Distributed Systems

CS425/ECE428

Feb 5 2021

Instructor: Radhika Mittal

Something to think while we wait…

- What are the practical usecases of clocks and timestamps?
- Do we necessarily need synchronization to reason about ordering of events across processes?

Logistics Related

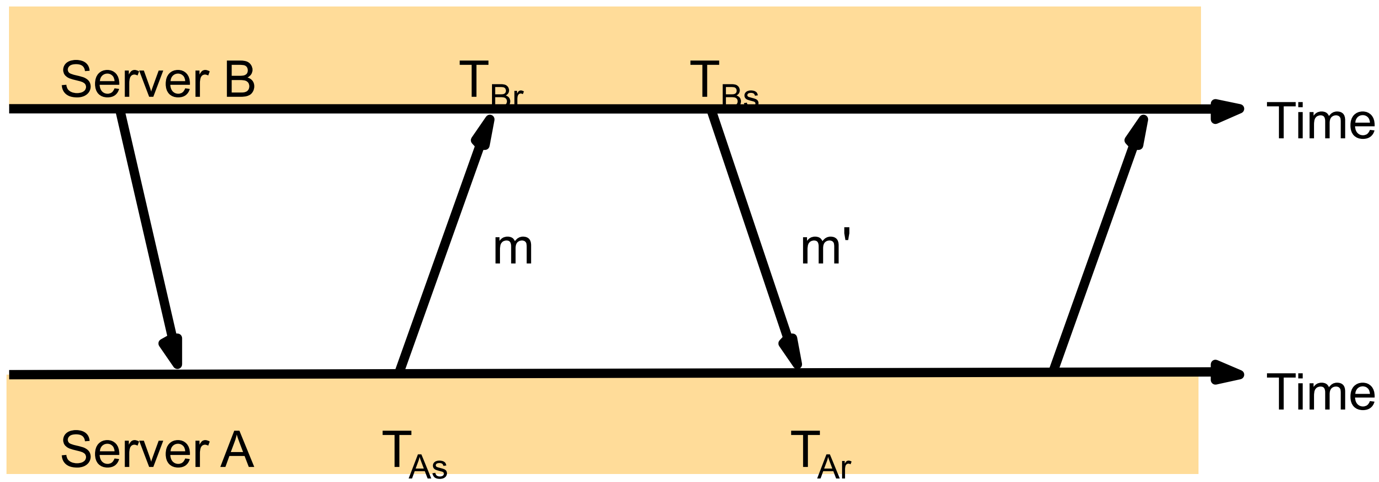
- VM clusters were assigned on Wednesday.
- HW1 has been released today.
  - Four questions in total (each with a few subparts).
  - You should be able to solve the first three questions by the end of today's class.
  - You might need to wait until Wednesday's class for the last question.

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 between lower strata servers for greater accuracy.

NTP Symmetric Mode



- t and t': actual transmission times for m and m'(unknown)
- o: <u>true</u> offset of clock at B relative to clock at A (unknown)
- o_i: <u>estimate</u> of actual offset between the two clocks
- d_i: estimate of <u>accuracy</u> of o_i ; $d_i = t + t'$
- skew estimate = $d_i/2$

$$T_{Br} = T_{As} + t + o$$
$$T_{Ar} = T_{Bs} + t' - o$$

$$o = ((T_{Br} - T_{As}) - (T_{Ar} - T_{Bs}) + (t' - t))/2$$
$$\mathbf{o_i = ((T_{Br} - T_{As}) - (T_{Ar} - T_{Bs}))/2}$$
$$o = o_i + (t' - t)/2$$
$$\mathbf{d_i = t + t' = (T_{Br} - T_{As}) + (T_{Ar} - T_{Bs})}$$

$$(o_i - d_i / 2) \leq o \leq (o_i + d_i / 2) \quad \text{given } t, t' \geq 0$$

Today's agenda

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

Today's agenda

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

Event Ordering

- A usecase of synchronized clocks:
  - Reasoning about order of events.
- 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 $\mathbf{e}$ *happened before* $\mathbf{e'}$.

