relationship.

- *Algorithm:* Each process $\mathbf{p_i}$
  1. initializes local clock $\mathbf{L_i = 0}$.
  2. increments $\mathbf{L_i}$ before timestamping each event.
  3. piggybacks $\mathbf{L_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{L_i = max(t, L_i)}$
     - increments $\mathbf{L_i}$ before timestamping the receive event (as per step 2).

Logical Timestamps: Example



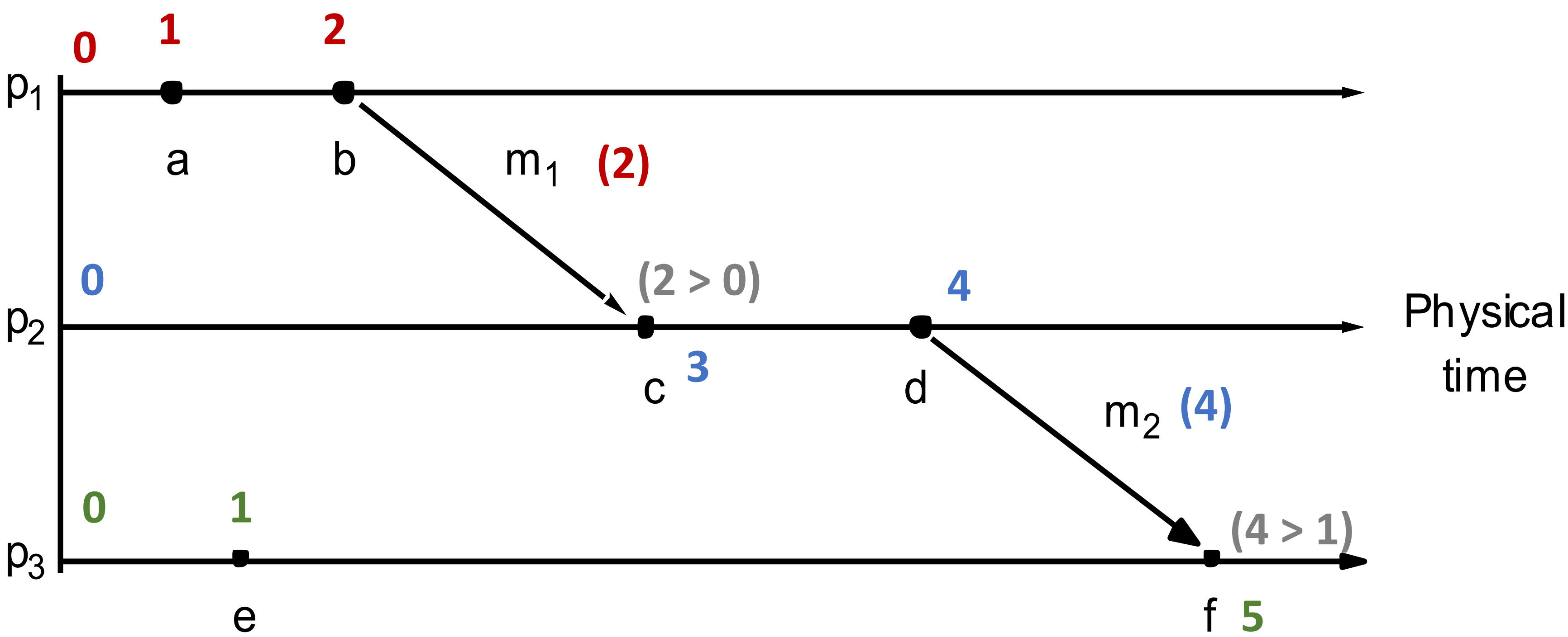
Lamport's Logical Clock

- Logical timestamp for each event that captures the *happened-before* relationship.

- If **e** → **e'** then
  - **L(e) < L(e')**

- What if **L(e) < L(e')**?
  - We cannot say that **e** → **e'**
  - We can say: **e'** ↛ **e**
  - Either **e** → **e'** or **e** || **e'**

Logical Timestamps: Example



L(e) < L(d), e || d   L(e) < L(f), e → f

Vector Clocks

- Each event associated with a vector timestamp.
- Each process p_i maintains vector of clocks V_i
- The size of this vector is the same as the no. of processes.
    - $V_i[j]$ is the clock for process p_j as maintained by p_i
- Algorithm: each process p_i:

Vector Clocks

- Each event associated with a vector timestamp.
- Each process p_i maintains vector of clocks V_i
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$ is the clock for process p_j as maintained by p_i
- Algorithm: each process p_i:
  1. initializes local clock $V_i[j] = 0$

Vector Clocks

- Each event associated with a vector timestamp.
- Each process p_i maintains vector of clocks V_i
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$ is the clock for process p_j as maintained by p_i
- Algorithm: each process p_i:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.

