

# Time & Clocks

*CS 475: Concurrent & Distributed Systems (Fall 2021)*

Lecture 6

Yue Cheng

Some material taken/derived from:

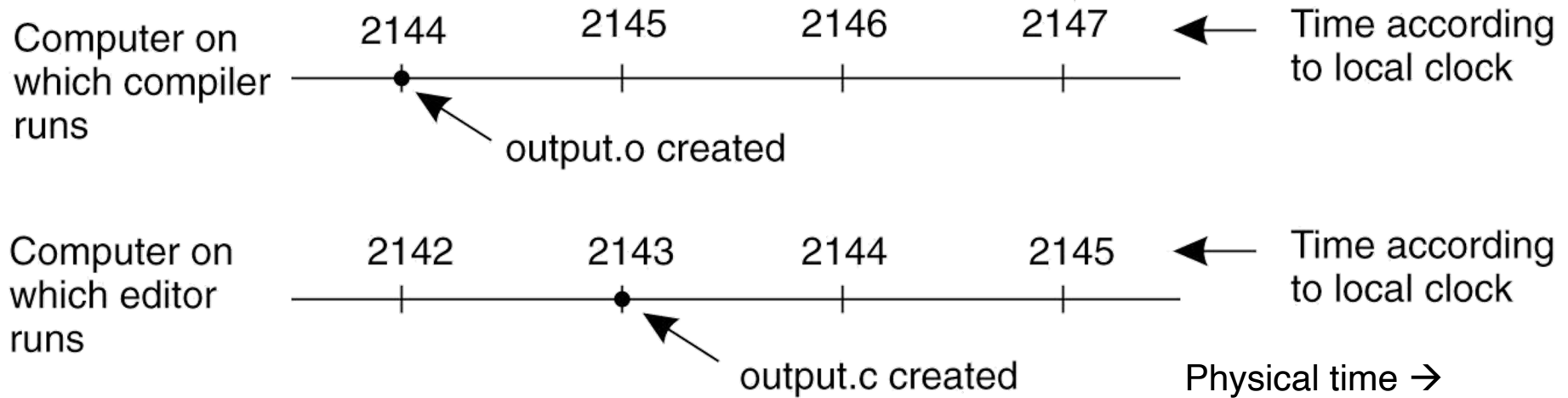
- Princeton COS-418 materials created by Michael Freedman and Wyatt Lloyd.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.

Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

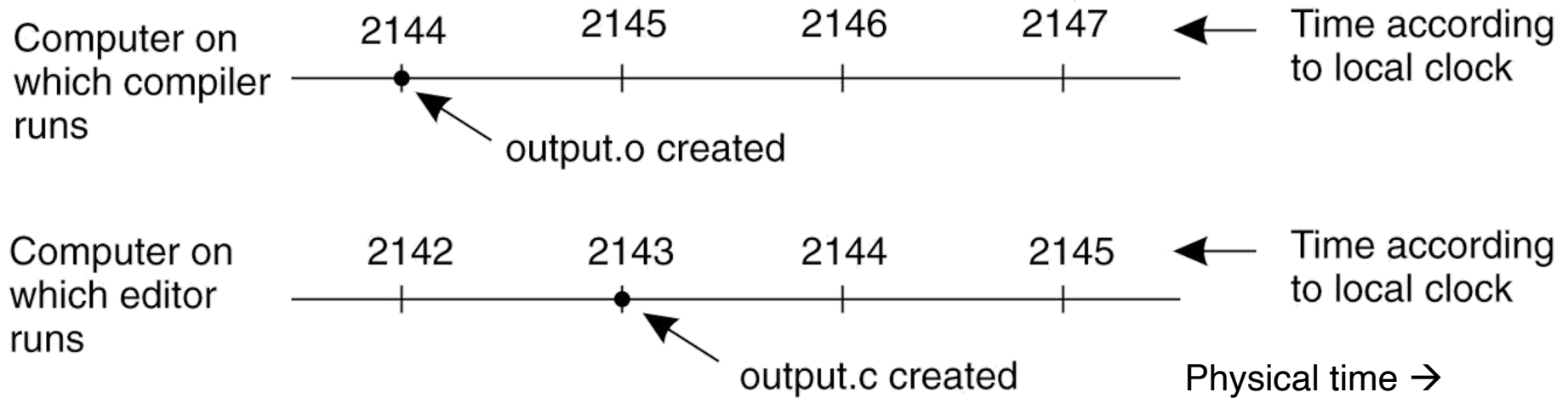
# Today's outline

- The need for time synchronization
- “Wall clock time” synchronization
- Logical Time: Lamport Clocks
- Vector clocks

# A distributed edit-compile workflow

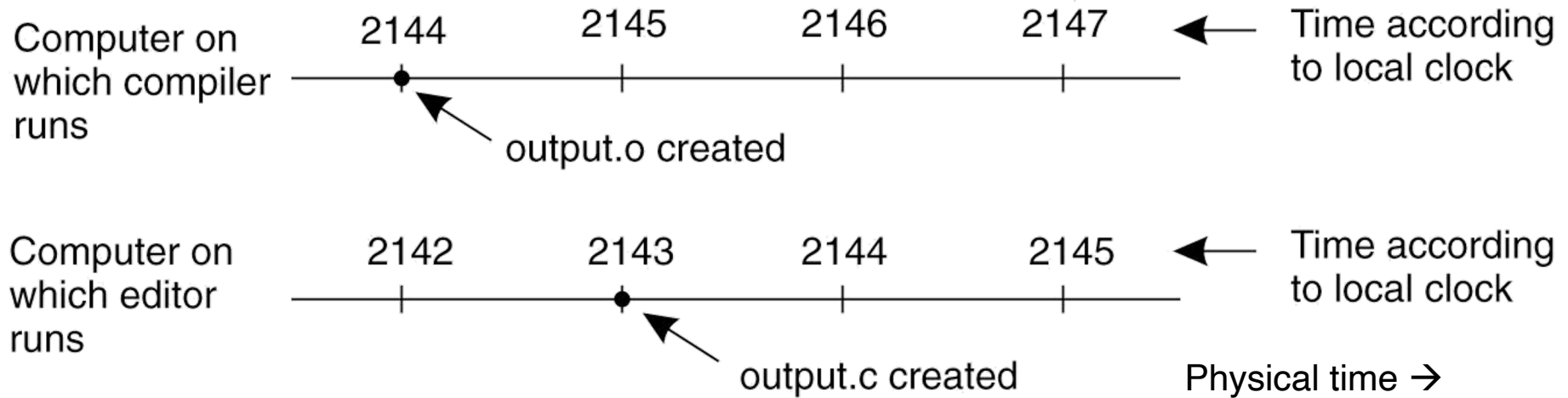


# A distributed edit-compile workflow



- $2143 < 2144 \Rightarrow$  make doesn't call compiler

# A distributed edit-compile workflow



- $2143 < 2144 \Rightarrow$  make doesn't call compiler

Lack of time synchronization result –  
possible object file mismatch

# What makes time synchronization hard?

1. Quartz oscillator sensitive to temperature, age, vibration, radiation
  - Accuracy ~one part per million
    - (one second of clock drift over 12 days)
2. The internet is:
  - Asynchronous: arbitrary message delays
  - Best-effort: messages don't always arrive

# Today's outline

- The need for time synchronization
- “Wall clock time” synchronization
  - Cristian's algorithm, NTP
- Logical Time: Lamport Clocks
- Vector clocks

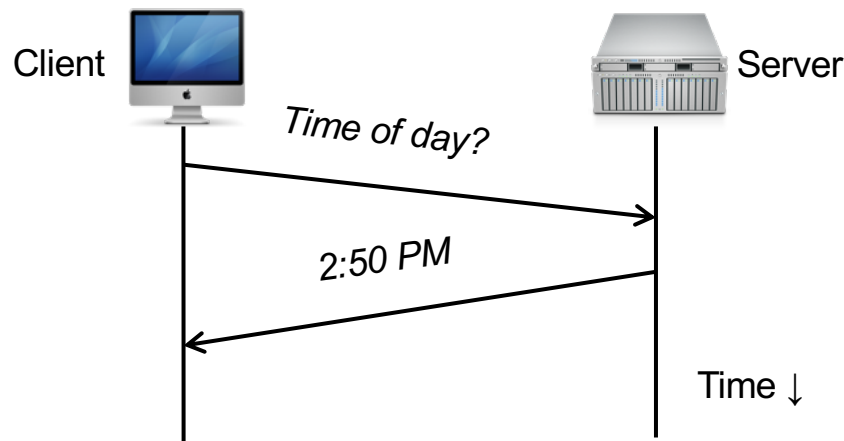
# Just use Coordinated Universal Time?

