

# Distributed Systems

ECE428

Lecture 4

*Adopted from Spring 2021*

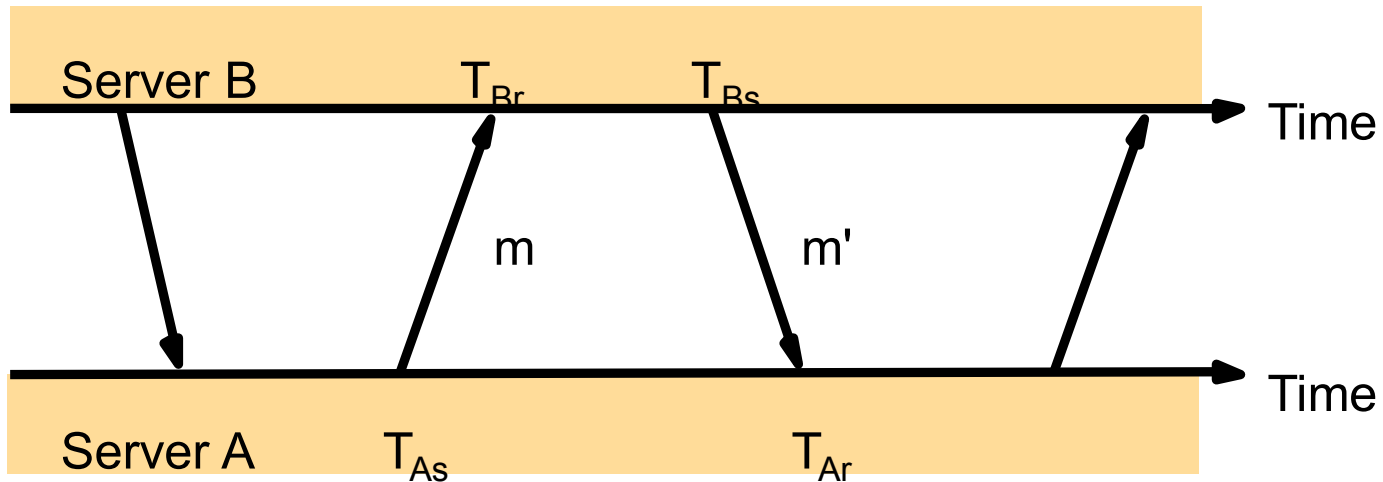
# Something to think while we wait...

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

# 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 =  $(T_{\text{round}} / 2) - \min \leq T_{\text{round}} / 2$
  - 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)
- $o$ : true offset of clock at B relative to clock at A (unknown)
- $o_i$ : estimate of actual offset between the two clocks
- $d_i$ : estimate of accuracy of  $o_i$  ;  $d_i = t + t'$
- skew estimate =  $d_i / 2$

$$T_{Br} = T_{As} + t + o$$

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

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

$$o_i = ((T_{Br} - T_{As}) - (T_{Ar} - T_{Bs})) / 2$$

$$o = o_i + (t' - t) / 2$$

$$d_i = t + t' = (T_{Br} - T_{As}) + (T_{Ar} - T_{Bs})$$

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

# 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 use-case 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:  
 $\langle p_1, p_2, p_3, \dots, p_n \rangle$
- Each process  $p_i$  is described by its *state*  $s_i$  that gets transformed over time.
  - State includes values of all local variables, affected files, etc.
- $s_i$  gets transformed when an *event* occurs.
- Three types of events:
  - Local computation.
  - Sending a message.
  - Receiving a message.



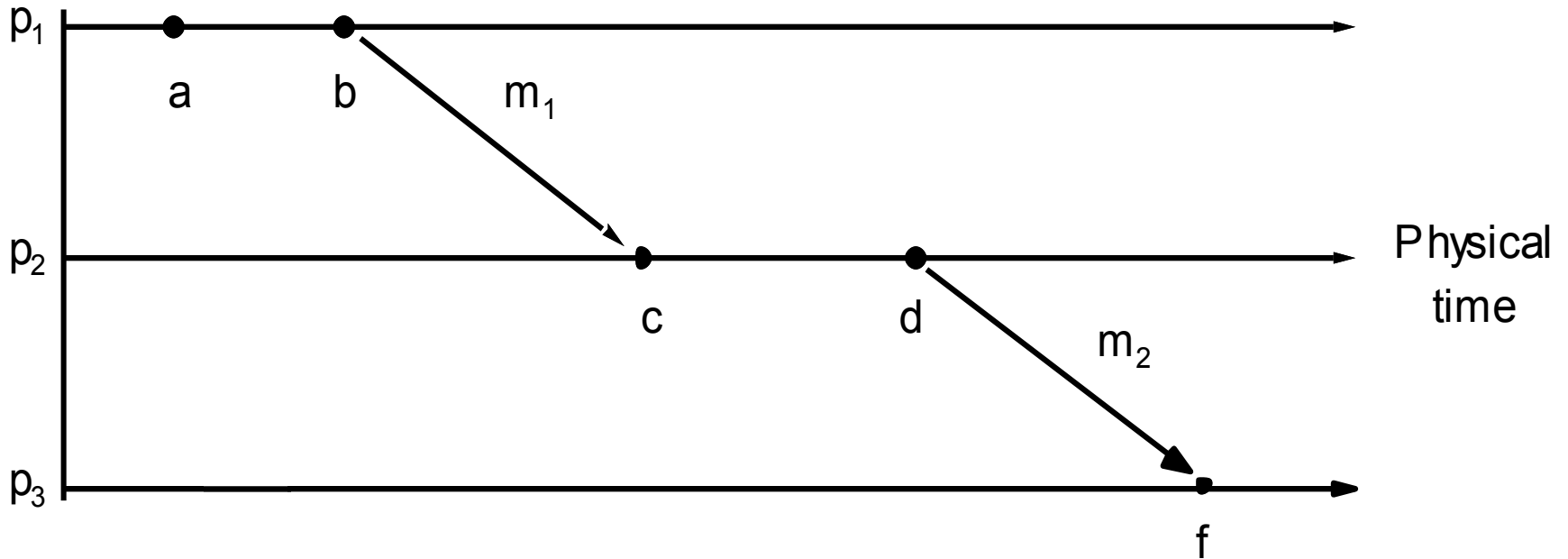
# Event Ordering

- Easy to order events within a single process  $p_i$ , 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  $\rightarrow$ .
  - $e \rightarrow e'$  means *e happened before e'*.