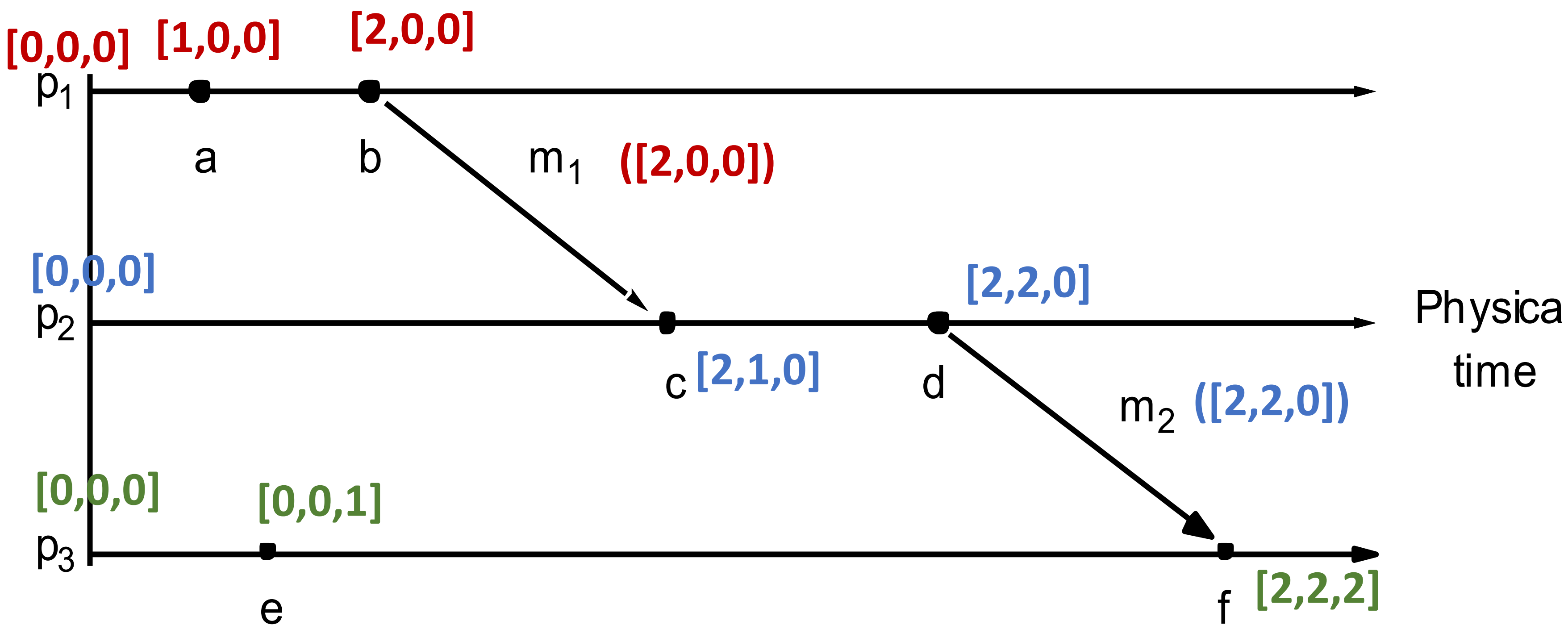
Vector Clocks

- Each event associated with a vector timestamp.
- Each process p_i maintains vector of clocks V_i
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$ is the clock for process p_j as maintained by p_i
- Algorithm: each process p_i:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.
  3. piggybacks V_i when sending a message.

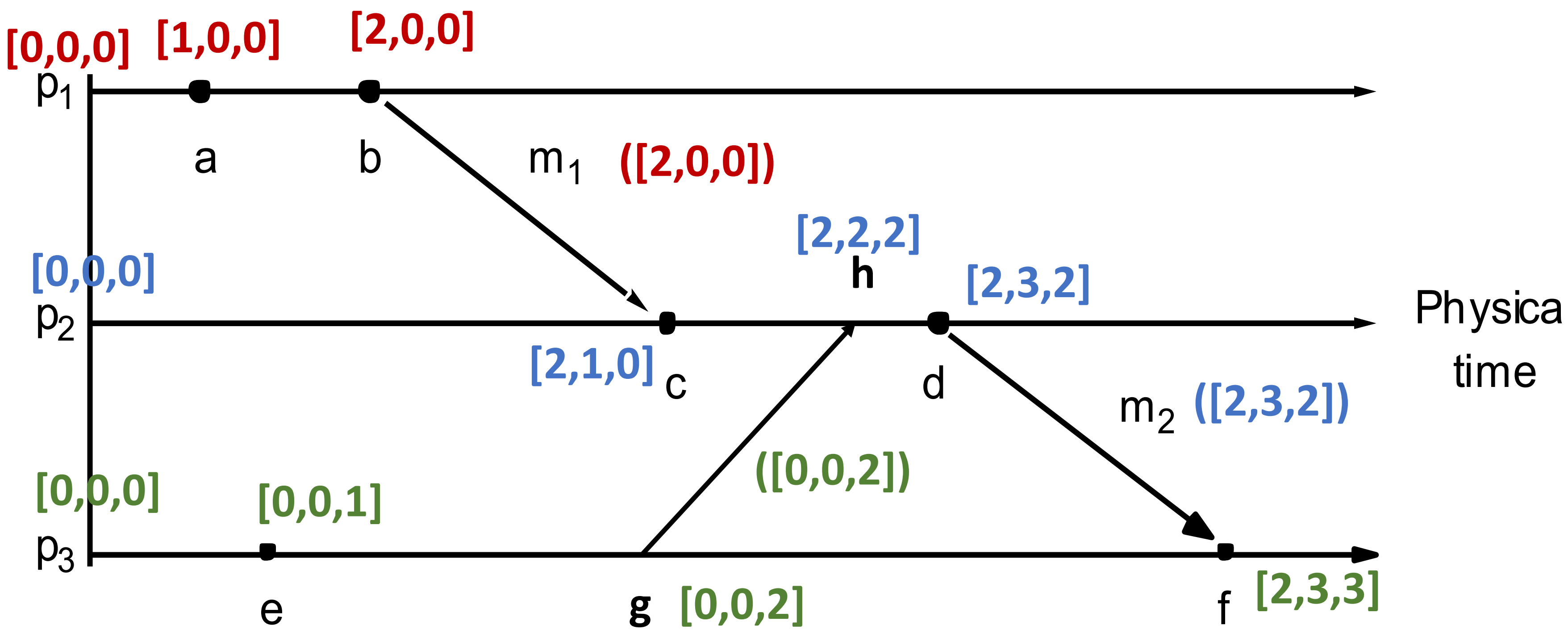
Vector Clocks

- Each event associated with a vector timestamp.
- Each process p_i maintains vector of clocks V_i
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$ is the clock for process p_j as maintained by p_i
- Algorithm: each process p_i:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.
  3. piggybacks V_i when sending a message.
  4. upon receiving a message with vector clock value **v**
     - sets $V_i[j] = \max(V_i[j], v[j])$ for all j=1…n.
     - increments $V_i[i]$ before timestamping receive event (as per step 2).