- UTC is broadcast from radio stations on land and satellite (e.g., the Global Positioning System)
  - Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1–10 milliseconds
- Signals from GPS are accurate to about one microsecond
  - *Why can't we put GPS receivers on all our computers?*



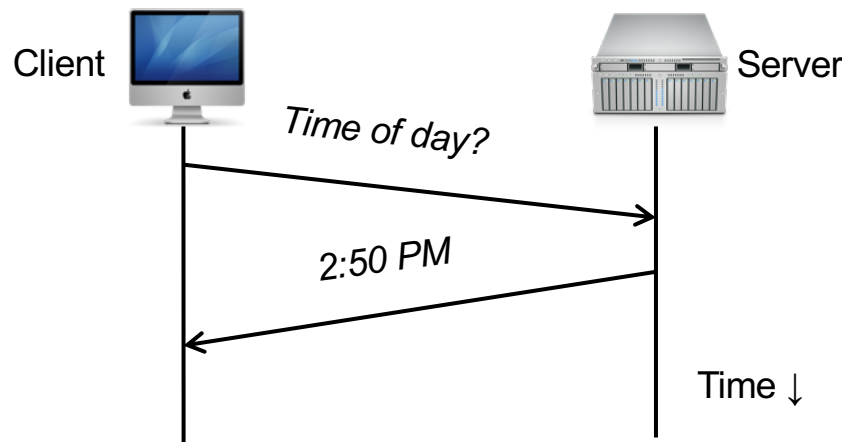
# Synchronization to a time server

- Suppose a server with an accurate clock (e.g., GPS-receiver)
  - Could simply issue an RPC to obtain the time:



# Synchronization to a time server

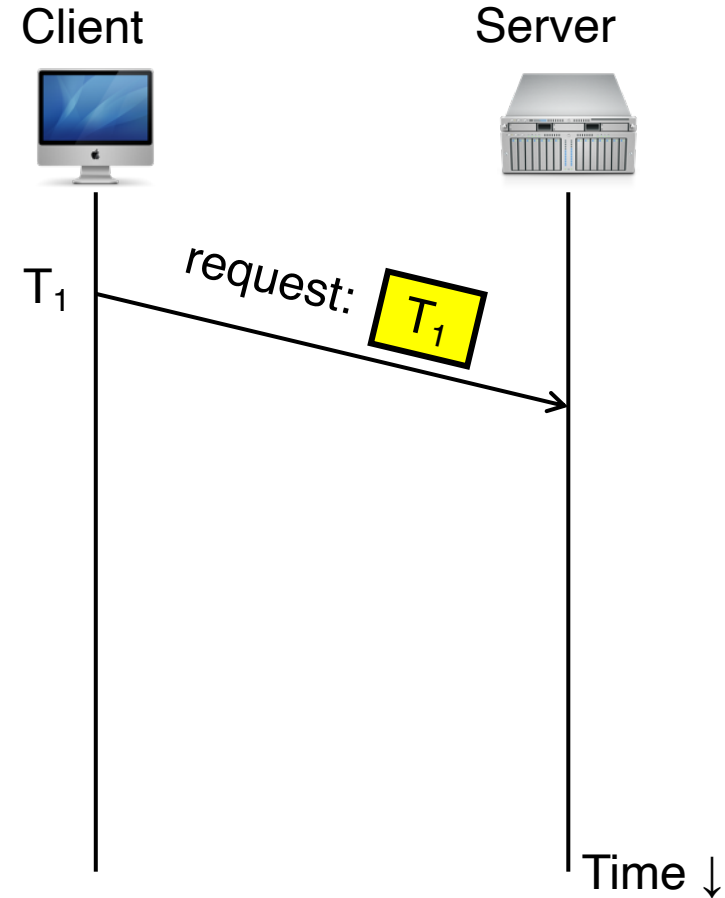
- Suppose a server with an accurate clock (e.g., GPS-receiver)
  - Could simply issue an RPC to obtain the time:



- But this doesn't account for network latency
  - Message delays will have **outdated** server's answer

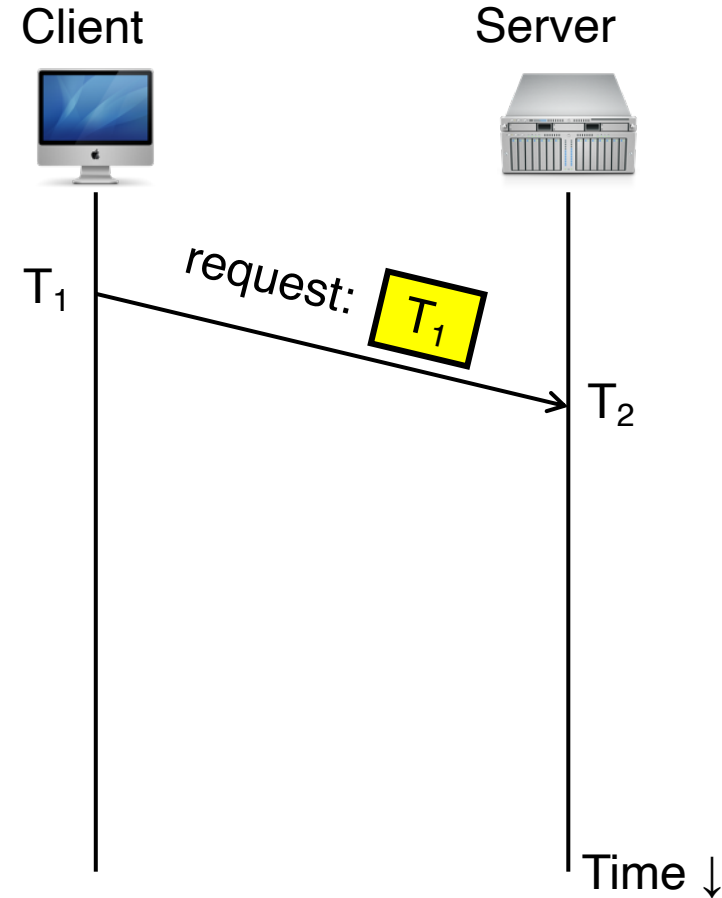
# Cristian's algorithm: Outline

1. Client sends a **request** packet, timestamped with its local clock  $T_1$



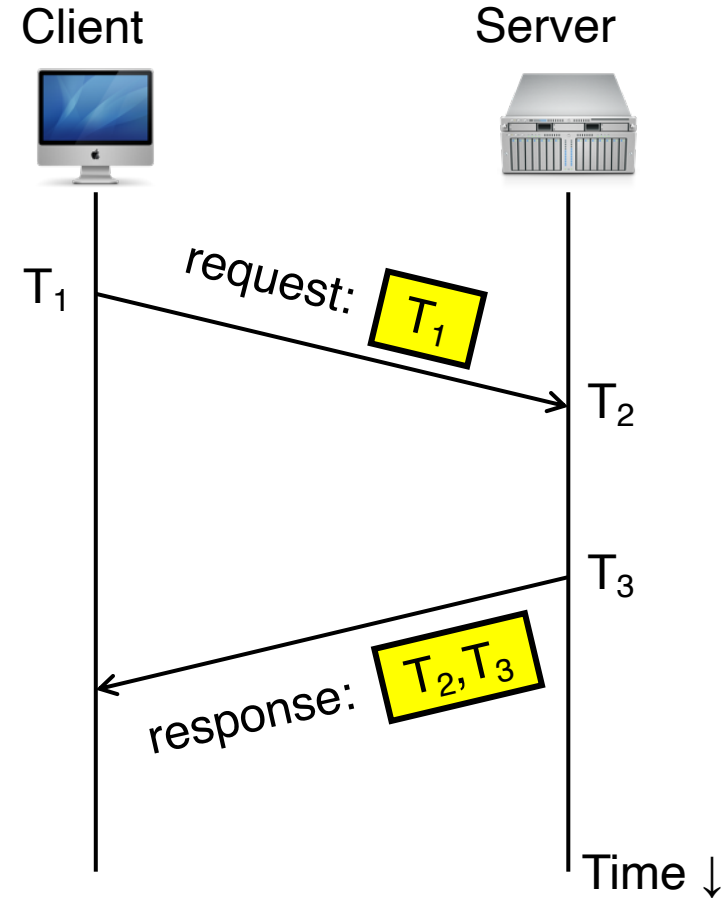
# Cristian's algorithm: Outline

1. Client sends a request packet, timestamped with its local clock  $T_1$
2. Server timestamps its receipt of the request  $T_2$  with its local clock