# Happened-Before Relationship

- *Happened-before* (HB) relationship denoted by  $\rightarrow$ .
  - $e \rightarrow e'$  means  $e$  *happened before*  $e'$ .
  - $e \rightarrow_i e'$  means  $e$  *happened before*  $e'$ , as observed by  $p_i$ .
- HB rules:
  - If  $\exists p_i$ ,  $e \rightarrow_i e'$  then  $e \rightarrow e'$ .
  - For any message  $m$ ,  $\text{send}(m) \rightarrow \text{receive}(m)$
  - If  $e \rightarrow e'$  and  $e' \rightarrow e''$  then  $e \rightarrow e''$
- Also called “*causal*” or “*potentially causal*” ordering.

# Event Ordering: Example

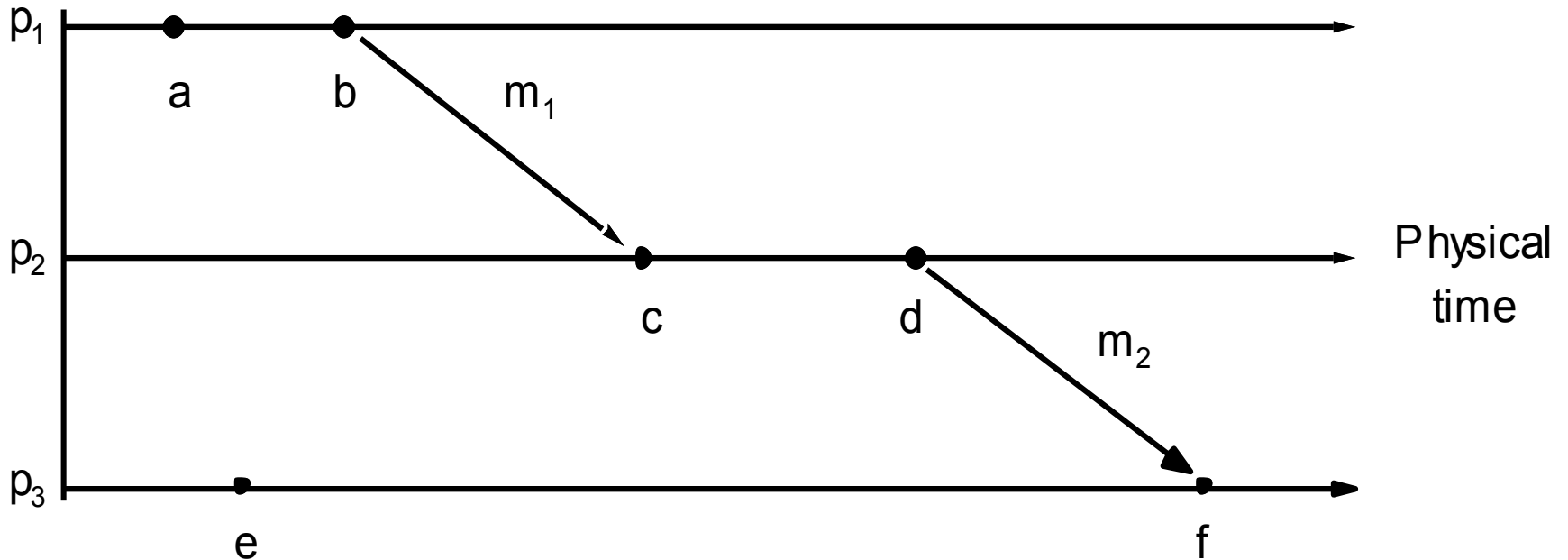


Which event happened first?

$a \rightarrow b$  and  $b \rightarrow c$  and  $c \rightarrow d$  and  $d \rightarrow f$

$a \rightarrow b$  and  $a \rightarrow c$  and  $a \rightarrow d$  and  $a \rightarrow f$

# Event Ordering: Example



What can we say about  $e$ ?