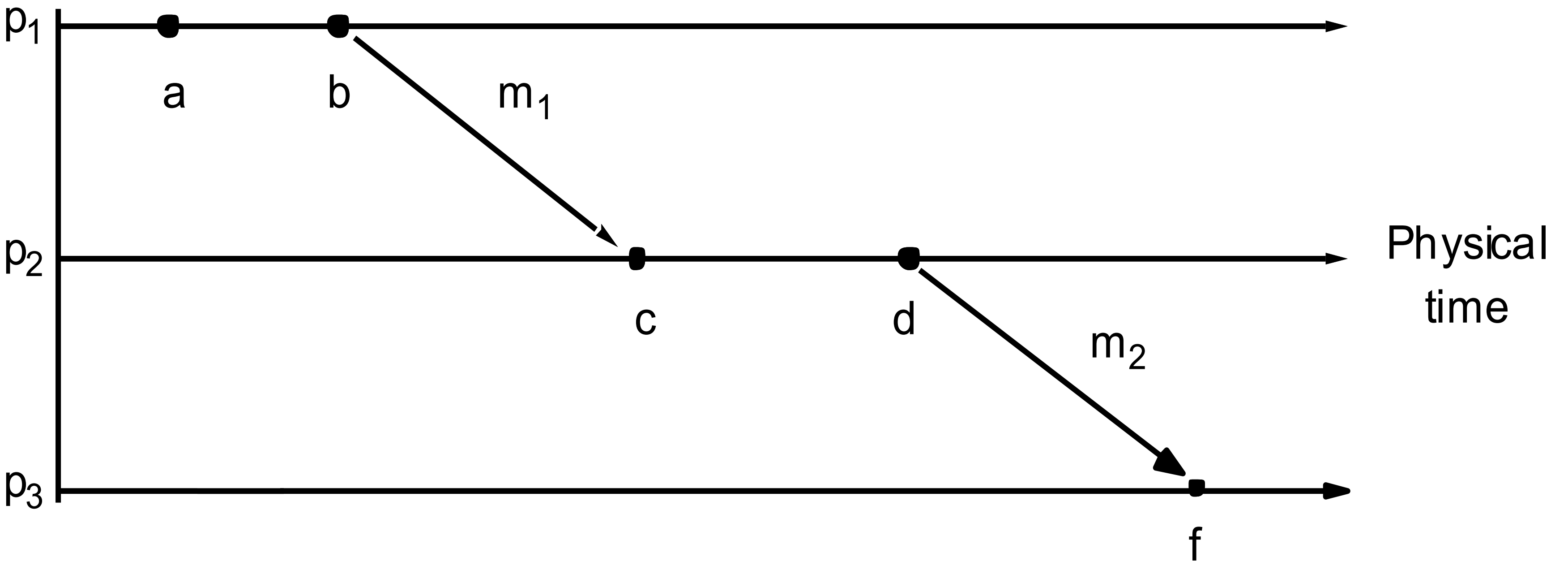
Happened-Before Relationship

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

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

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

Event Ordering: Example

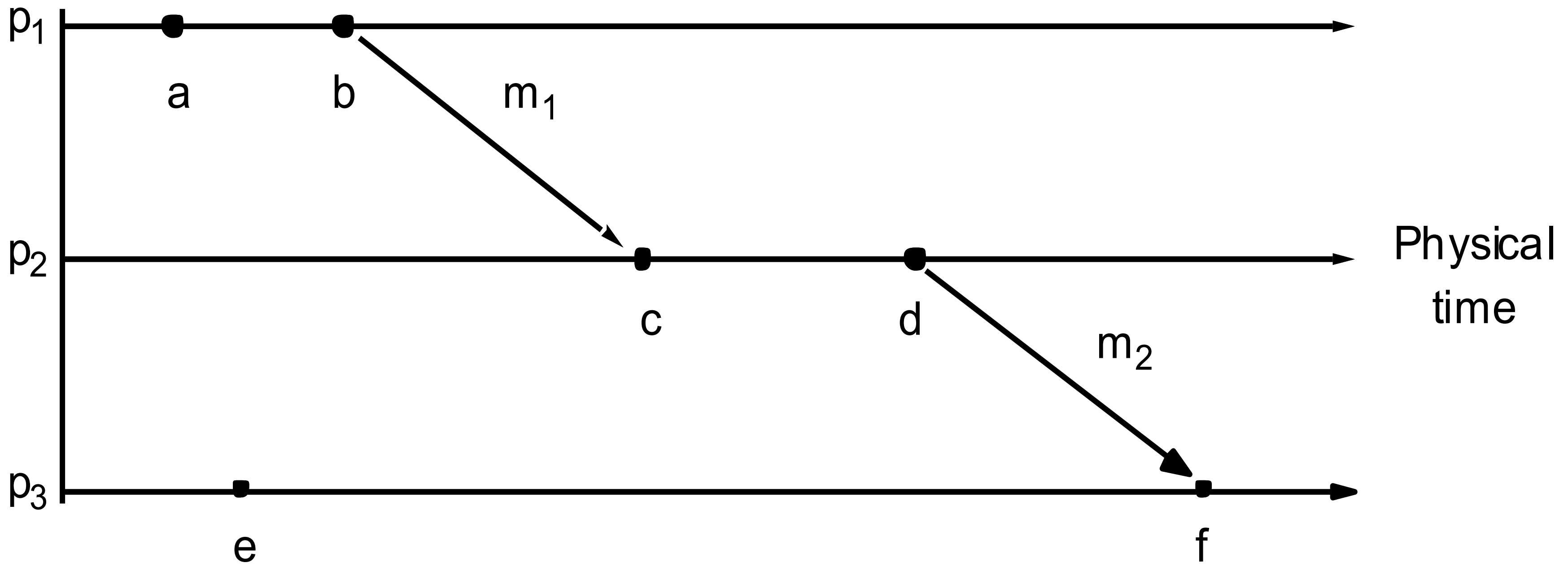


Which event happened first?

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**?