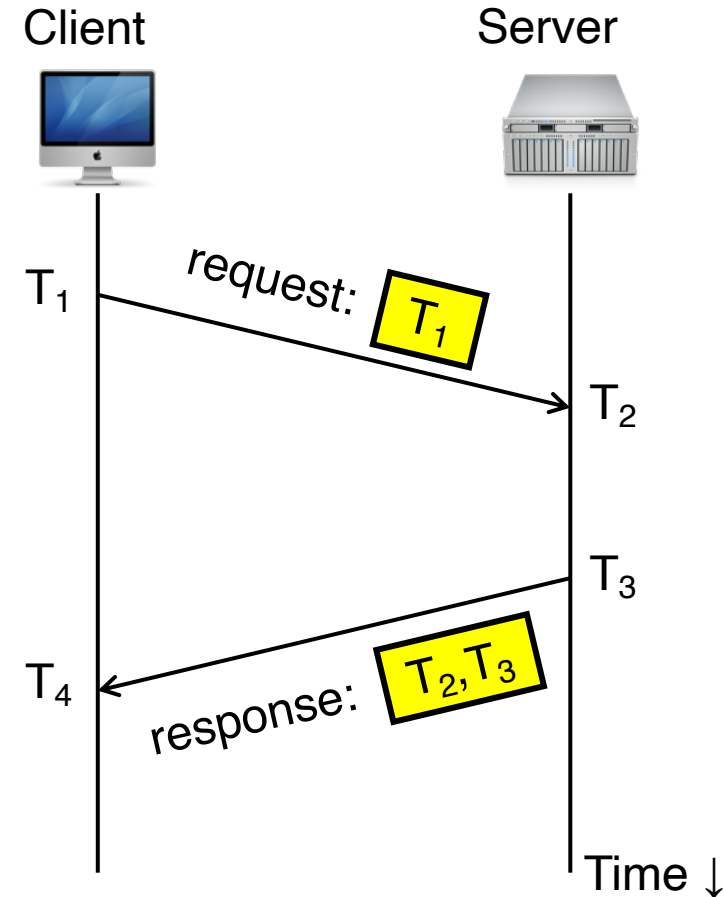
# Cristian's algorithm: Outline

1. Client sends a request packet, timestamped with its local clock  $T_1$
2. Server timestamps its receipt of the request  $T_2$  with its local clock
3. Server sends a **response** packet with its local clock  $T_3$  and  $T_2$



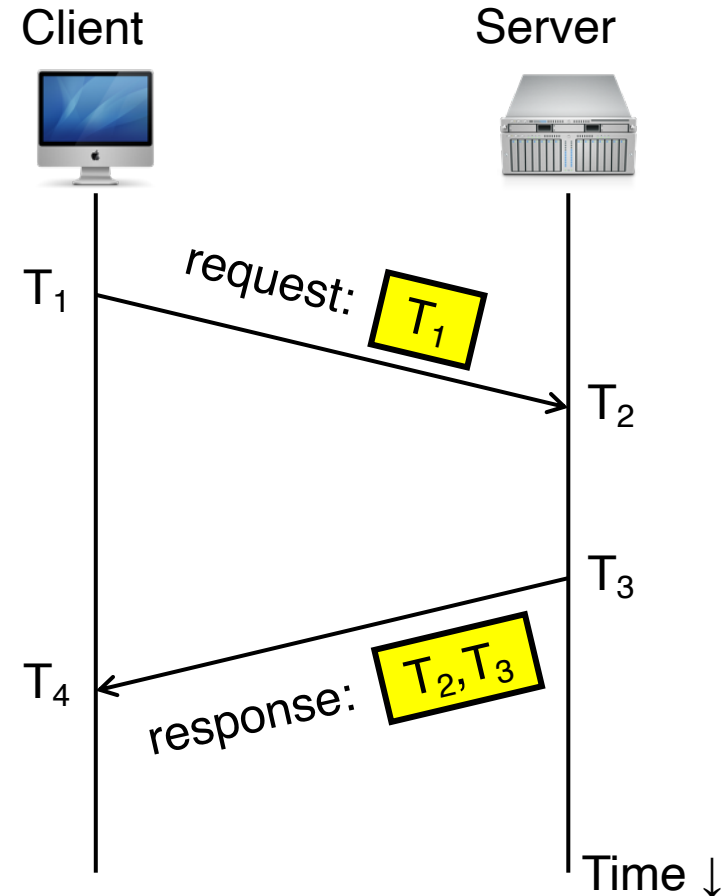
# Cristian's algorithm: Outline

1. Client sends a request packet, timestamped with its local clock  $T_1$
2. Server timestamps its receipt of the request  $T_2$  with its local clock
3. Server sends a response packet with its local clock  $T_3$  and  $T_2$
4. Client locally timestamps its receipt of the server's response  $T_4$



# Cristian's algorithm: Outline

1. Client sends a request packet, timestamped with its local clock  $T_1$
2. Server timestamps its receipt of the request  $T_2$  with its local clock
3. Server sends a response packet with its local clock  $T_3$  and  $T_2$
4. Client locally timestamps its receipt of the server's response  $T_4$

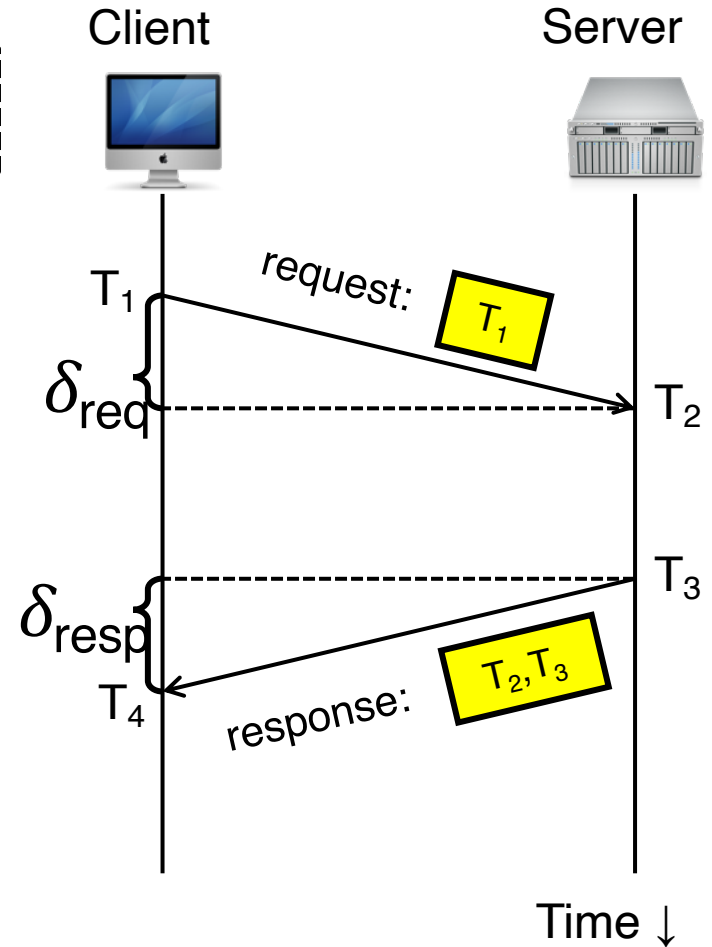


**How can the client use these timestamps to synchronize its local clock to the server's local clock?**

# Cristian's algorithm: Offset sample calculation

Goal: Client sets clock  $\leftarrow T_3 + \delta_{\text{resp}}$

- Client samples round trip time  $\delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2)$