Vector Timestamps: Example



Vector Timestamps: Example



Comparing Vector Timestamps

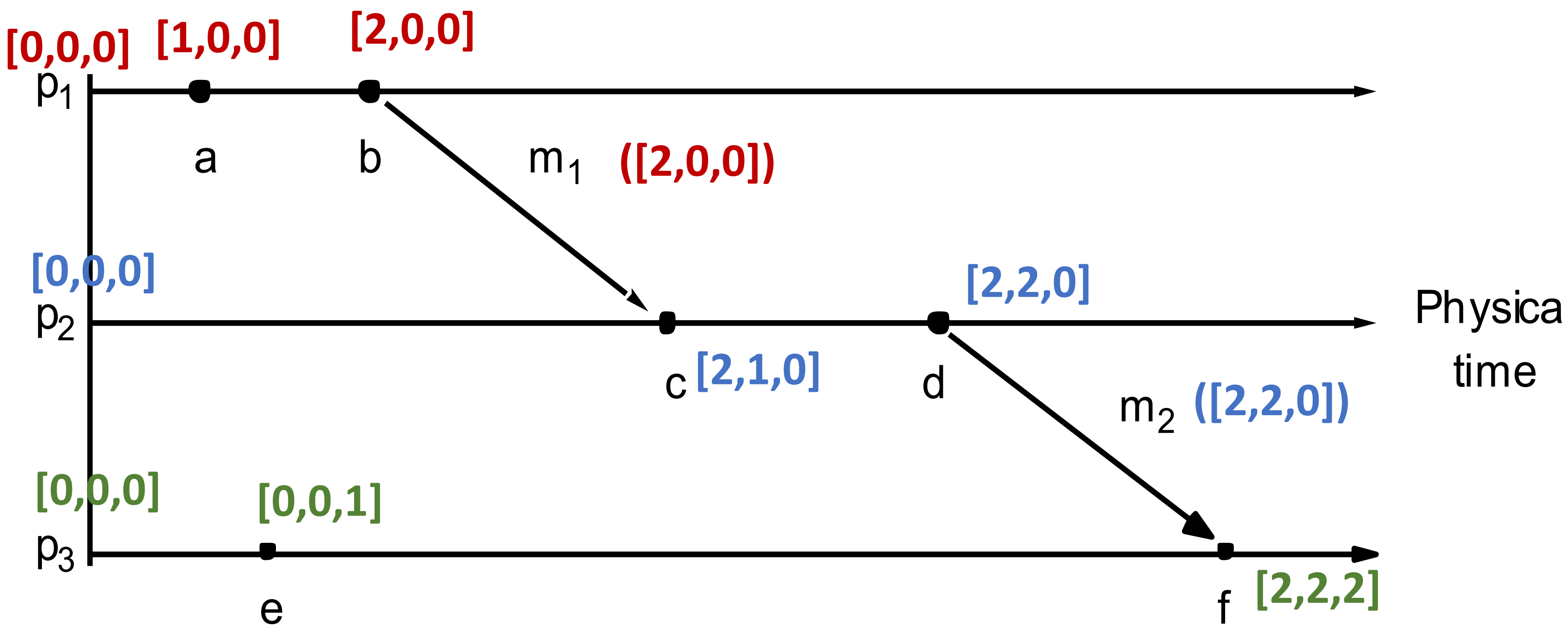
- Let V(e) = V and V(e') = V'

- V = V', iff V[i] = V'[i], for all i = 1, … , n
- V ≤ V', iff V[i] ≤ V'[i], for all i = 1, … , n
- V < V', iff V ≤ V' & V ≠ V'

  iff V ≤ V' & ∃ j such that (V[j] < V'[j])

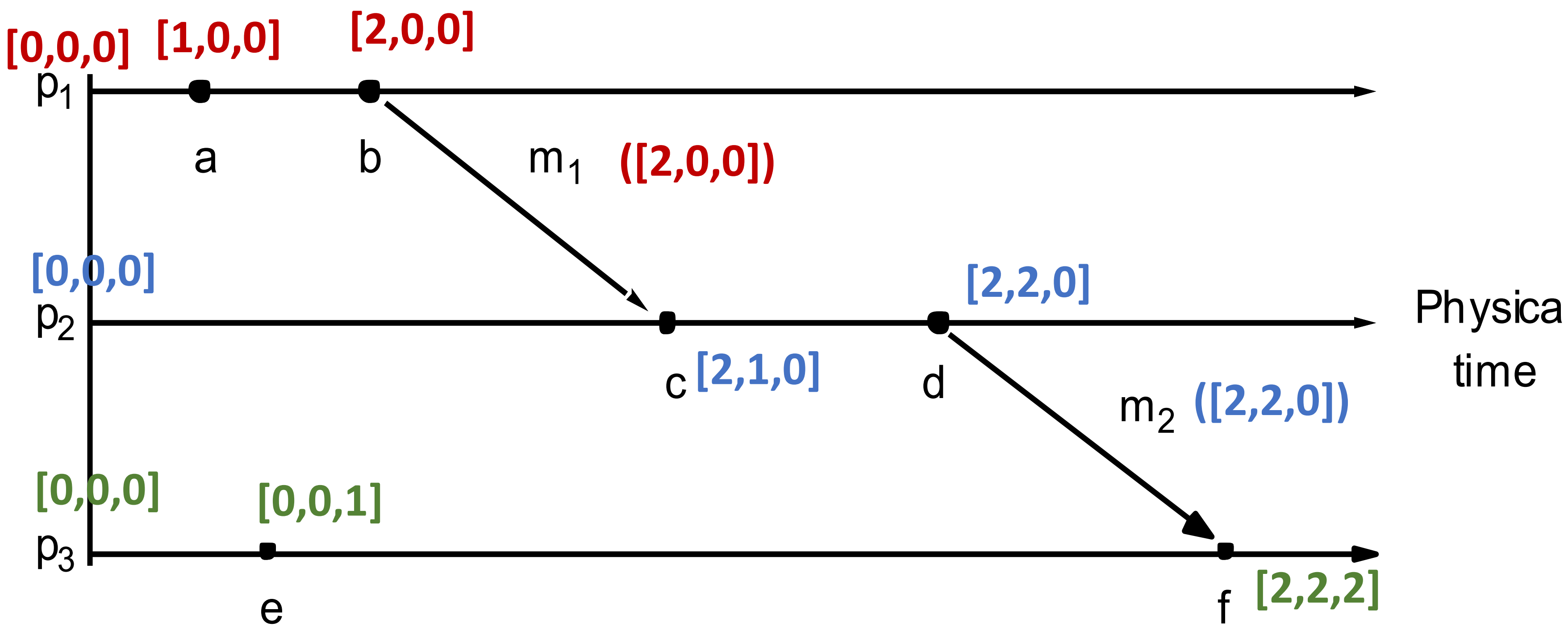
- e → e' iff V < V'
  - (V < V' implies e → e') and (e → e' implies V < V')
- e || e' iff (V ≮ V' and V' ≮ V)