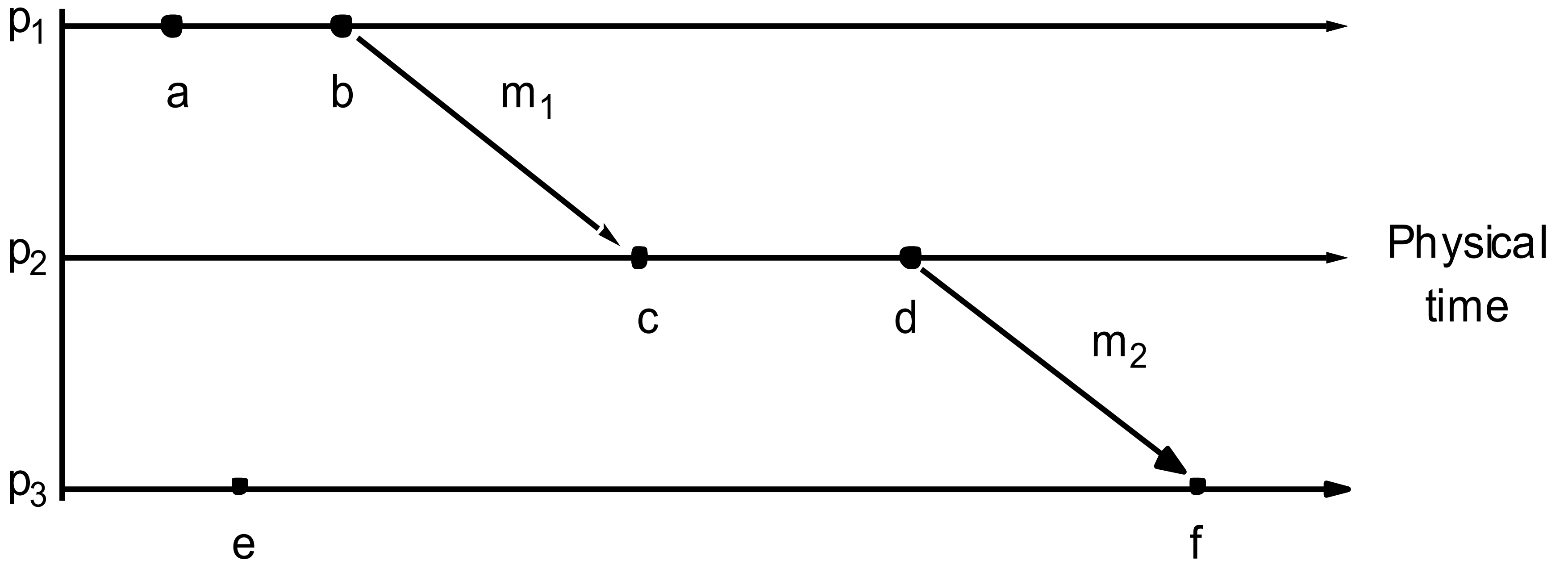
e $\rightarrow$ **f**

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

a || **e**

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

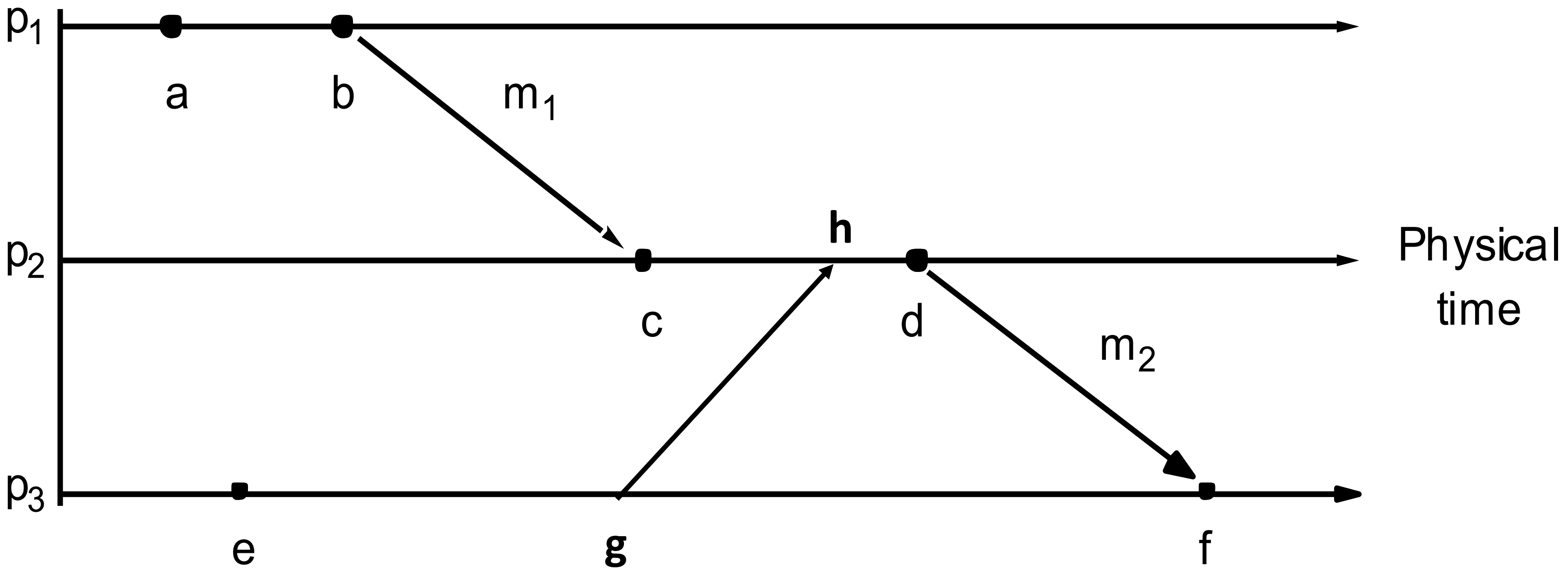
Event Ordering: Example



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

e || d

Logical Timestamps: Example



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

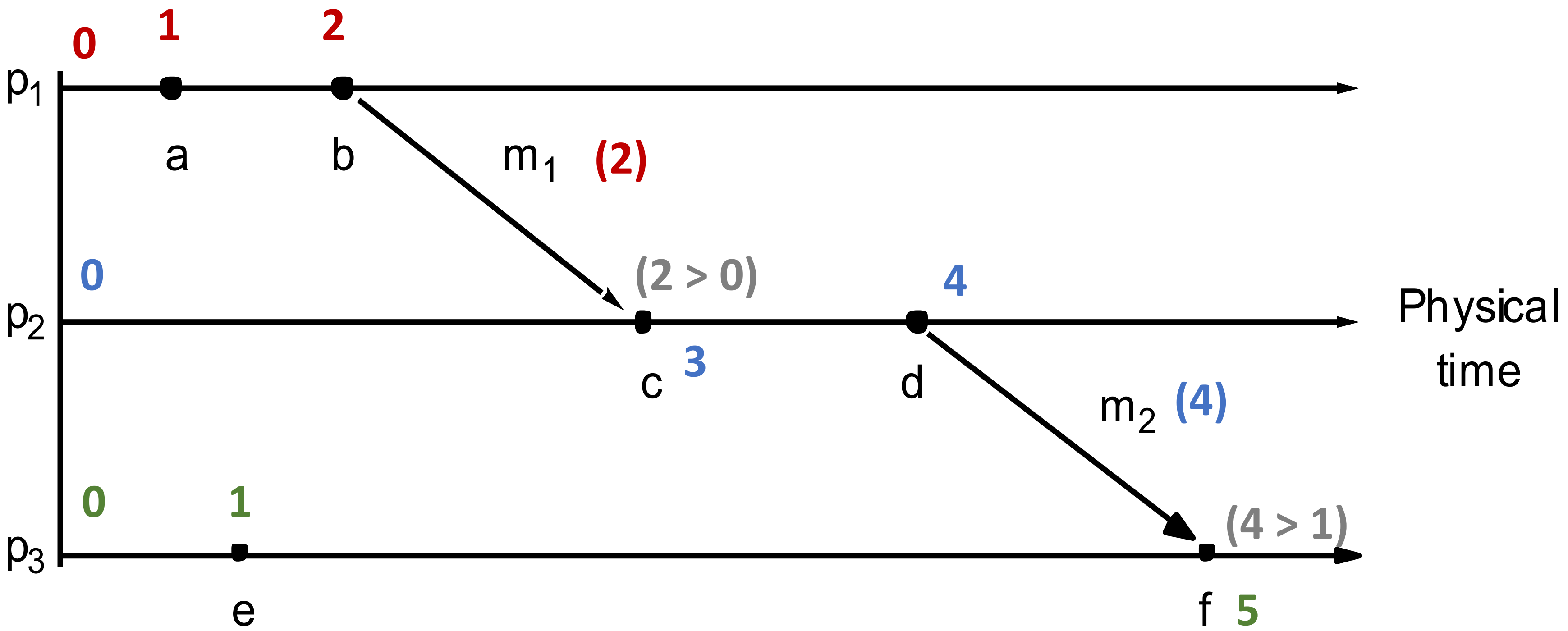
e → d

Lamport's Logical Clock

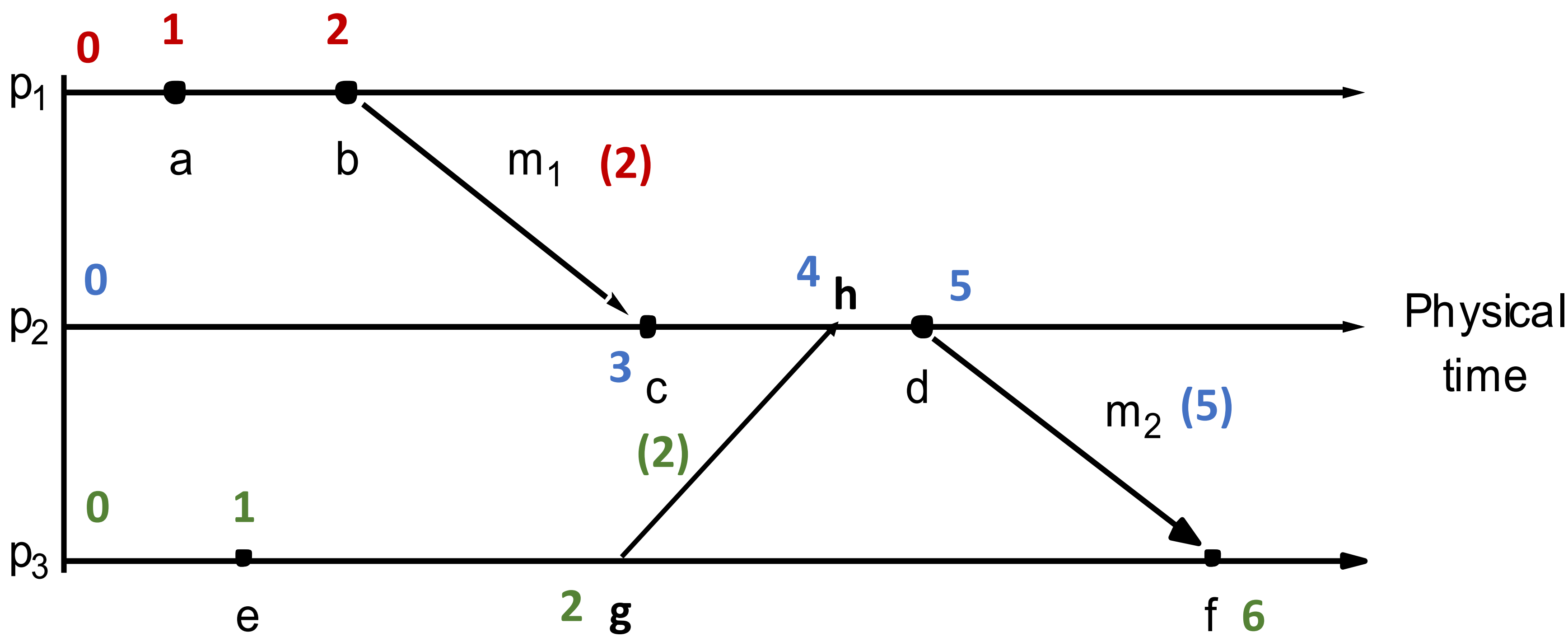
- Logical timestamp for each event that captures the *happened-before* relationship.

- *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



Logical Timestamps: Example



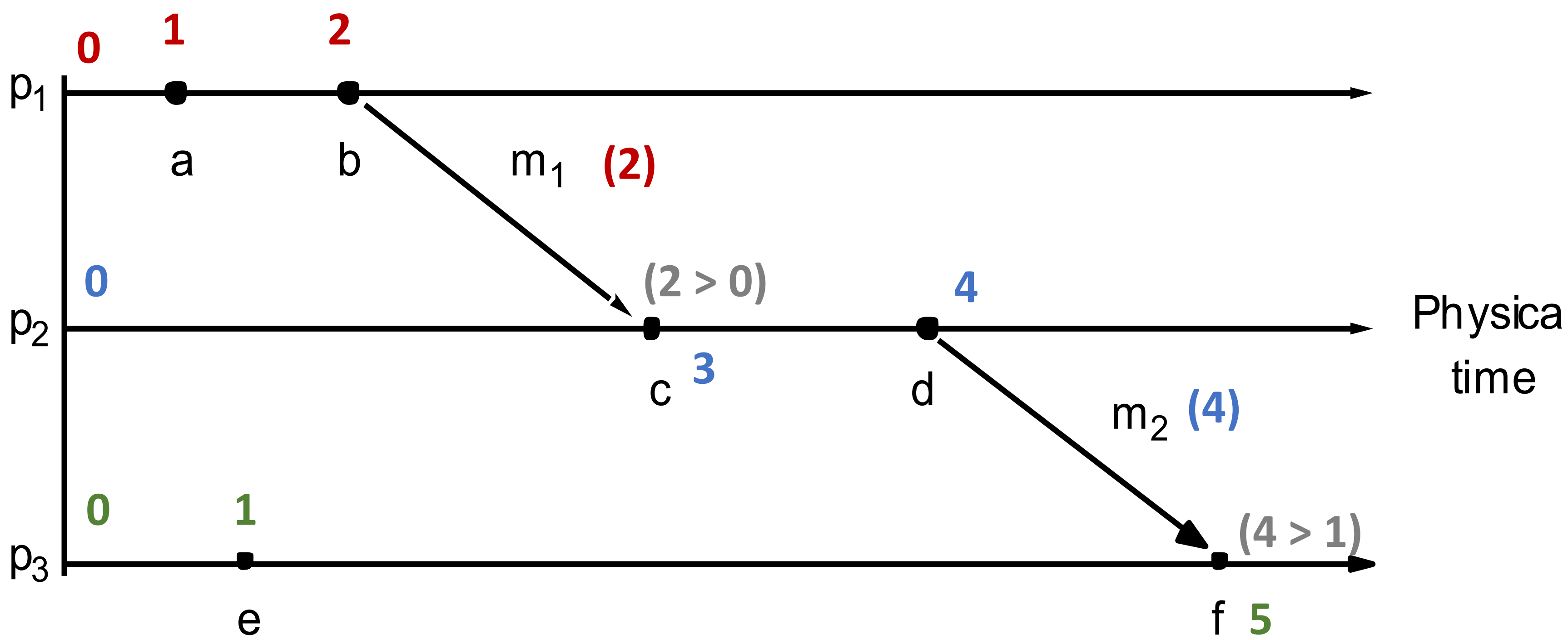
Lamport's Logical Clock

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

- If **e** $\rightarrow$ **e'** then **L(e) < L(e')**

- What if **L(e) < L(e')**?
  - We cannot say that **e** $\rightarrow$ **e'**
  - We can say: **e'** $\not\rightarrow$ **e**
  - Either **e** $\rightarrow$ **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:
  1. initializes local clock $L_i = 0$.
  2. increments L_i before timestamping each event.
  3. piggybacks L_i when sending a message.
  4. upon receiving a message with clock value t
     - sets $L_i = \max(t, L_i)$
     - increments L_i before timestamping the receive event (as per step 2).

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 L_i before timestamping each event.
  3. piggybacks L_i when sending a message.
  4. upon receiving a message with clock value t
     - sets $L_i = \max(t, L_i)$
     - increments L_i before timestamping the receive event (as per step 2).

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 L_i when sending a message.
  4. upon receiving a message with clock value t
     - sets $L_i = \max(t, L_i)$
     - increments L_i before timestamping the receive event (as per step 2).