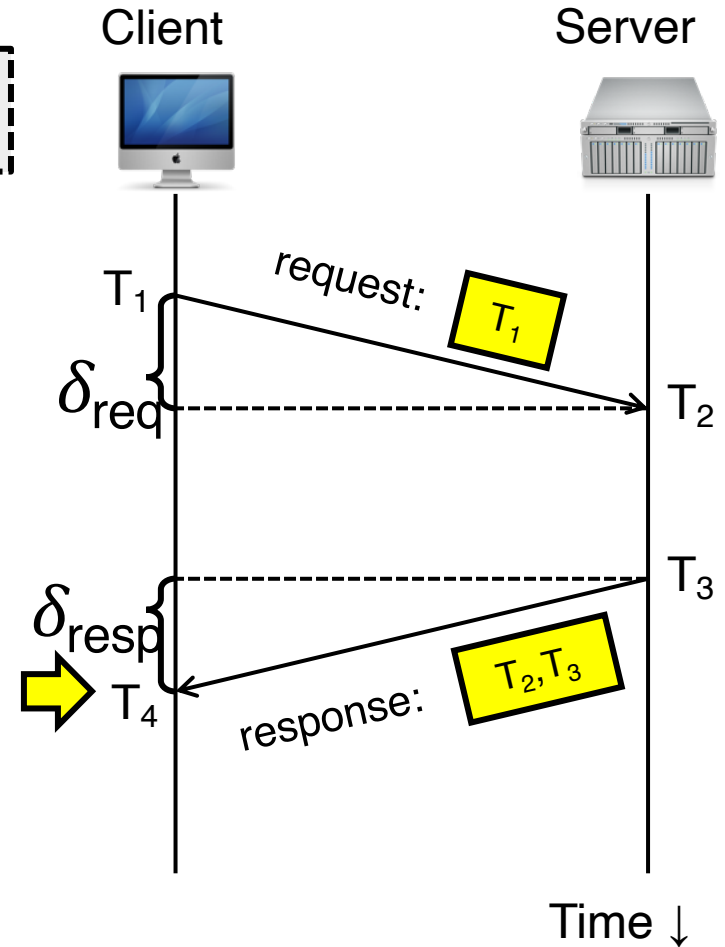




# Cristian's algorithm: Offset sample calculation

Goal: Client sets clock  $\leftarrow T_3 + \delta_{\text{resp}}$

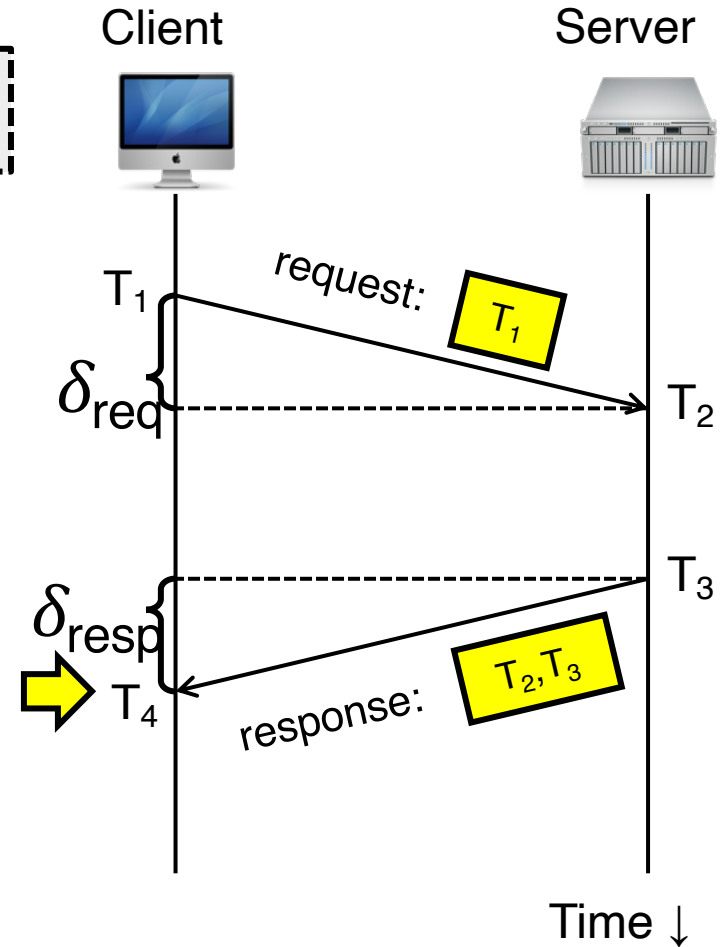
- Client samples round trip time  $\delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2)$



# Cristian's algorithm: Offset sample calculation

Goal: Client sets clock  $\leftarrow T_3 + \delta_{\text{resp}}$

- Client samples round trip time  $\delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2)$
- But client knows  $\delta$ , not  $\delta_{\text{resp}}$

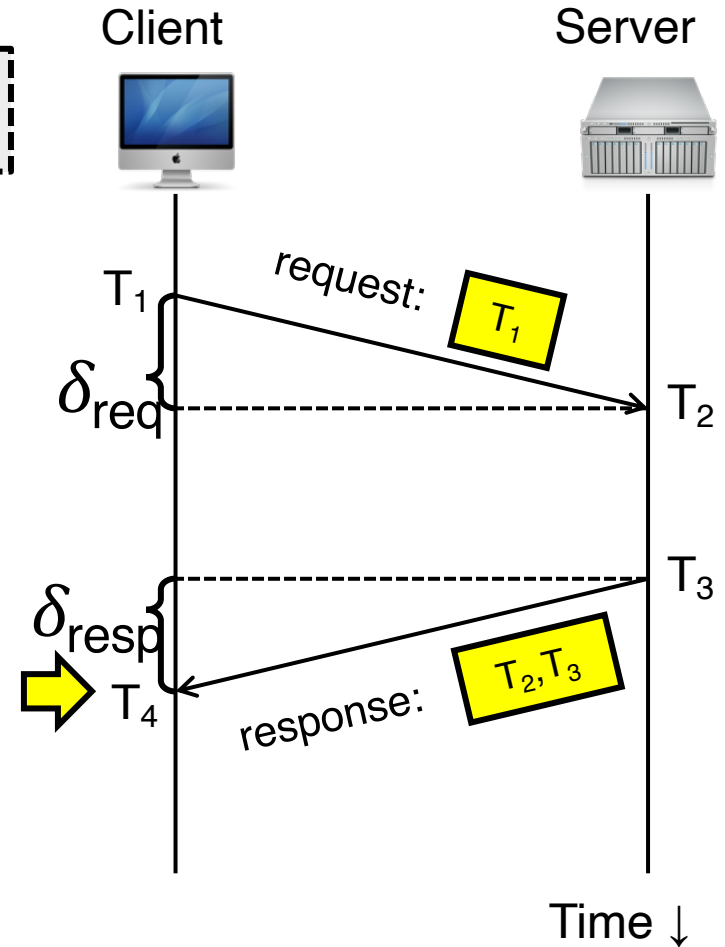


# Cristian's algorithm: Offset sample calculation

Goal: Client sets clock  $\leftarrow T_3 + \delta_{\text{resp}}$

- Client samples round trip time  $\delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2)$
- But client knows  $\delta$ , not  $\delta_{\text{resp}}$