Vector Timestamps: Example



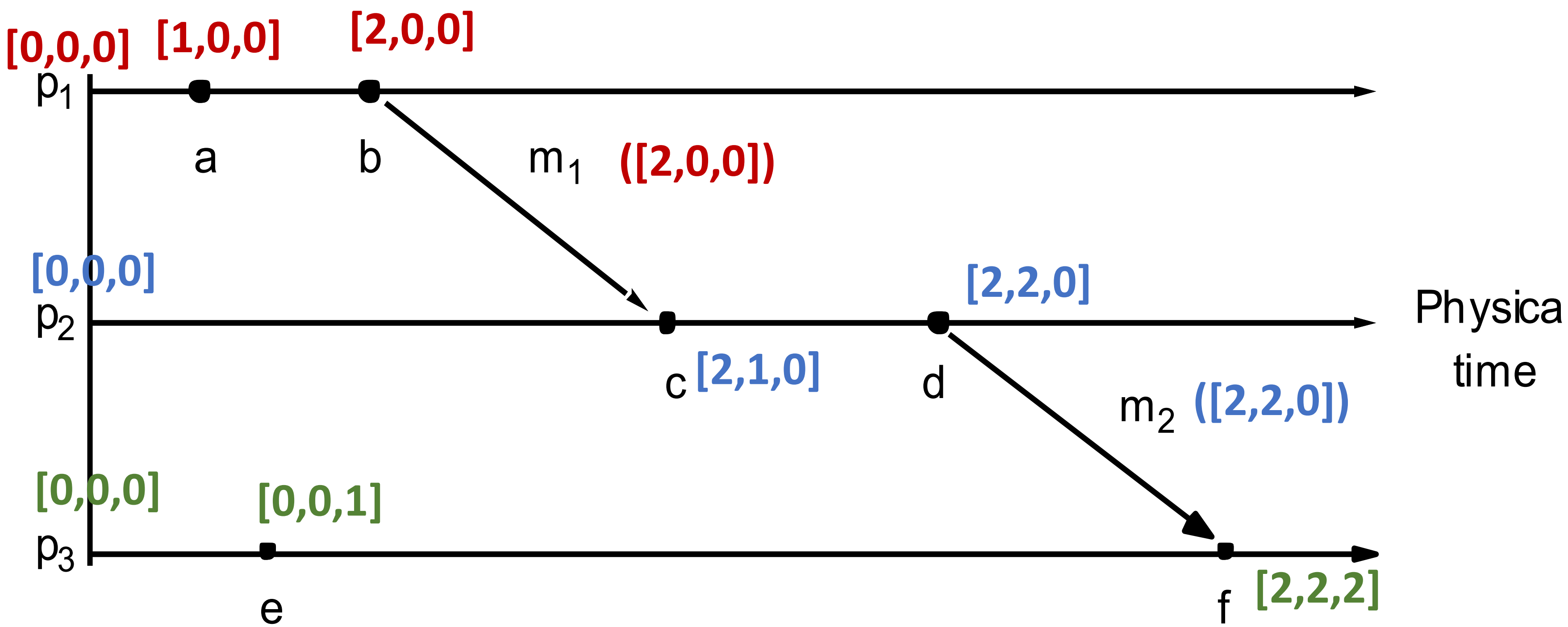
What can we say about e & f based on their vector timestamps?