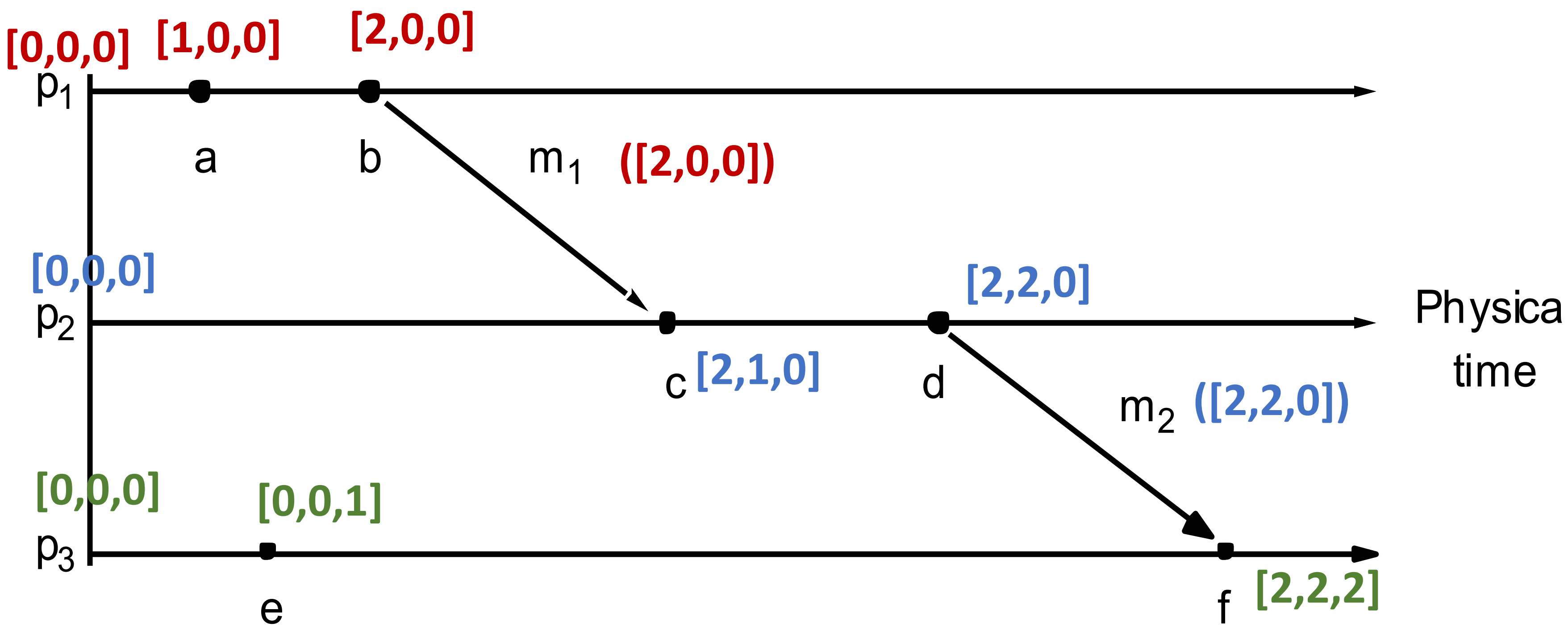
Vector Clocks

- 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 clock value t
     - sets $L_i = \max(t, L_i)$
     - increments L_i before timestamping the receive event (as per step 2).