Assume:  $\delta_{\text{req}} \approx \delta_{\text{resp}}$



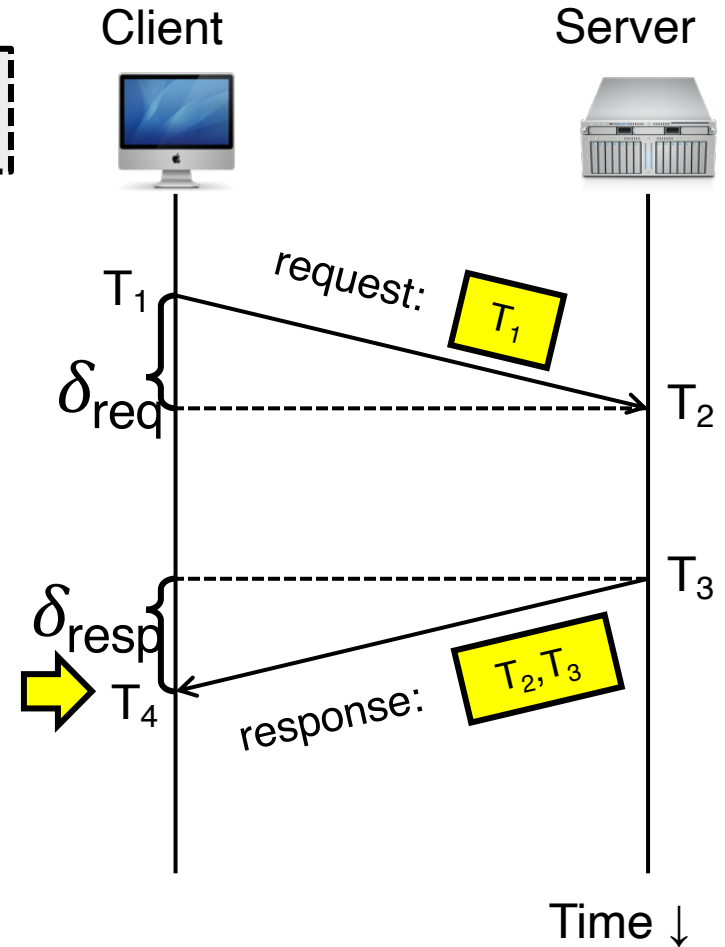
# Cristian's algorithm: Offset sample calculation

Goal: Client sets clock  $\leftarrow T_3 + \delta_{\text{resp}}$

- Client samples round trip time  $\delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2)$
- But client knows  $\delta$ , not  $\delta_{\text{resp}}$

Assume:  $\delta_{\text{req}} \approx \delta_{\text{resp}}$

Client sets clock  $\leftarrow T_3 + \frac{1}{2}\delta$



# Clock synchronization: Takeaway points

- Clocks on different systems will always behave differently
  - Disagreement between machines can result in undesirable behavior

# Clock synchronization: Takeaway points

- Clocks on different systems will always behave differently
  - Disagreement between machines can result in undesirable behavior
- NTP clock synchronization
  - Rely on timestamps to estimate network delays
  - 100s  $\mu$ s–ms accuracy
  - Clocks never exactly synchronized

# Clock synchronization: Takeaway points

- Clocks on different systems will always behave differently
  - Disagreement between machines can result in undesirable behavior
- NTP clock synchronization
  - Rely on timestamps to estimate network delays
  - 100s  $\mu$ s–ms accuracy
  - Clocks never exactly synchronized
- Often **inadequate** for distributed systems
  - Often need to reason about the order of events
  - Might need precision on the order of ns

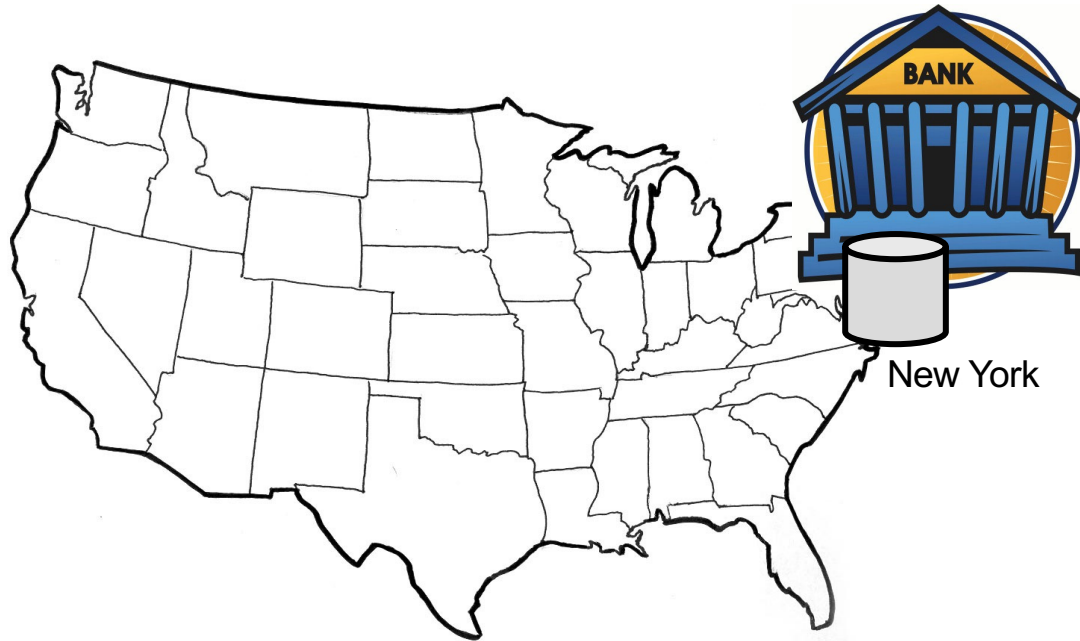
# Today's outline

- The need for time synchronization
- “Wall clock time” synchronization
  - Cristian's algorithm, NTP
- **Logical Time: Lamport Clocks**
- Vector clocks



# Motivation: Multi-site database replication

- A New York-based bank wants to make its transaction ledger database resilient to whole-site failures



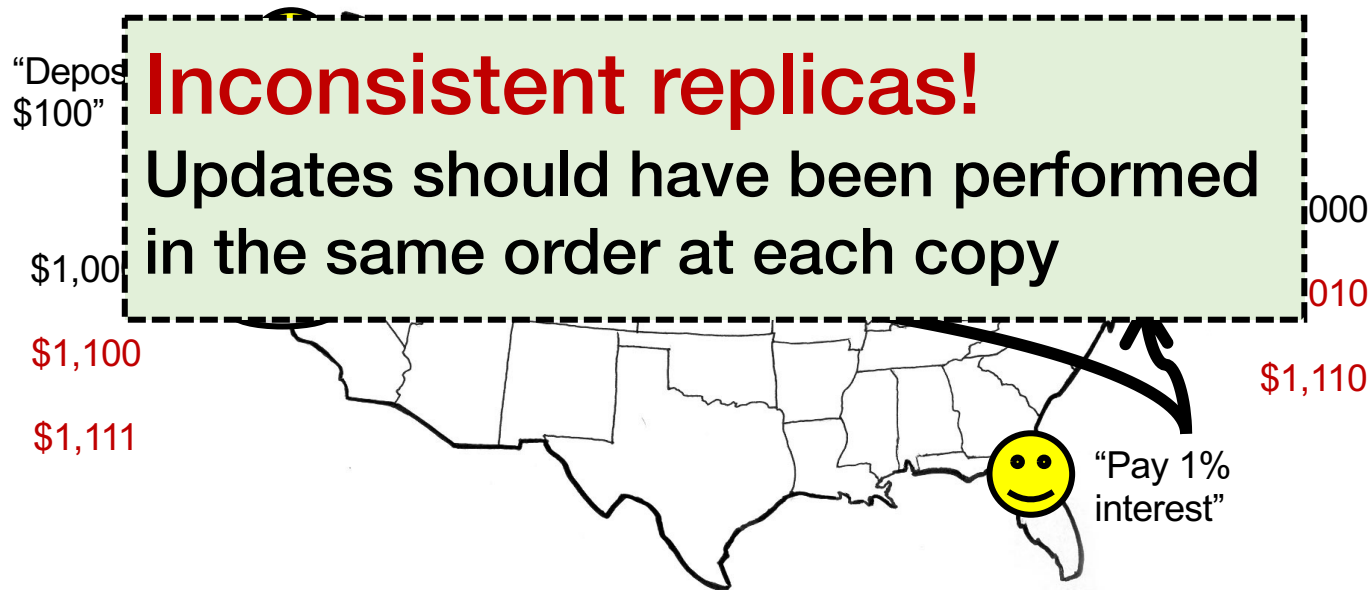
# Motivation: Multi-site database replication

- A New York-based bank wants to make its transaction ledger database resilient to whole-site failures
- **Replicate** the database, keep one copy in SF, one in NYC