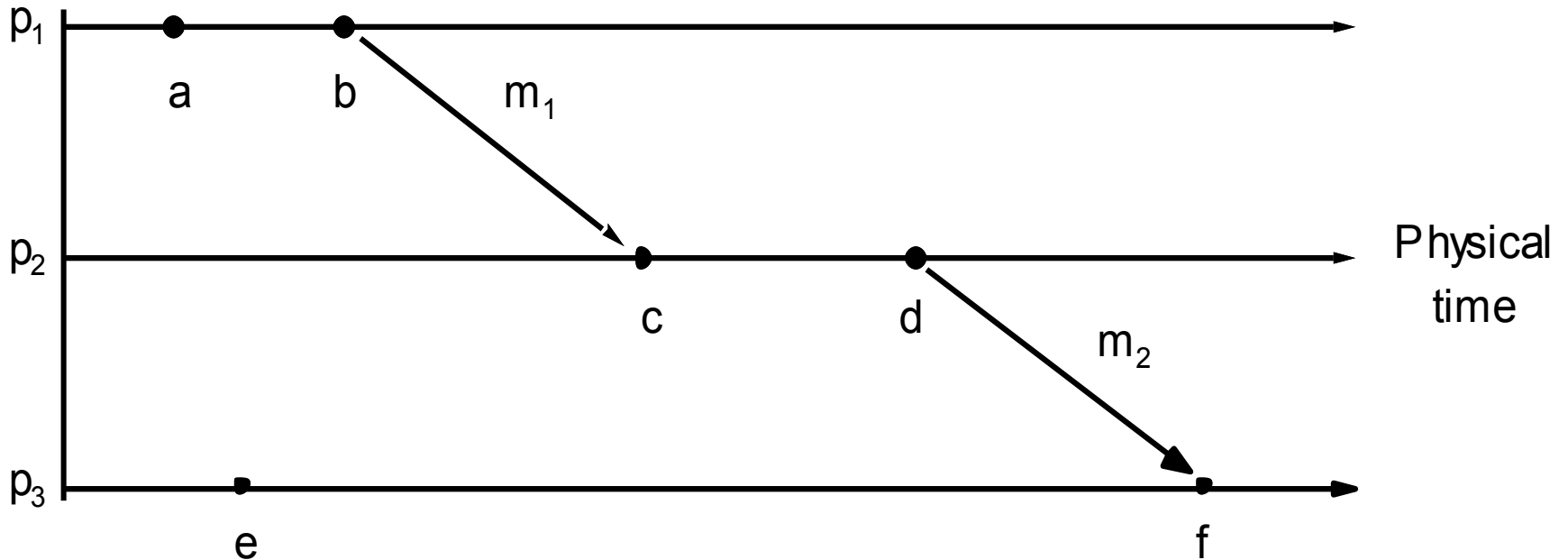
$e \rightarrow f$

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

$a \parallel e$

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

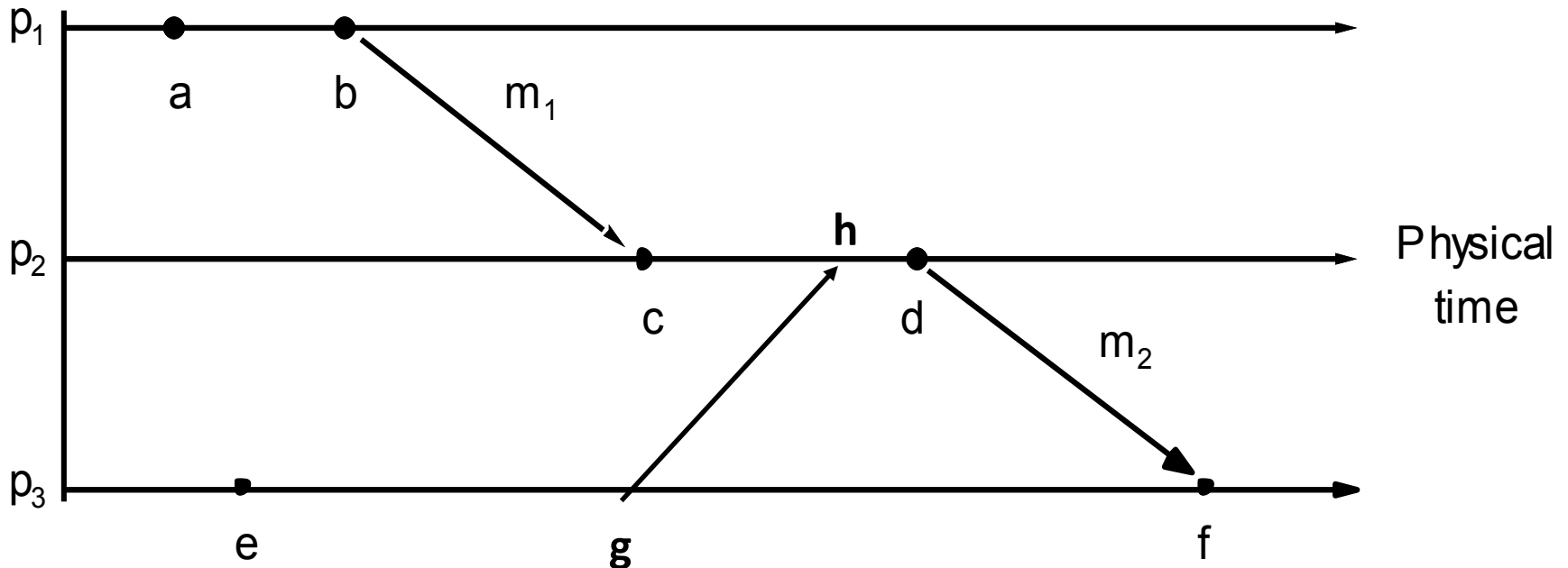
# Event Ordering: Example



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

$e \parallel d$

# Logical Timestamps: Example



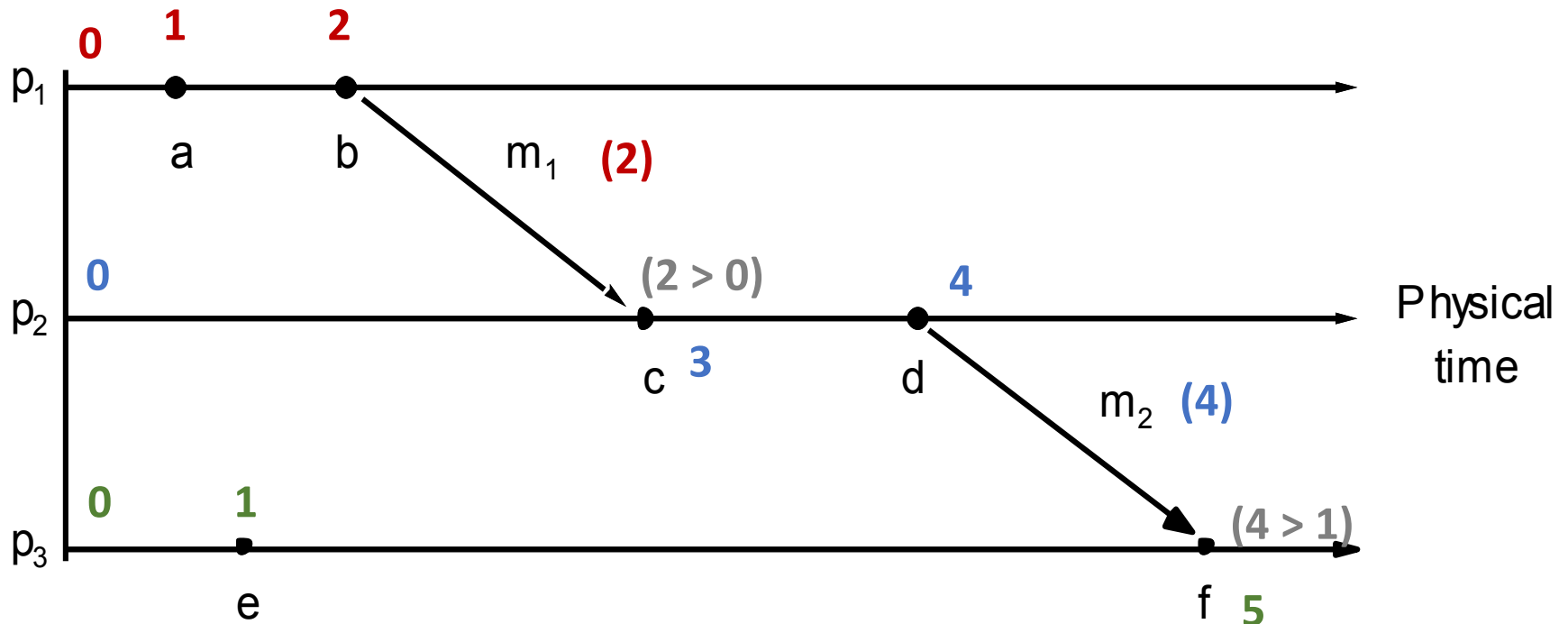
What can we say about  $e$  and  $d$ ?