Vector Timestamps: Example



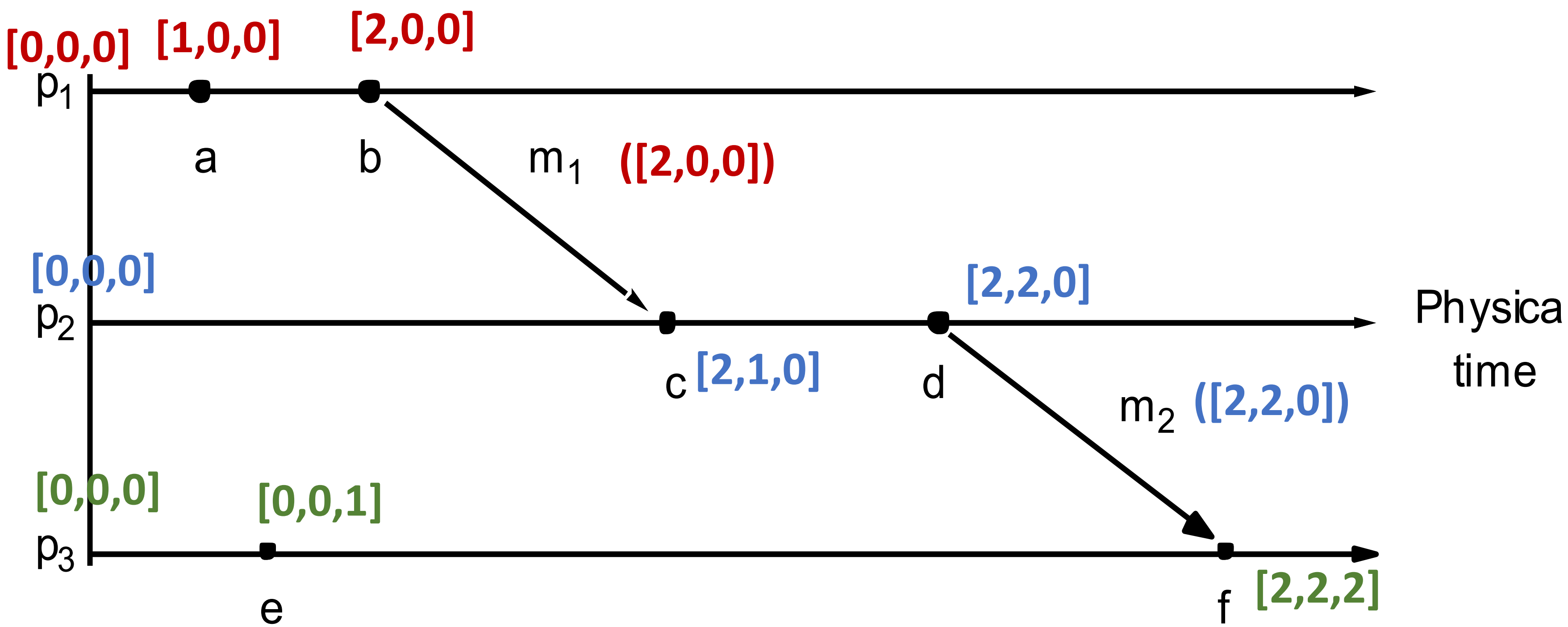
V(e) < V(f), e → f

Vector Timestamps: Example



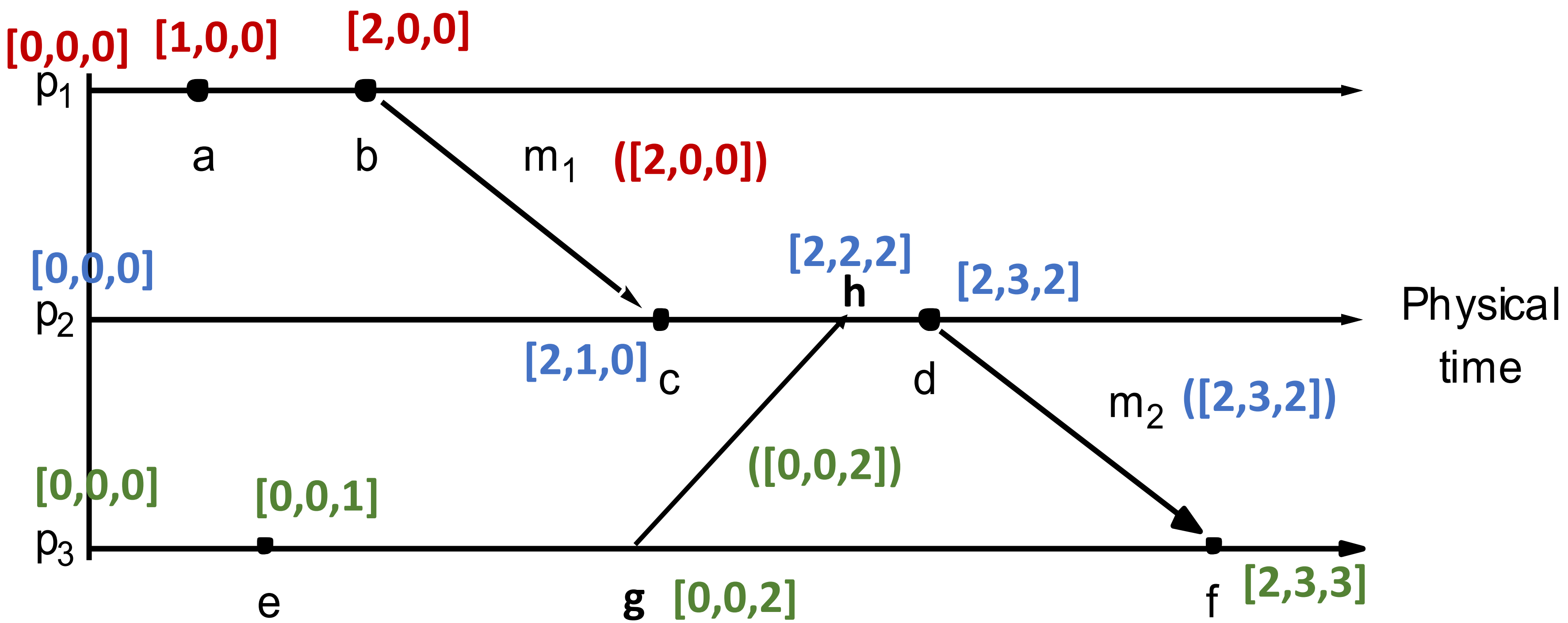
What can we say about e & d based on their vector timestamps?

Vector Timestamps: Example



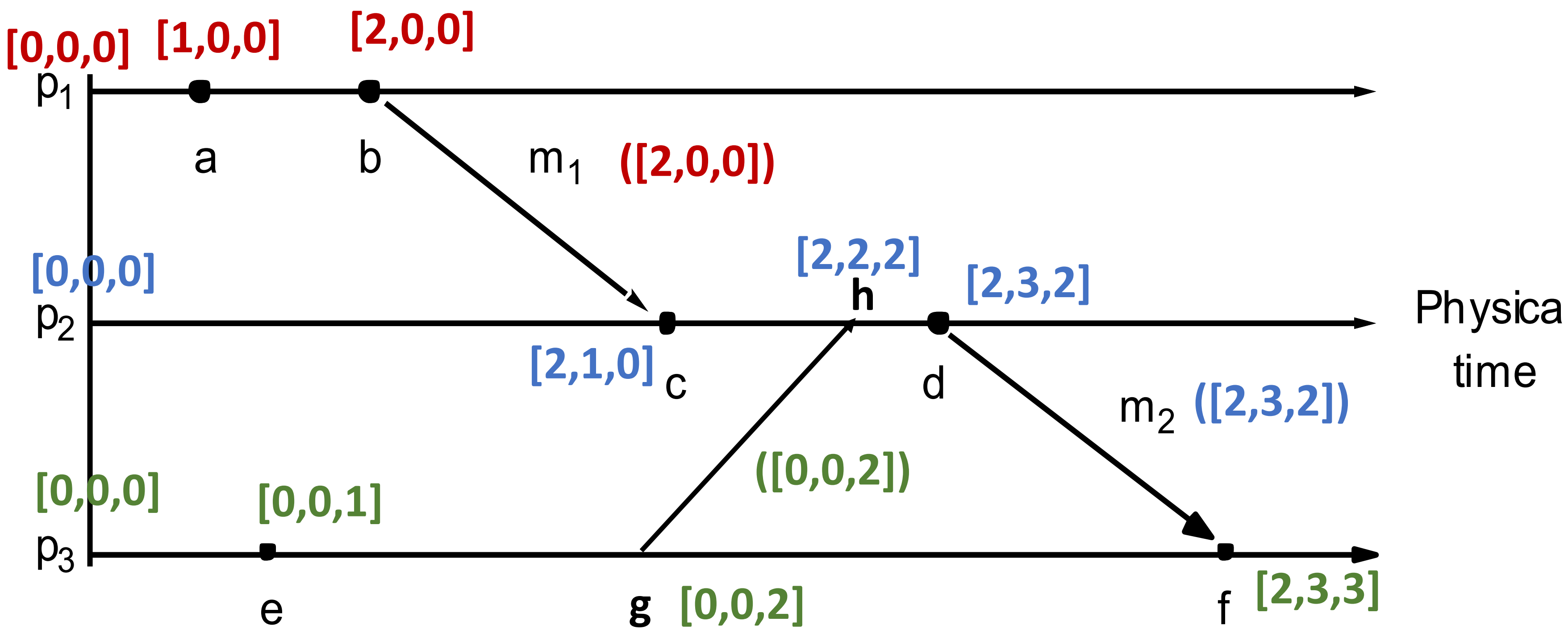
$V(e) \not< V(d)$ and $V(d) \not< V(e)$, $e \parallel d$

Vector Timestamps: Example



How about now?