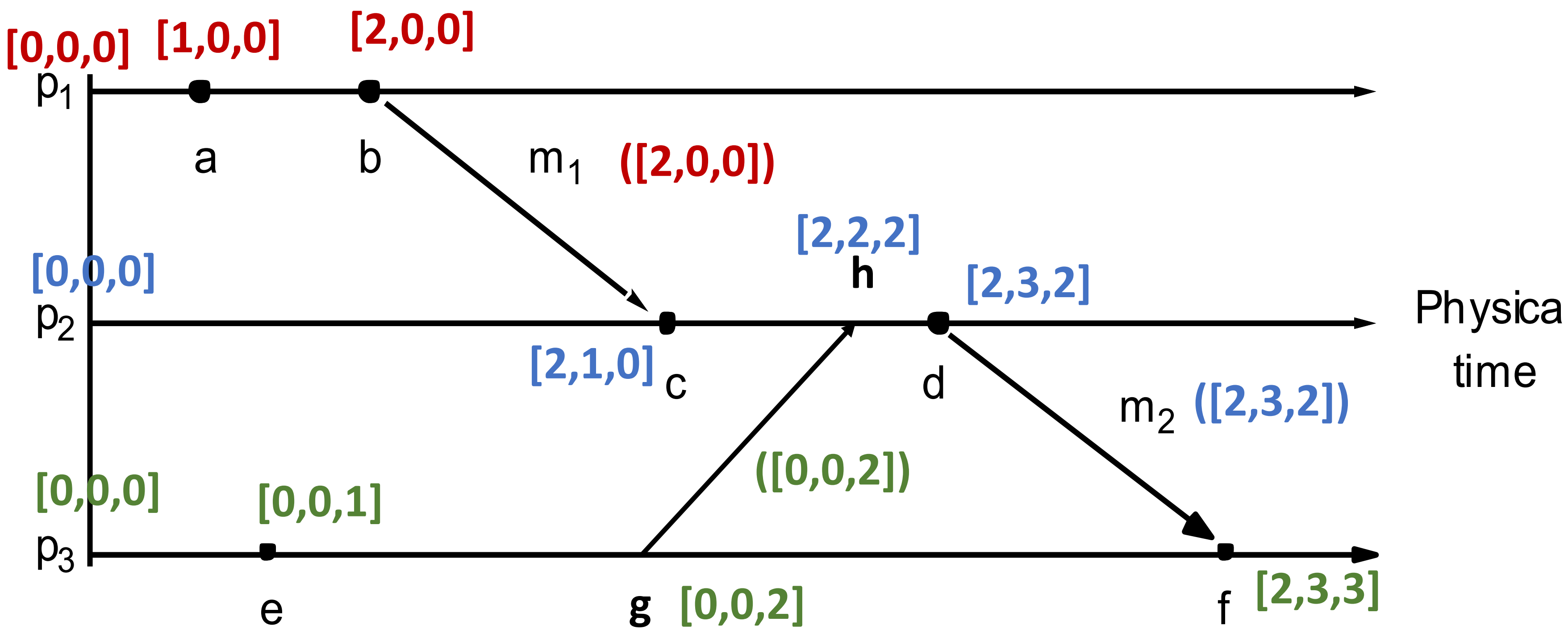
Vector Clocks

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

Vector Timestamps: Example



Vector Timestamps: Example



Comparing Vector Timestamps

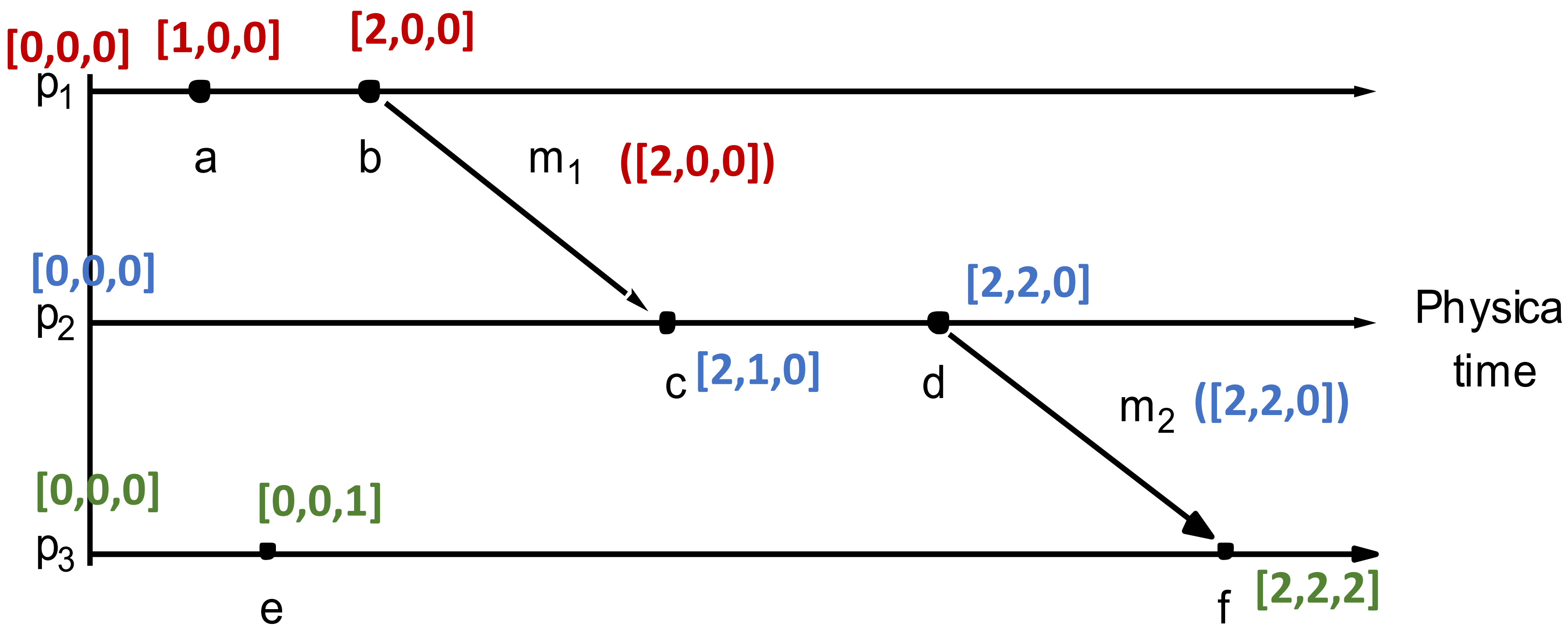
- Let $V(e) = V$ and $V(e') = V'$

- $V = V'$, iff $V[i] = V'[i]$, for all $i = 1, \ldots, n$
- $V \leq V'$, iff $V[i] \leq V'[i]$, for all $i = 1, \ldots, n$
- $V < V'$, iff $V \leq V'$ & $V \neq V'$

  iff $V \leq V'$ & $\exists$ j such that $(V[j] < V'[j])$

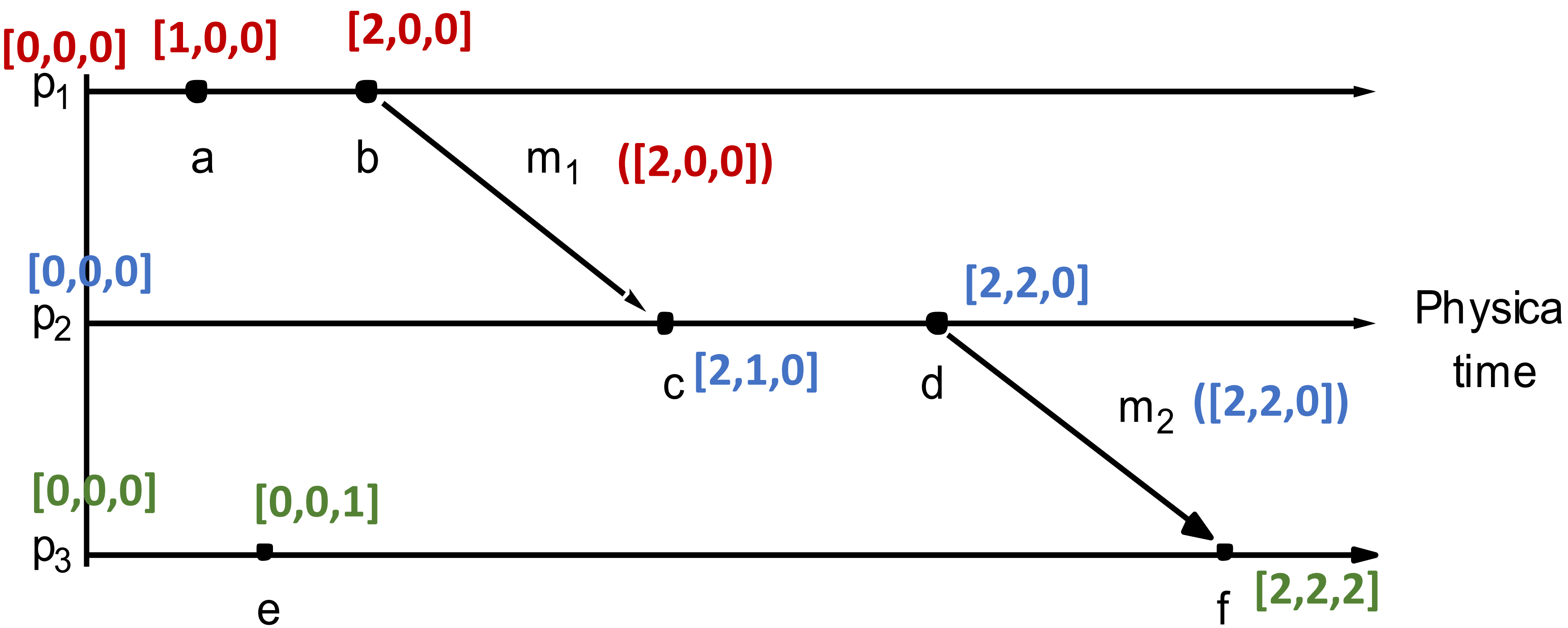
- $e \rightarrow e'$ iff $V < V'$
  - ($V < V'$ implies $e \rightarrow e'$) and ($e \rightarrow e'$ implies $V < V'$)
- $e \| e'$ iff ($V \nless V'$ and $V' \nless V$)