$e \rightarrow d$

# Lamport's Logical Clock

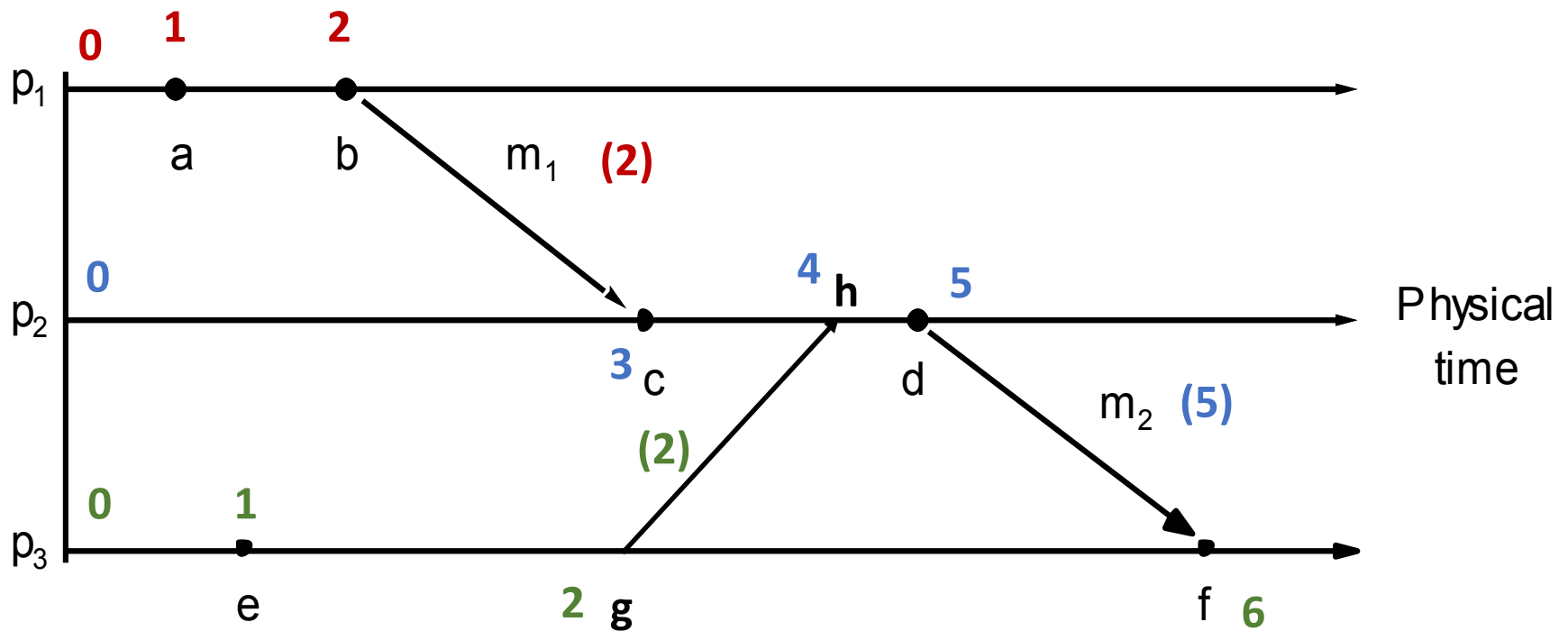
- Logical timestamp for each event that captures the *happened-before* relationship.
- *Algorithm:* Each process  $p_i$ 
  1. initializes local clock  $L_i = 0$ .
  2. increments  $L_i$  before timestamping each event.
  3. piggybacks  $L_i$  when sending a message.
  4. upon receiving a message with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

# Logical Timestamps: Example





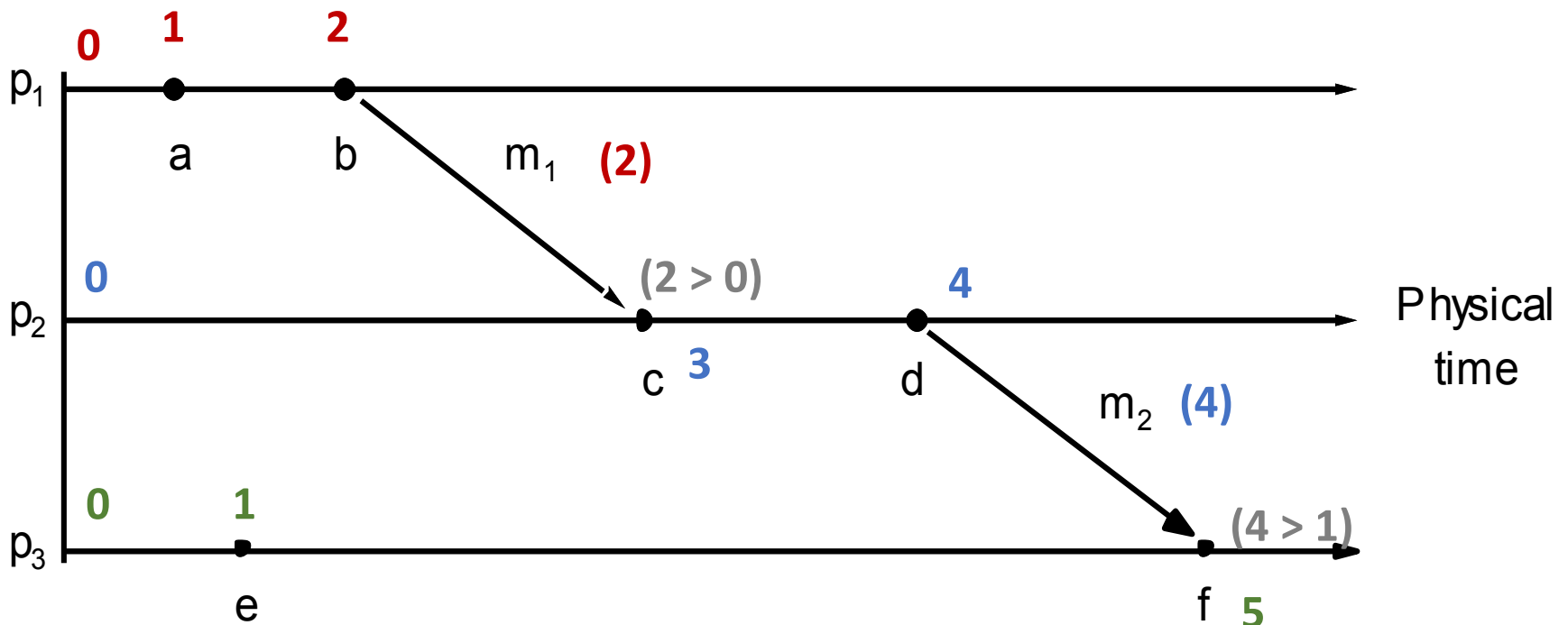
# Logical Timestamps: Example



# 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:  $e' \nrightarrow e$
  - Either  $e \rightarrow e'$  or  $e \parallel e'$

# Logical Timestamps: Example



$L(e) < L(d)$ ,  $e \parallel d$

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

# 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  $V_i[i] = 0$
  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