Vector Timestamps: Example



V(e) < V(f), e → f

V(e) < V(d), e → d

Timestamps Summary

- **Comparing timestamps across events is useful.**
  - Reconciling updates made to an object in a distributed datastore.
  - Rollback recovery during failures:

    1. Checkpoint state of the system; 2. Log events (with timestamps);
    3. Rollback to checkpoint and replay events in order if system crashes.

- **How to compare timestamps across different processes?**
  - **Physical timestamp:** requires clock synchronization.
    - Google's Spanner Distributed Database uses "TrueTime".
  - **Lamport's timestamps:** cannot fully differentiate between causal and concurrent ordering of events.
    - Oracle uses "System Change Numbers" based on Lamport's clock.
  - **Vector timestamps:** larger message sizes.
    - Amazon's DynamoDB uses vector clocks.

Timestamps Summary

- **Comparing timestamps across events is useful.**
  - Reconciling updates made to an object in a distributed datastore.
  - Rollback recovery during failures:

    1. Checkpoint state of the system; 2. Log events (with timestamps);
    3. Rollback to checkpoint and replay events in order if system crashes.

- **How to compare timestamps across different processes?**
  - **Physical timestamp:** requires clock synchronization.
    - Google's Spanner Distributed Database uses "TrueTime".
  - **Lamport's timestamps:** cannot fully differentiate between causal and concurrent ordering of events.
    - Oracle uses "System Change Numbers" based on Lamport's clock.
  - **Vector timestamps:** larger message sizes.
    - Amazon's DynamoDB uses vector clocks.

Today's agenda

- Logical Clocks and Timestamps
  - Chapter 14.4
- Global State
  - Chapter 14.5

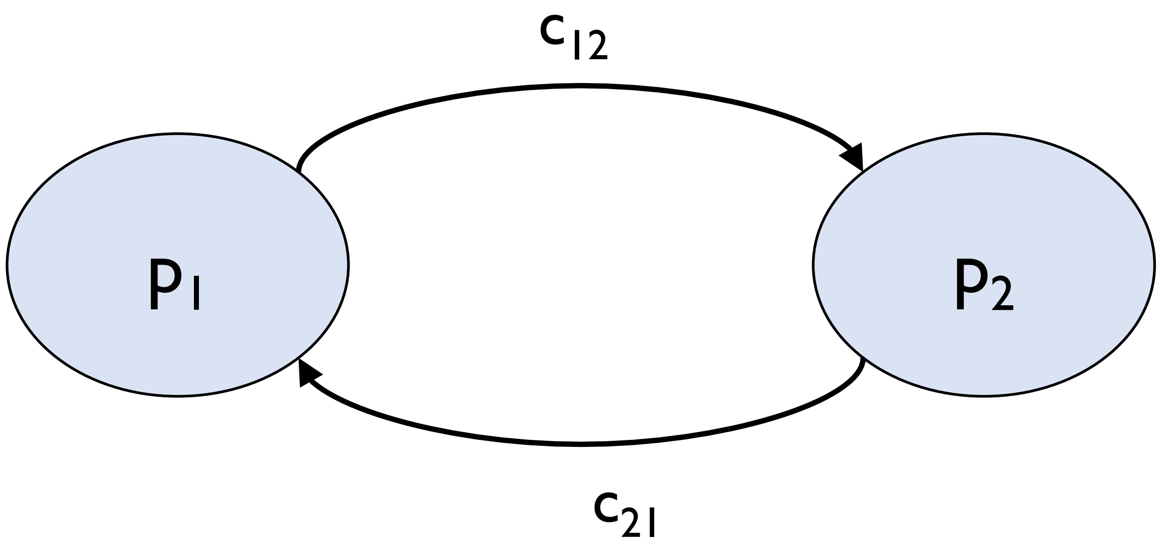
Process, state, events

- Consider a system with **n** processes: $<p_1, p_2, p_3, \ldots., p_n>$.
- Each process p_i is associated with *state* s_i.
  - State includes values of all local variables, affected files, etc.
- Each channel can also be associated with a state.
  - Which messages are currently *pending* on the channel.
  - Can be computed from process' state:
    - Record when a process sends and receives messages.
    - if p_i sends a message that p_j has not yet received, it is pending on the channel.
- State of a process (or a channel) gets transformed when an *event* occurs. 3 types of events:
  - local computation, sending a message, receiving a message.

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

- Example:



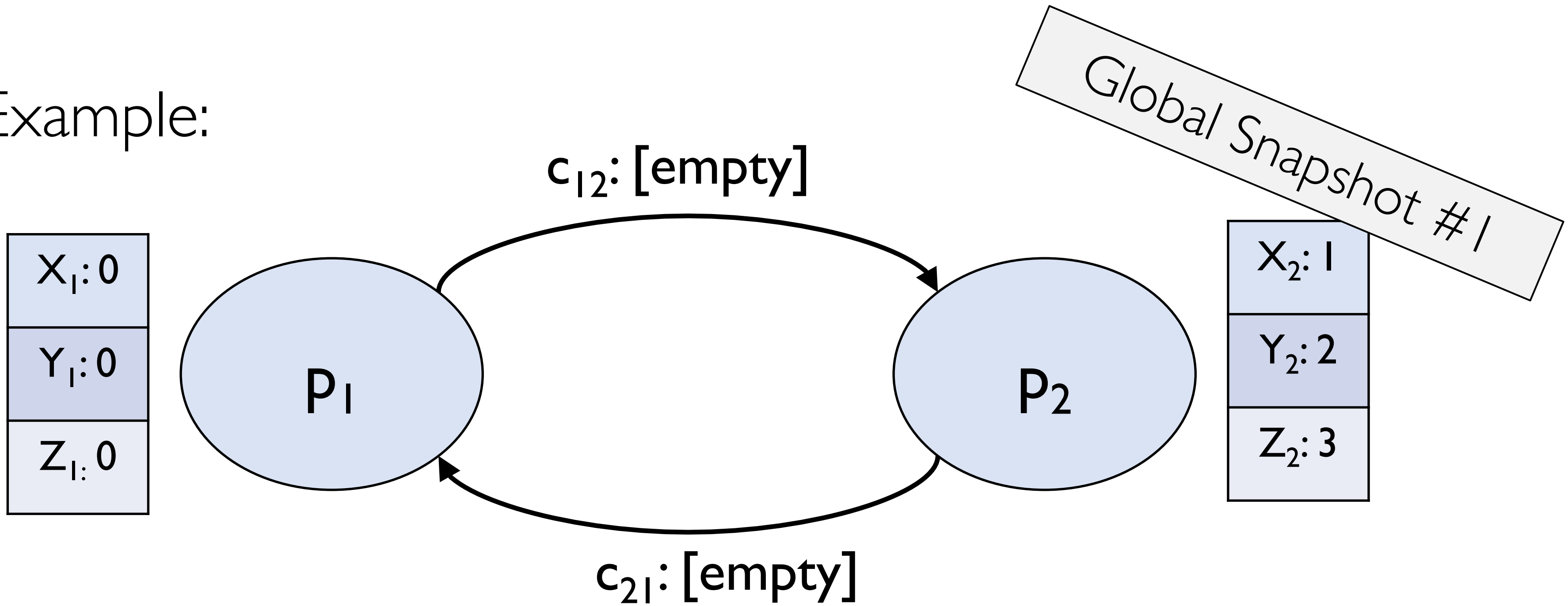
Two processes: $\mathbf{p_1}$ and $\mathbf{p_2}$.

$\mathbf{c_{12}}$: channel from $\mathbf{p_1}$ to $\mathbf{p_2}$. $\mathbf{c_{21}}$: channel from $\mathbf{p_2}$ to $\mathbf{p_1}$.

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

- Example:

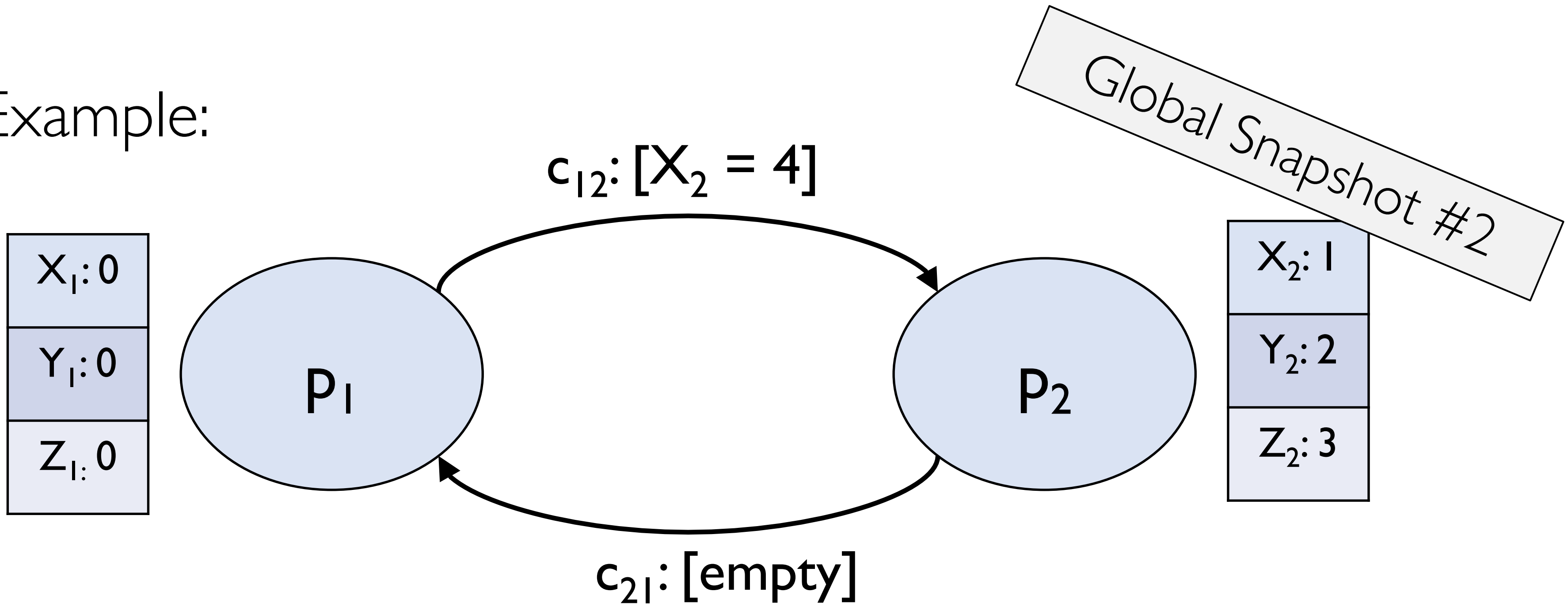


Process state for $\mathbf{p_1}$ and $\mathbf{p_2}$.
No pending messages on the channels.