Vector Timestamps: Example



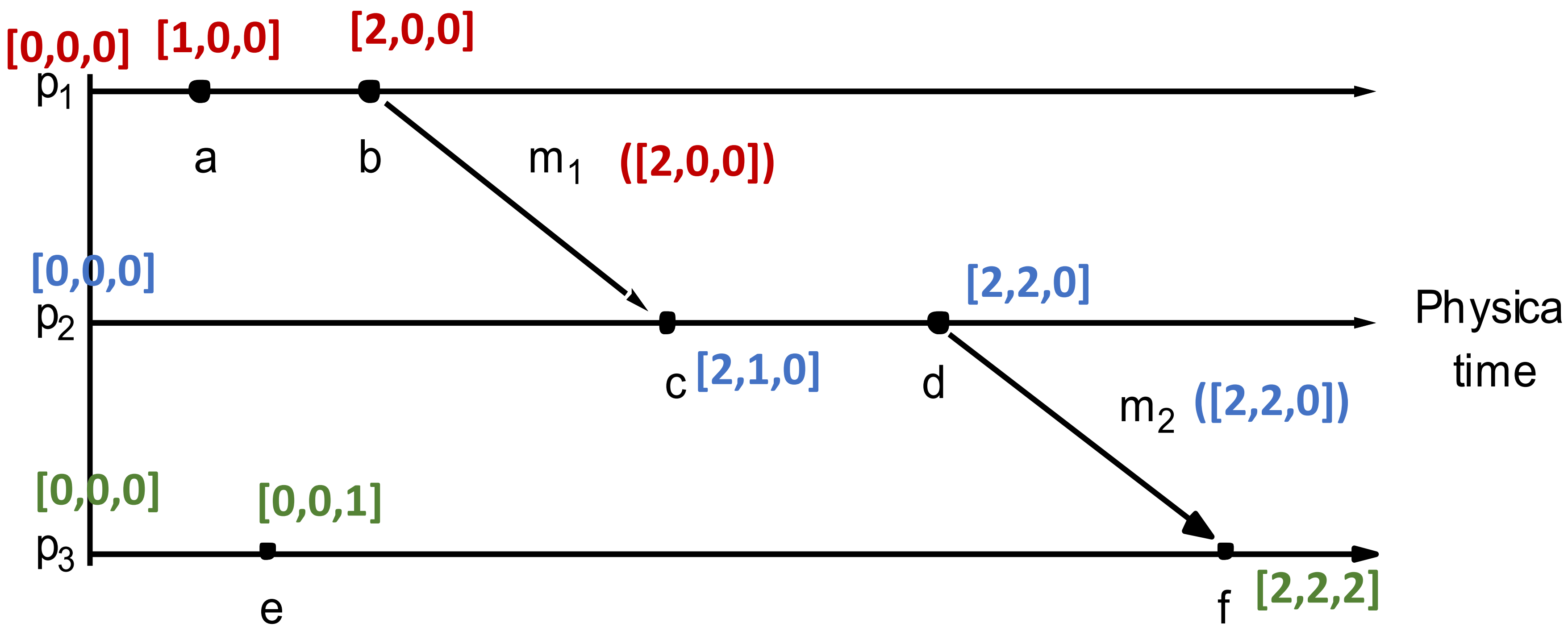
What can we say about e & f based on their vector timestamps?