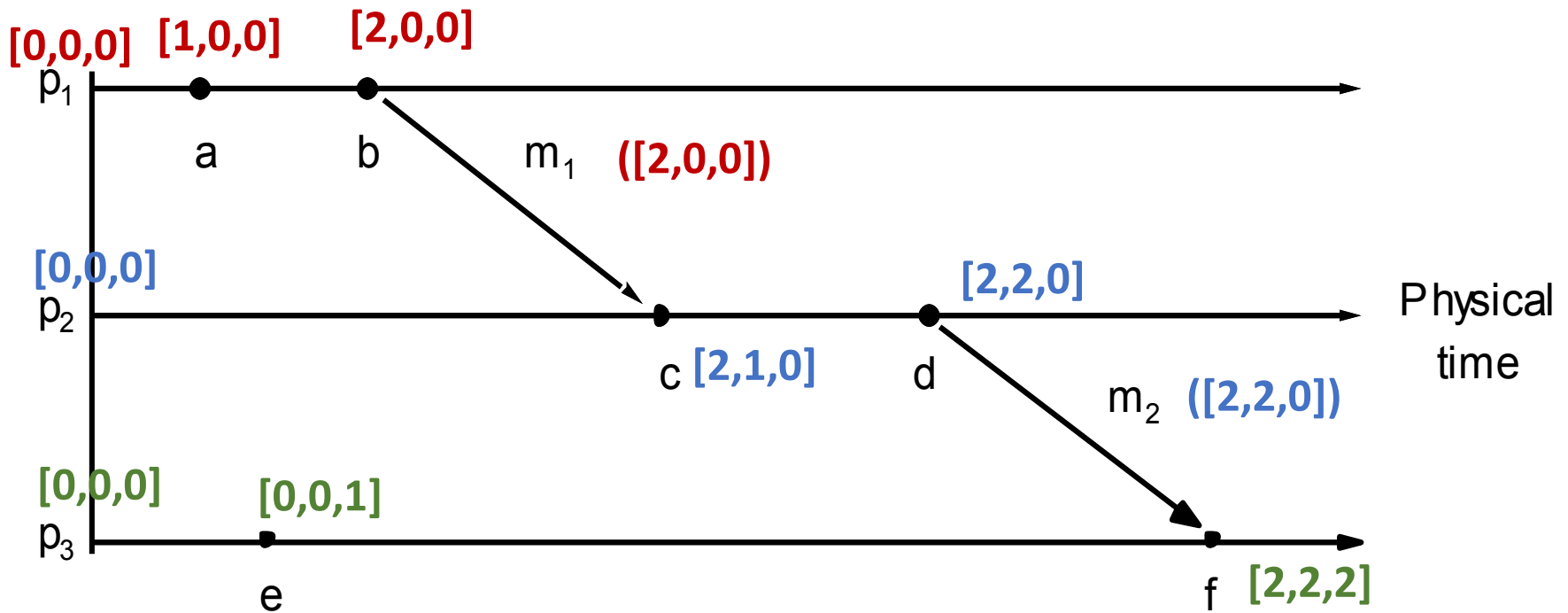
- Each event associated with a vector timestamp.
- Each process  $p_i$  maintains vector of clocks  $V_i$
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$  is the clock for process  $p_j$  as maintained by  $p_i$
- Algorithm: each process  $p_i$ :
  1. initializes local clock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

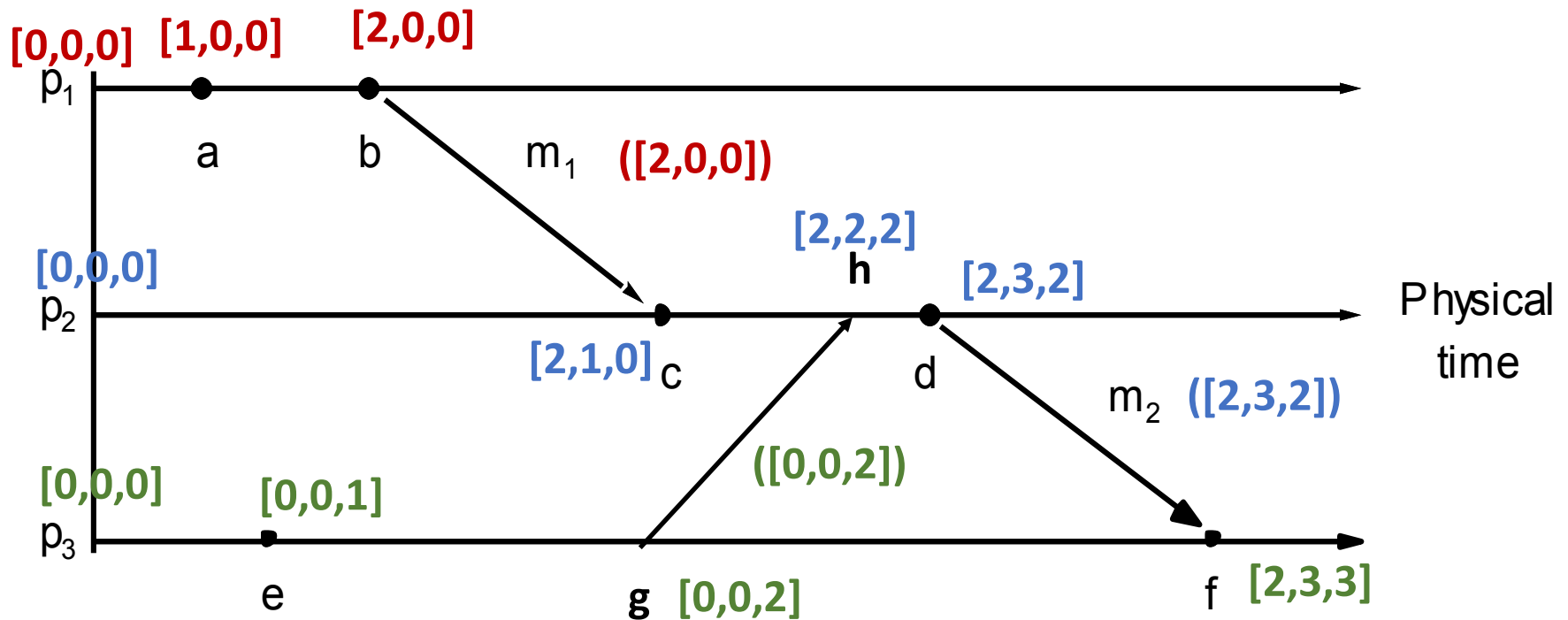
- Each event associated with a vector timestamp.
- Each process  $p_i$  maintains vector of clocks  $V_i$
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$  is the clock for process  $p_j$  as maintained by  $p_i$
- Algorithm: each process  $p_i$ :
  1. initializes local clock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