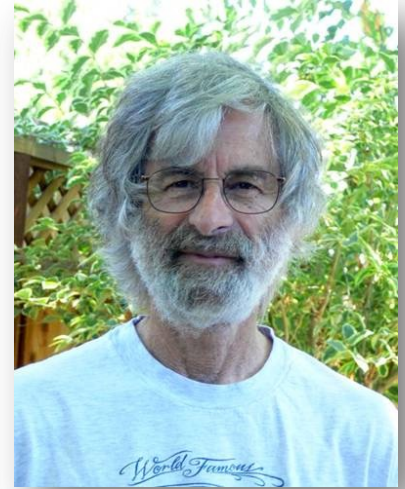
# The consequences of concurrent updates

- **Replicate** the database, keep one copy in **SF**, one in **NYC**
  - Client sends reads to the nearest copy
  - Client sends update to both copies



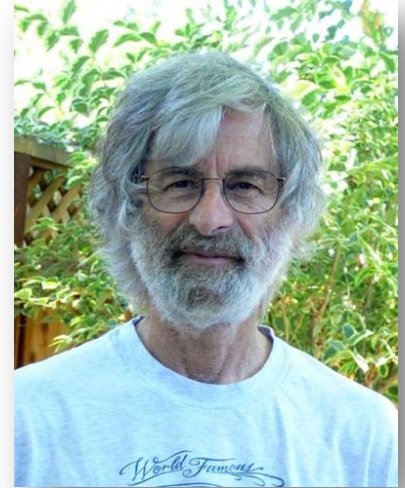
# Idea: Logical clocks

- Landmark 1978 paper by Leslie Lamport



# Idea: Logical clocks

- Landmark 1978 paper by Leslie Lamport
- Insights: only the **events themselves** matter

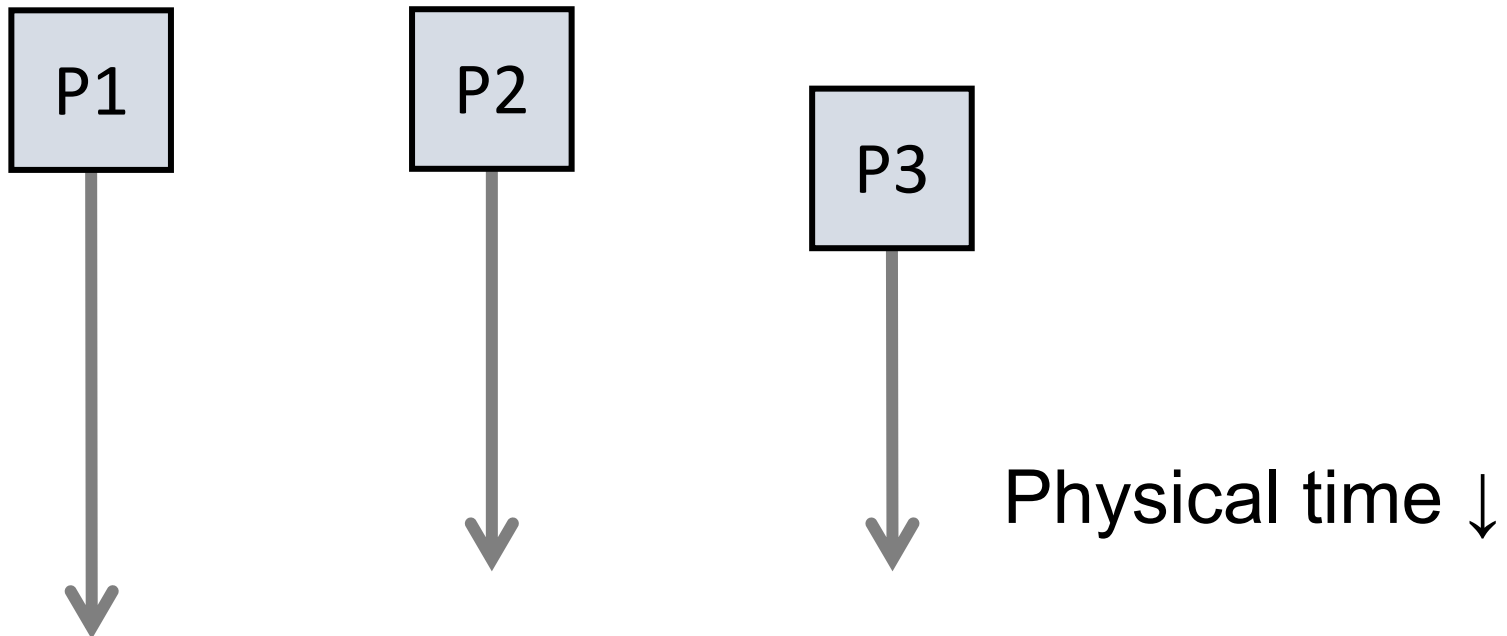


Idea: Disregard the precise clock time

Instead, capture **just** a “happens before” relationship between a pair of events

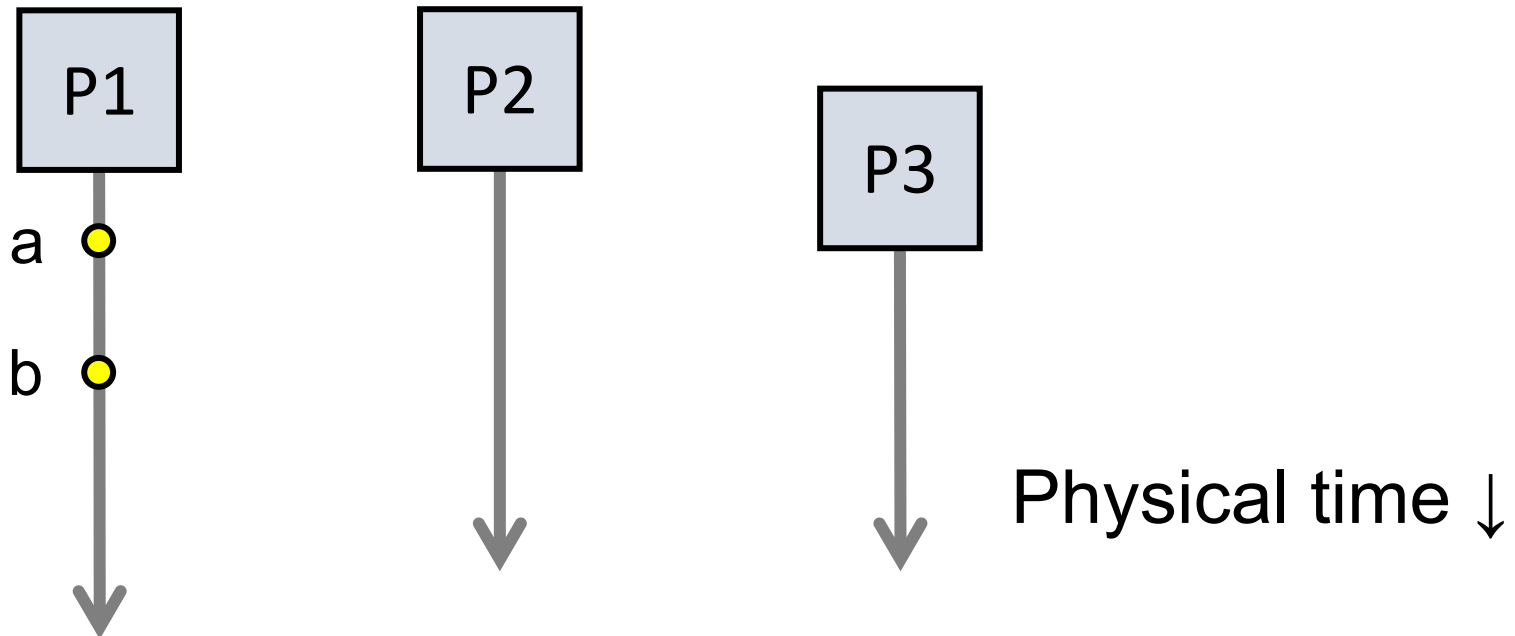
# Defining “happens-before” ( $\rightarrow$ )

- Consider three processes: P1, P2, and P3
- Notation: Event a happens before event b ( $a \rightarrow b$ )