Vector Timestamps: Example



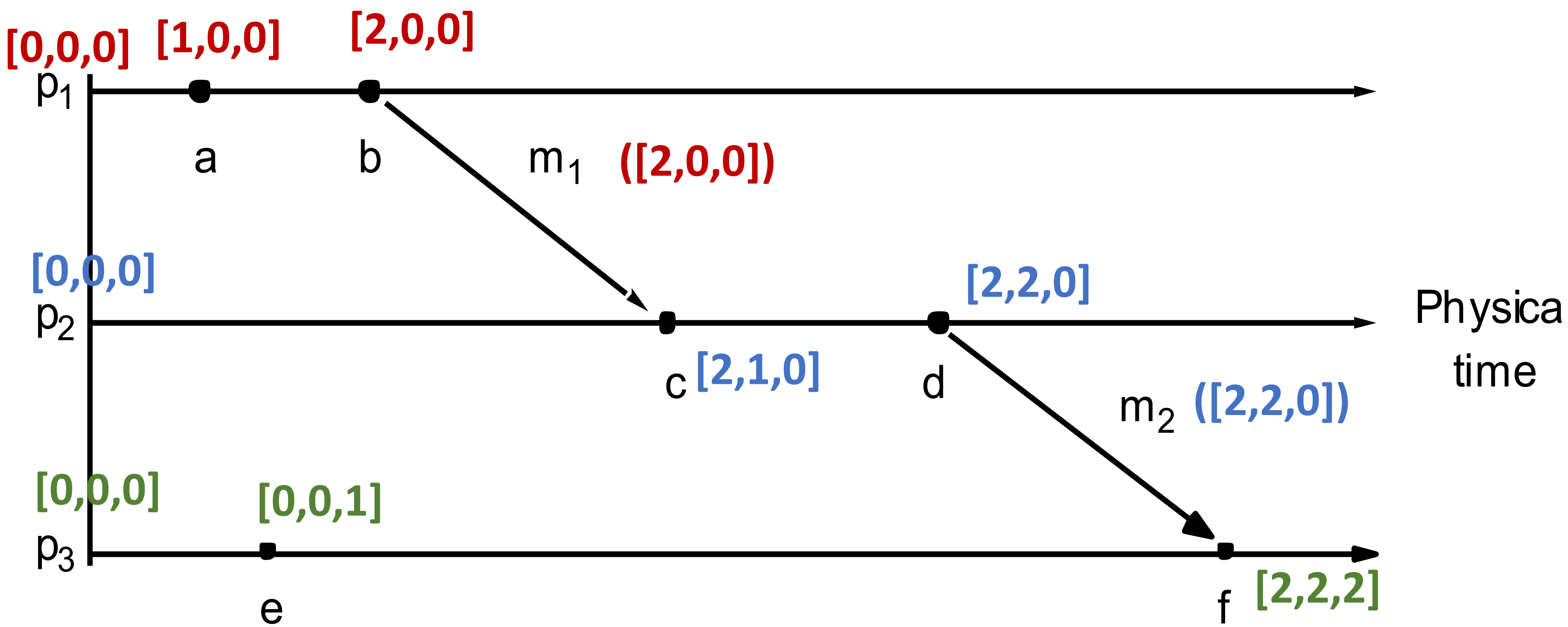
$V(e) < V(f)$, $e \rightarrow f$

Vector Timestamps: Example



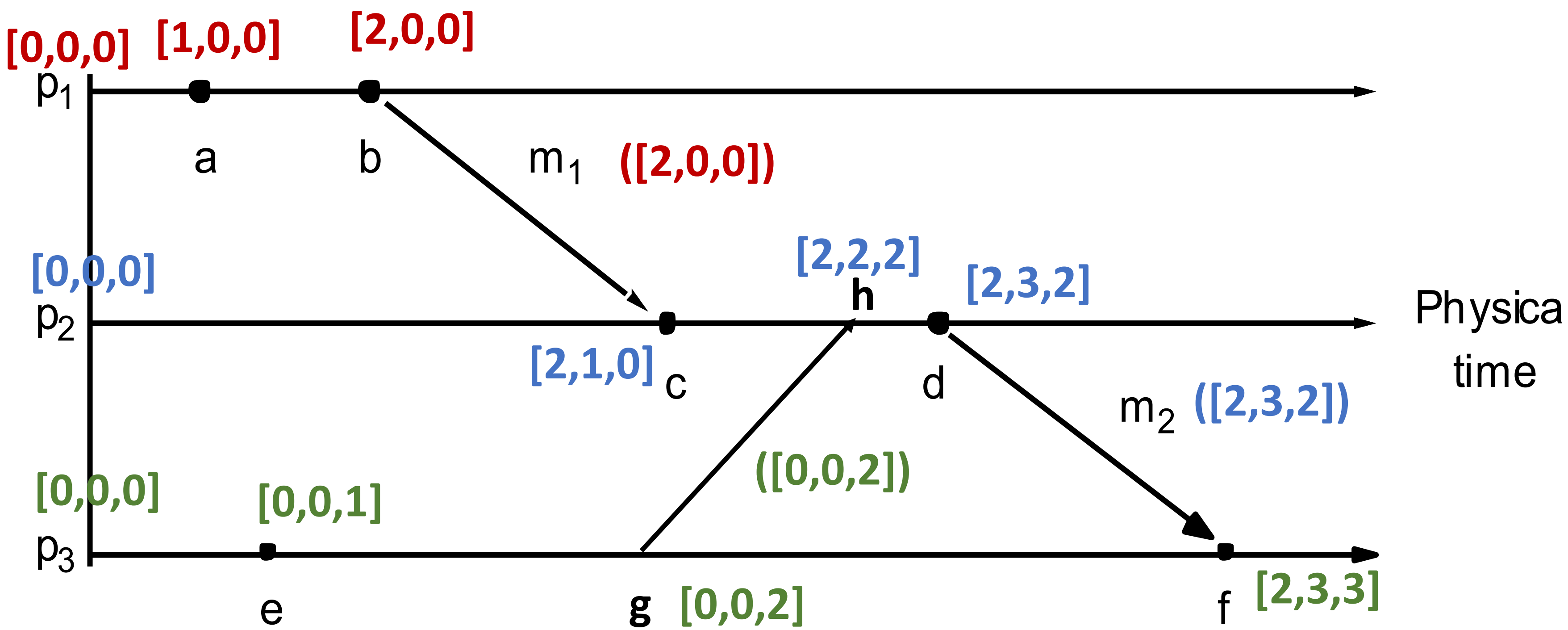
What can we say about e & d based on their vector timestamps?

Vector Timestamps: Example



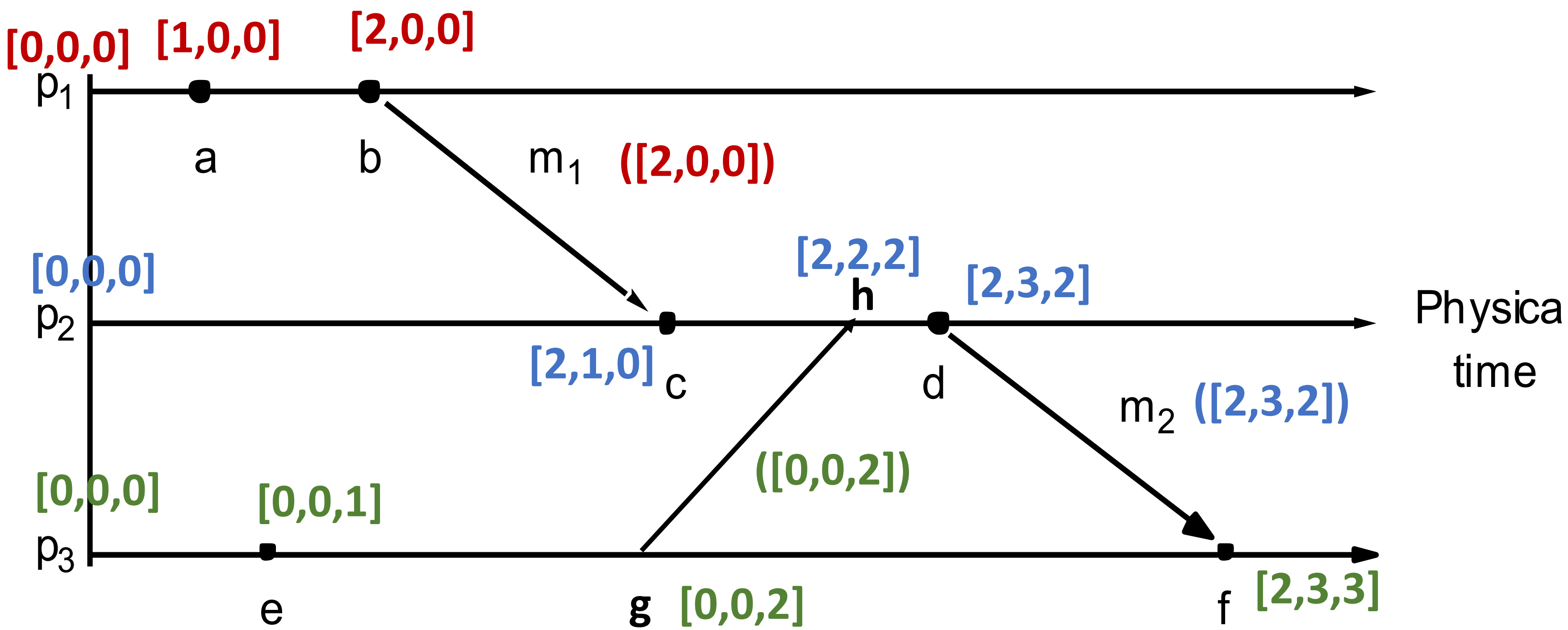
$V(e) \nless V(d)$ and $V(d) \nless V(e)$, $e \parallel d$

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.