# Vector Timestamps: Example



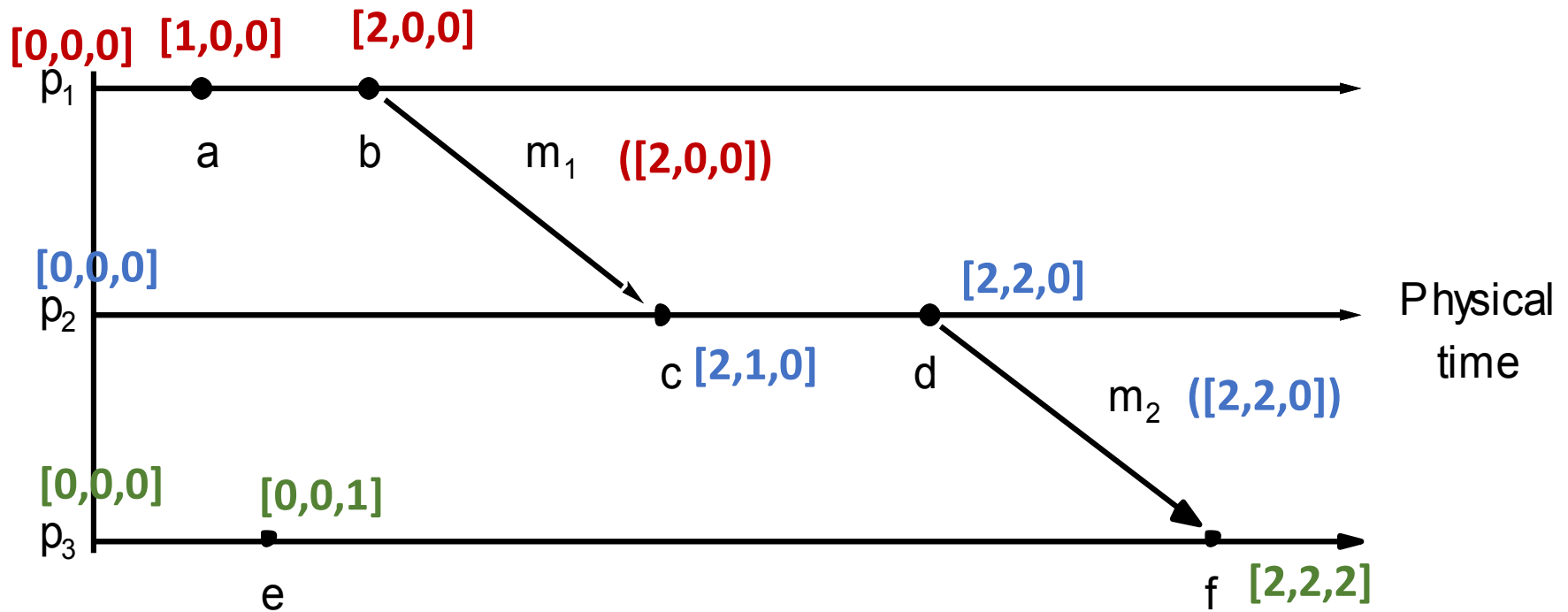
# Vector Timestamps: Example



# Comparing Vector Timestamps

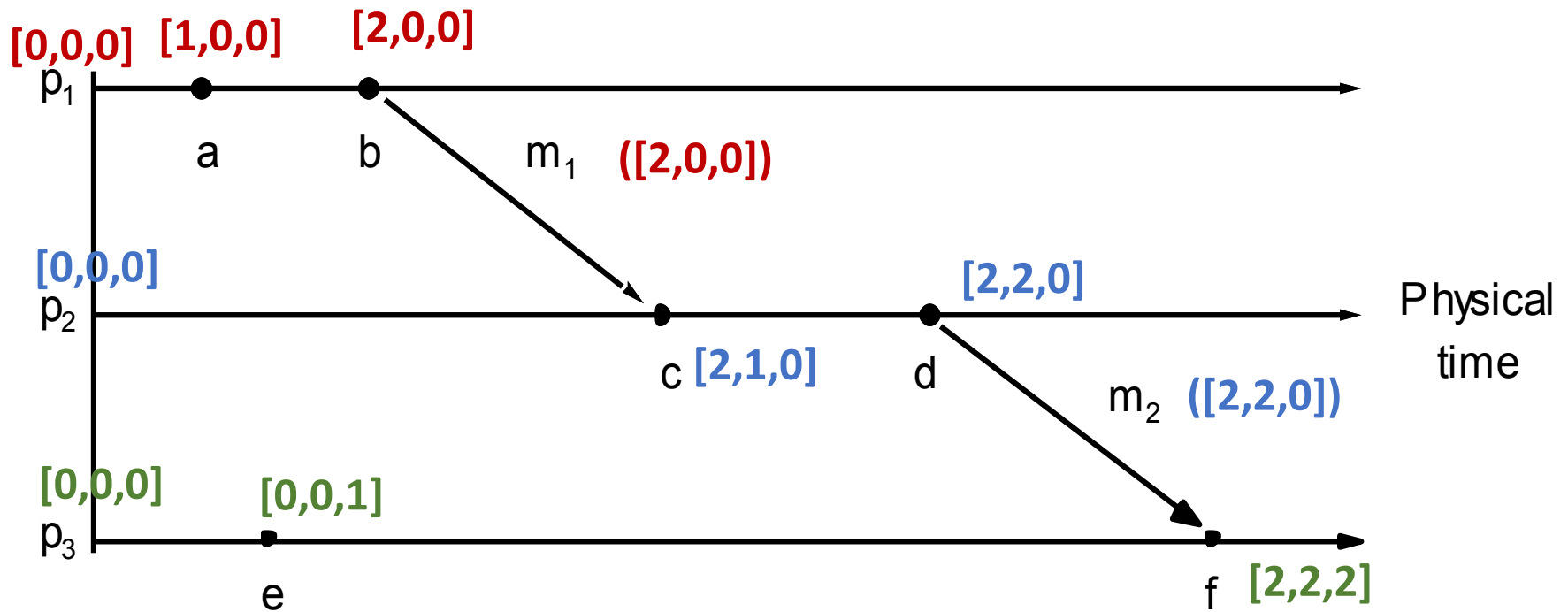
- Let  $V(e) = V$  and  $V(e') = V'$
- $V = V'$ , iff  $V[i] = V'[i]$ , for all  $i = 1, \dots, n$
- $V \leq V'$ , iff  $V[i] \leq V'[i]$ , for all  $i = 1, \dots, n$
- $V < V'$ , iff  $V \leq V' \ \& \ V \neq V'$   
iff  $V \leq V' \ \& \ \exists j \text{ such that } (V[j] < V'[j])$
- $e \rightarrow e'$  iff  $V < V'$ 
  - $(V < V' \text{ implies } e \rightarrow e') \text{ and } (e \rightarrow e' \text{ implies } V < V')$
- $e \parallel e'$  iff  $(V \not< V' \text{ and } V' \not< V)$