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

- Example:

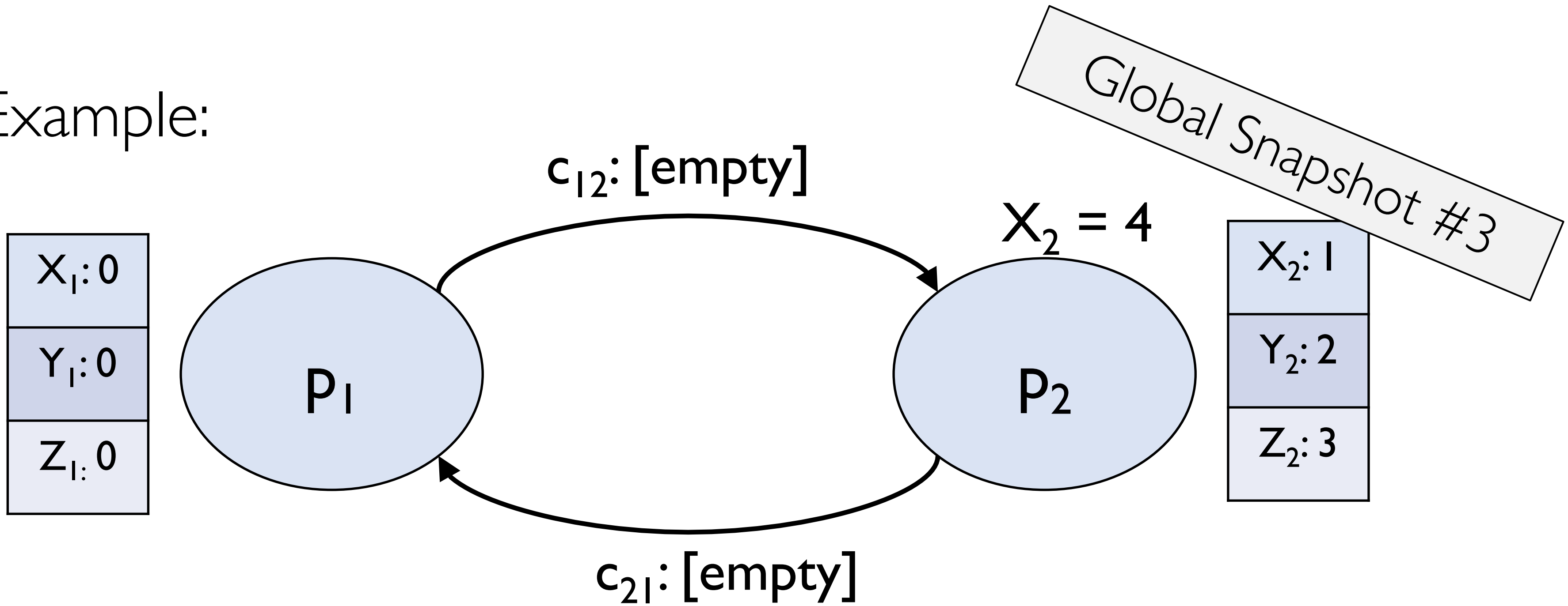


p_1 sends a message to p_2 asking it to set $X_2 = 4$

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

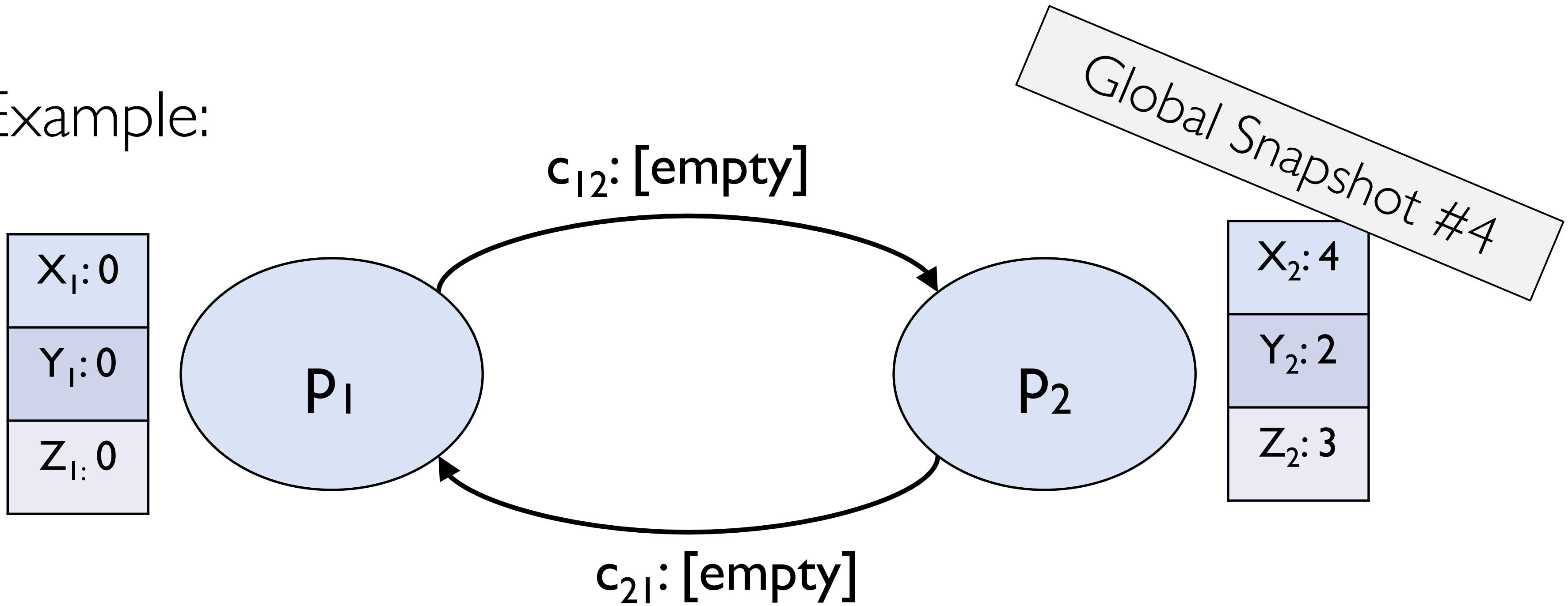
- Example:



p_2 receives the message.

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.
- Example:



p_2 changes the value of X_2

Capturing a global snapshot

- Useful to capture a global snapshot of the system:
  - *Checkpointing* the system state.
  - Reasoning about unreferenced objects (for garbage collection).
  - Distributed debugging.

Capturing a global snapshot

- Difficult to capture a global snapshot of the system.
- Global state or global snapshot is state of each process (and each channel) in the system at a given *instant of time*.
- Strawman:
  - Each process records its state at 3:15pm.
  - We get the global state of the system at 3:15pm.
  - *But precise clock synchronization is difficult to achieve.*
- **How do we capture global snapshots without precise time synchronization across processes?**
  - Our agenda for next class!