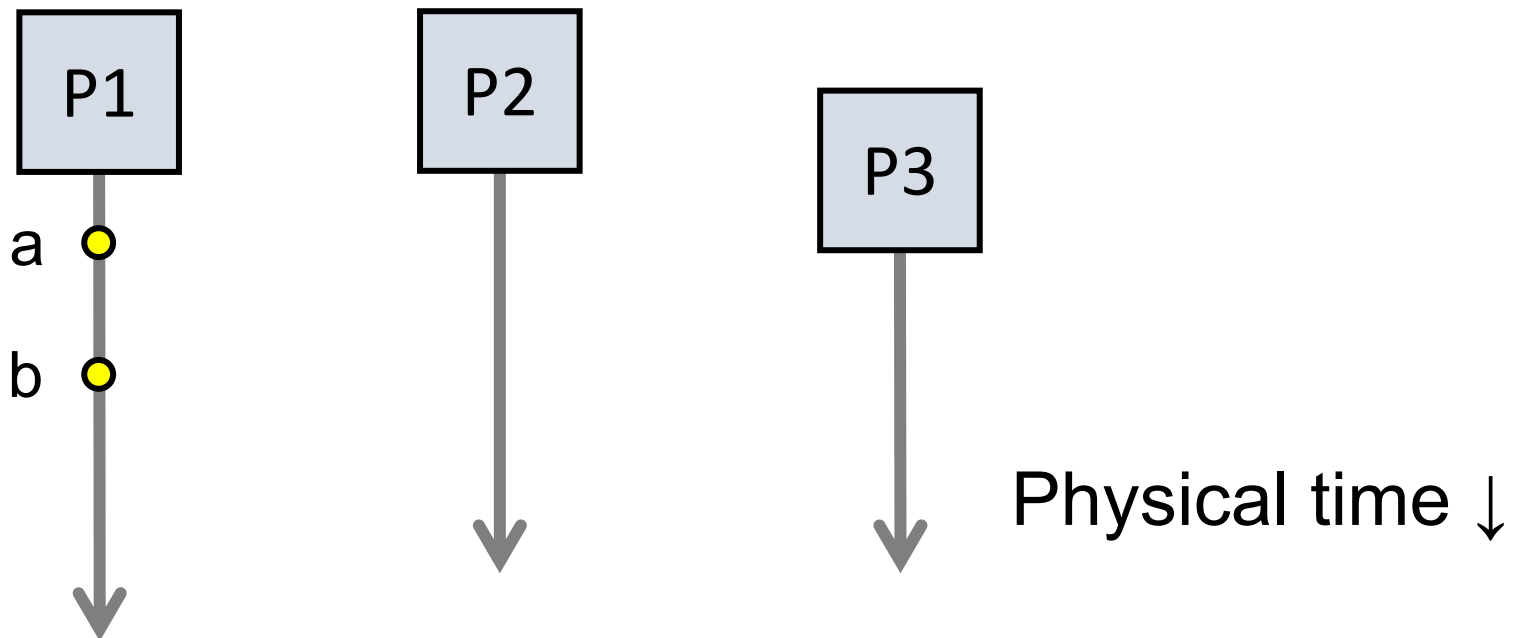
# Defining “happens-before” ( $\rightarrow$ )

- Can observe event order at a single process



# Defining “happens-before” ( $\rightarrow$ )

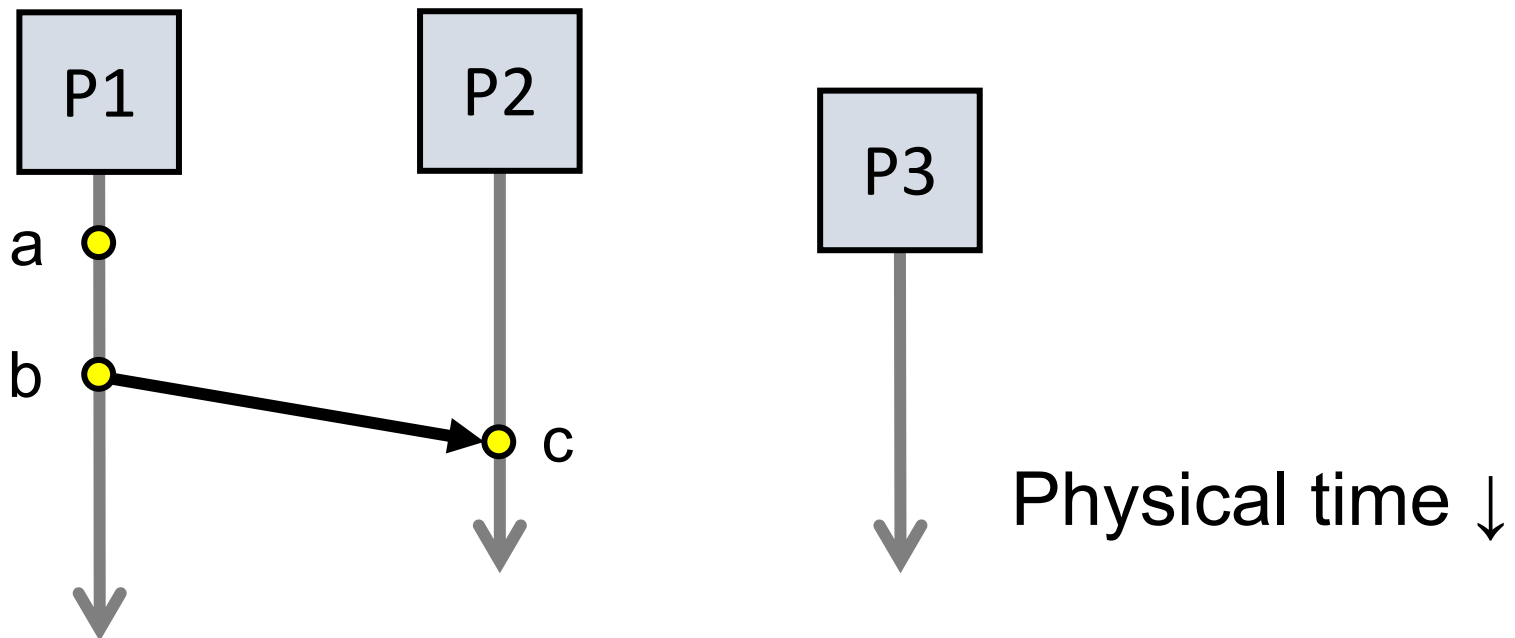
1. If same process and a occurs before b, then  $a \rightarrow b$





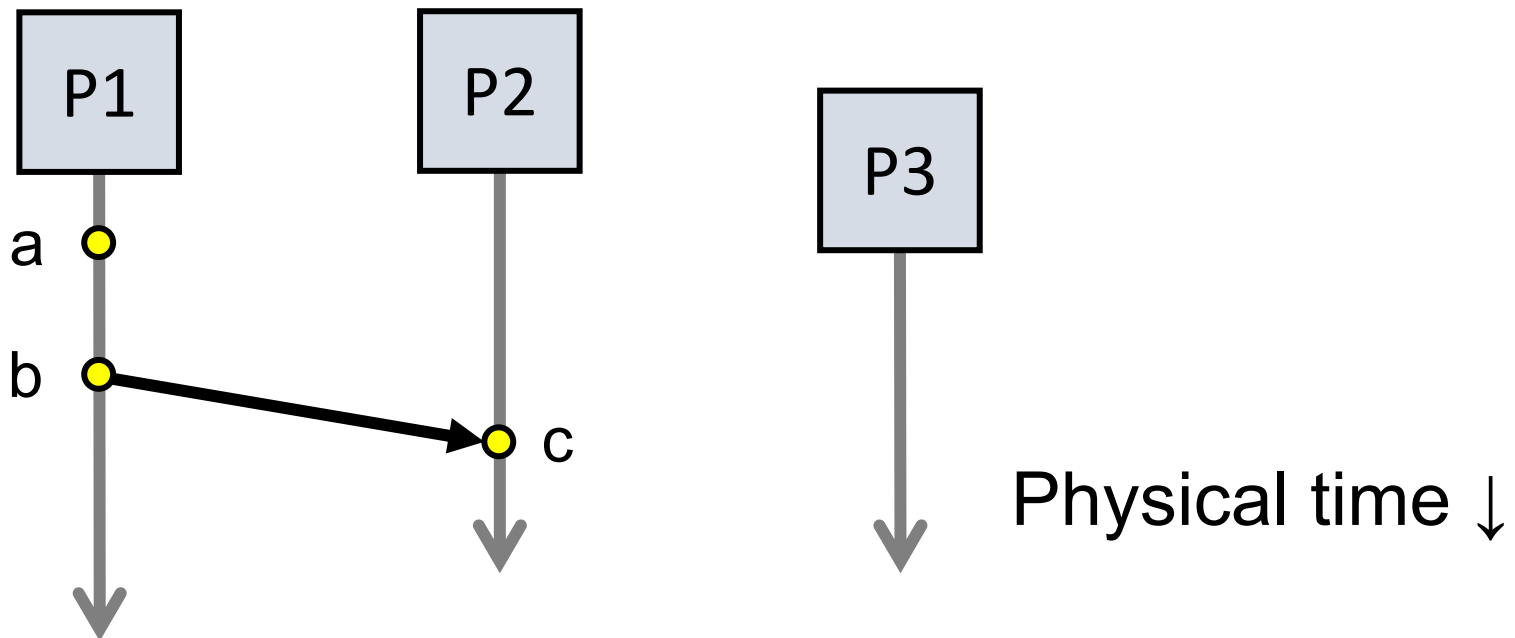
# Defining “happens-before” ( $\rightarrow$ )