# Vector Timestamps: Example



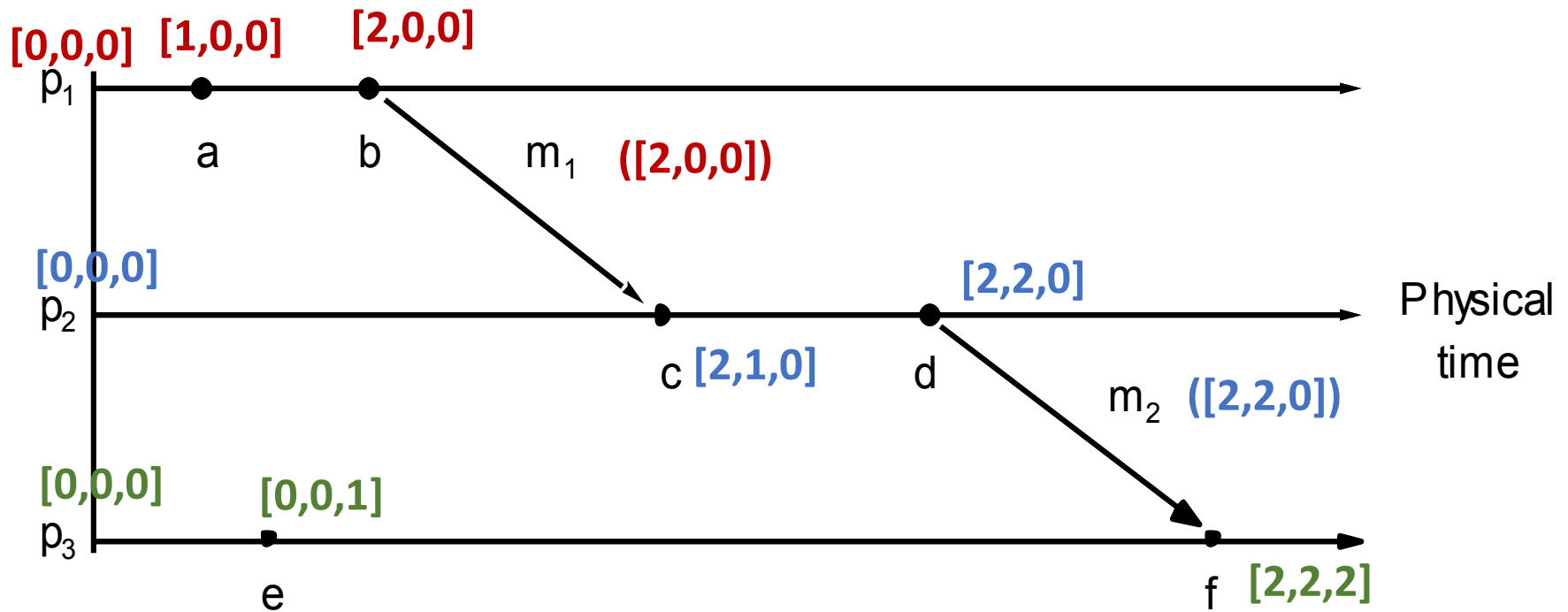
What can we say about  $e$  &  $f$  based on their vector timestamps?

# Vector Timestamps: Example



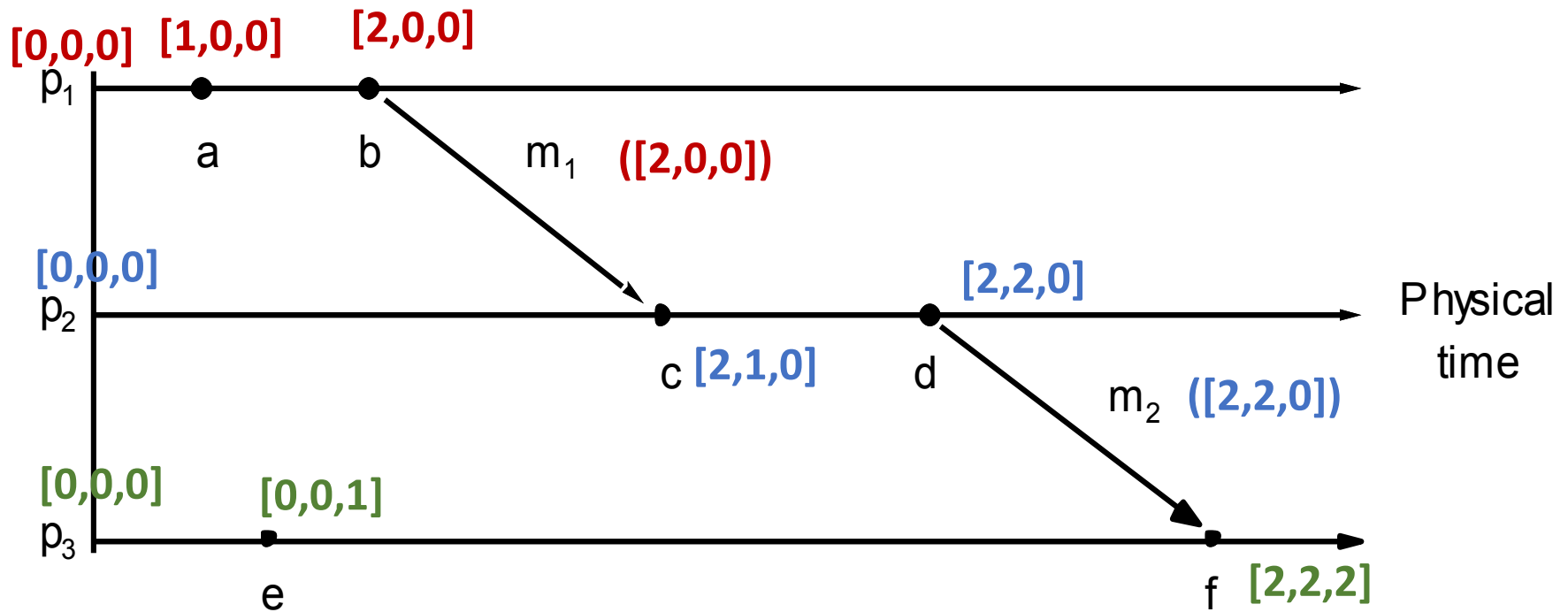
$$V(e) < V(f), \text{ } e \rightarrow f$$

# Vector Timestamps: Example



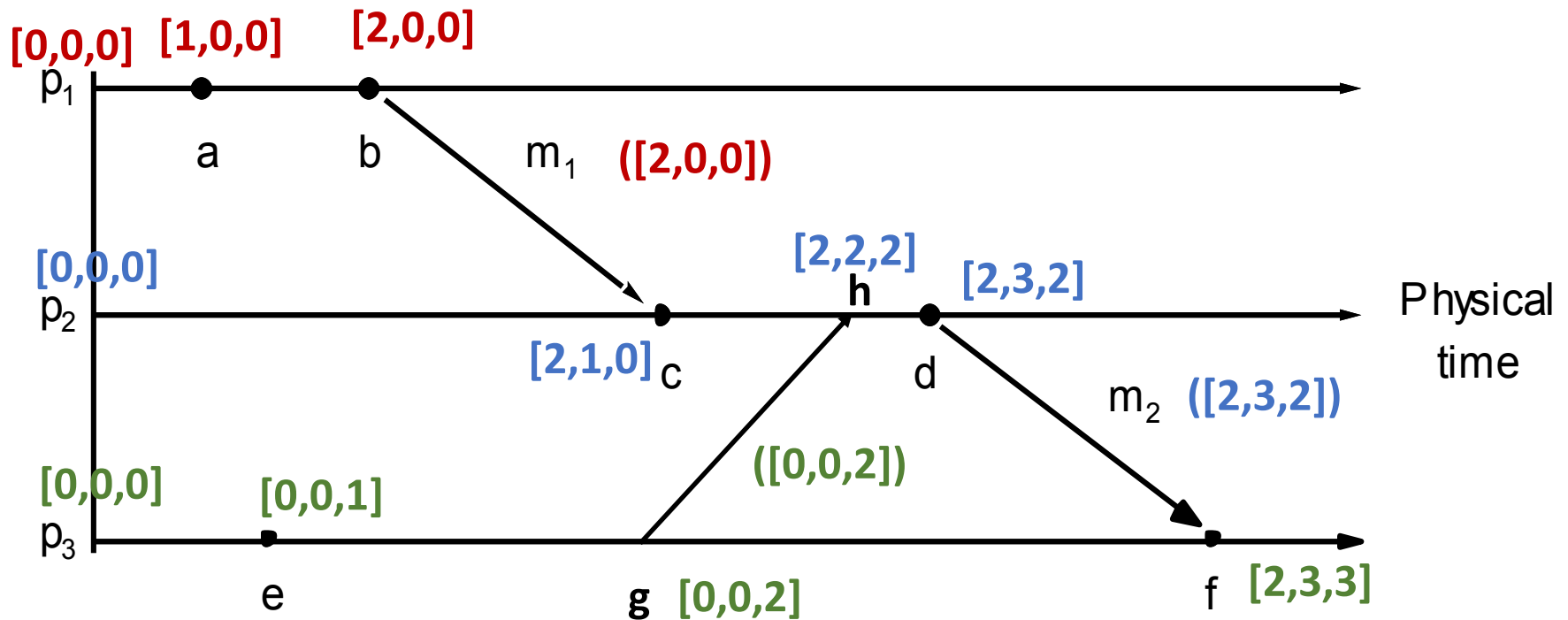
What can we say about  $e$  &  $d$  based on their vector timestamps?

# Vector Timestamps: Example



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

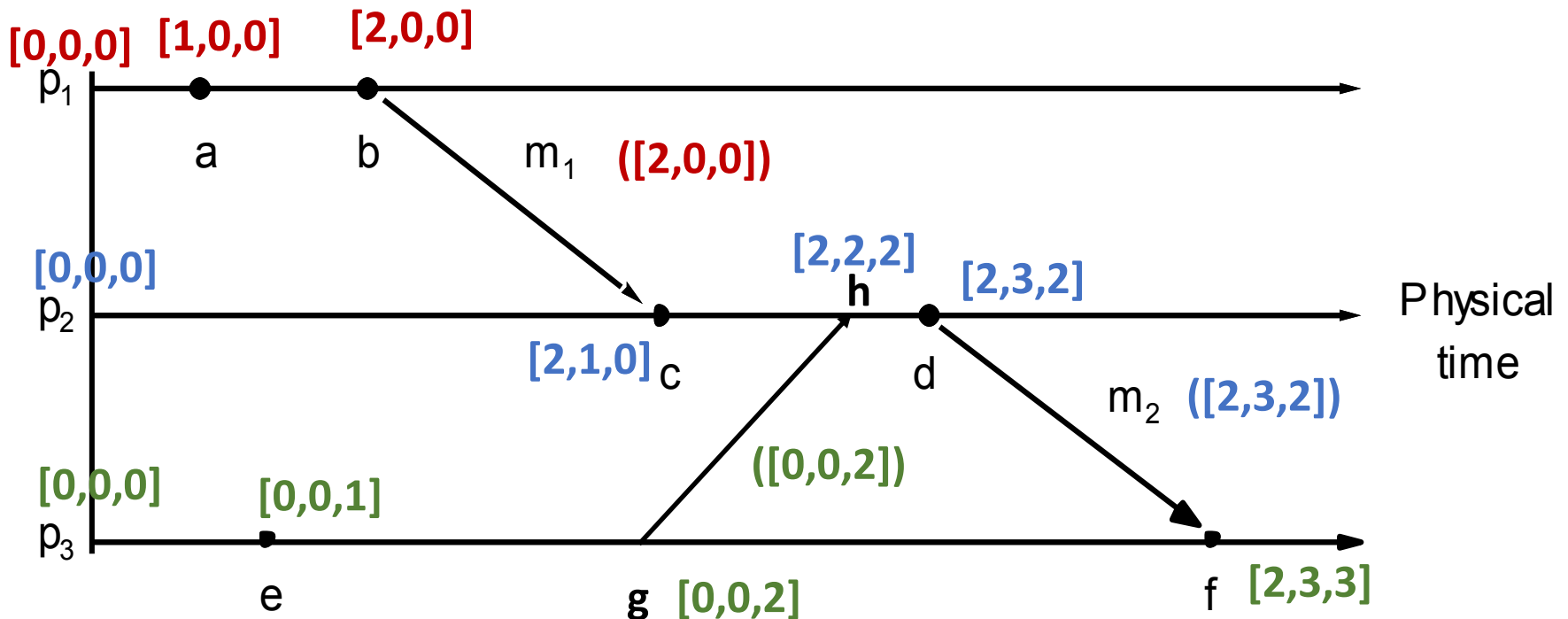
# Vector Timestamps: Example



How about now?



# Vector Timestamps: Example



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

# 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.