1. If same process and  $a$  occurs before  $b$ , then  $a \rightarrow b$
2. Can observe ordering when processes communicate



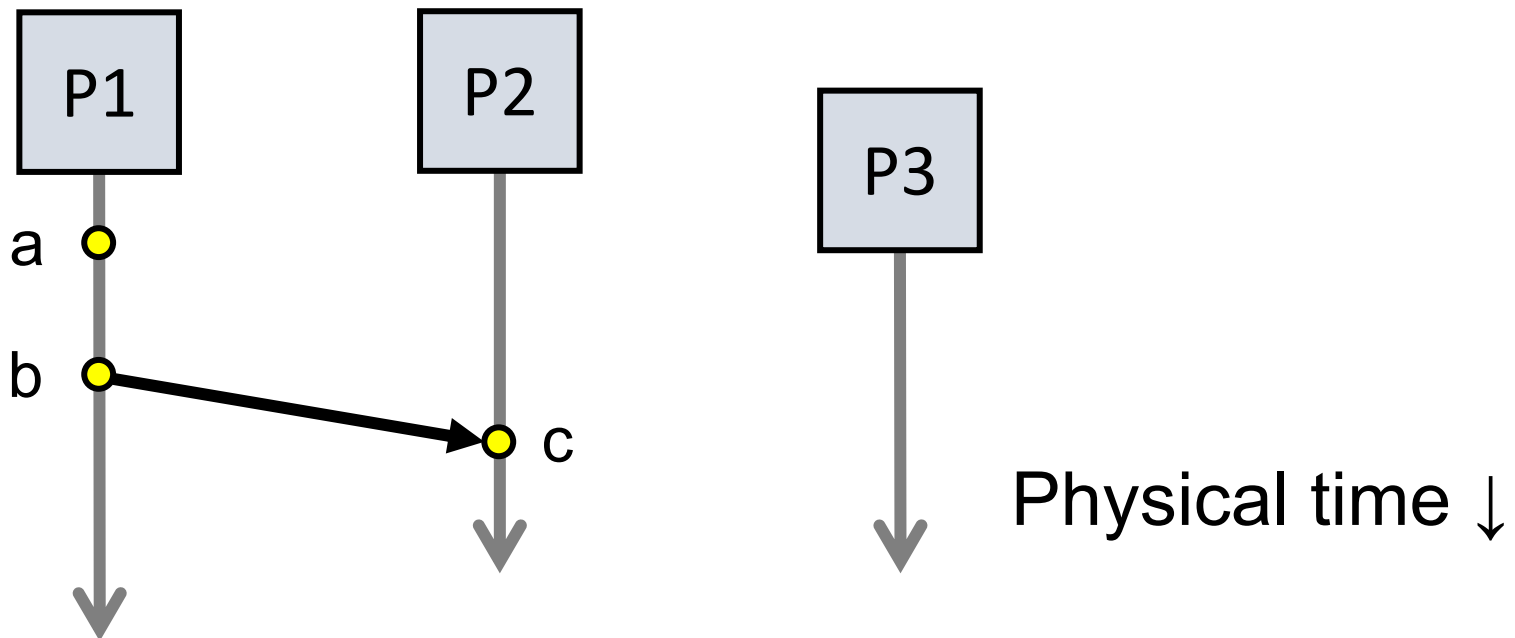
# Defining “happens-before” ( $\rightarrow$ )

1. If same process and  $a$  occurs before  $b$ , then  $a \rightarrow b$
2. If  $c$  is a message receipt of  $b$ , then  $b \rightarrow c$



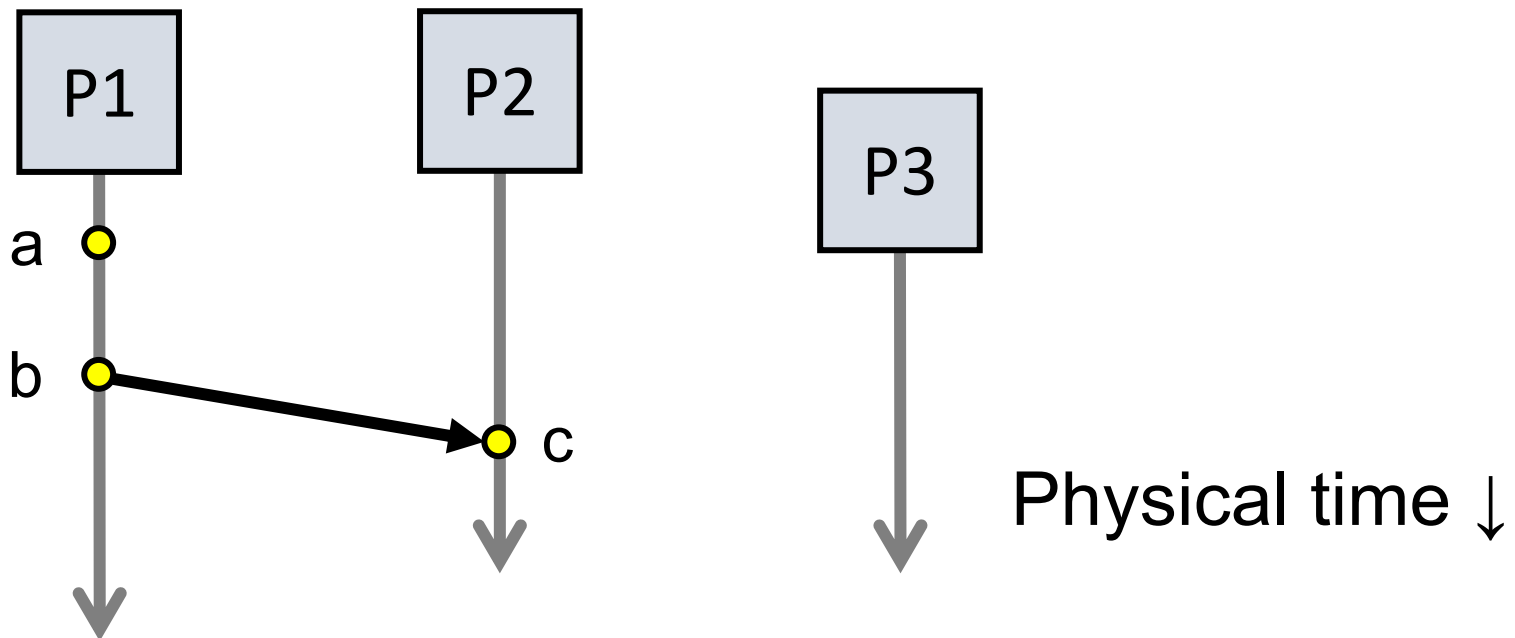
# Defining “happens-before” ( $\rightarrow$ )

1. If same process and  $a$  occurs before  $b$ , then  $a \rightarrow b$
2. If  $c$  is a message receipt of  $b$ , then  $b \rightarrow c$
3. Can observe ordering transitively



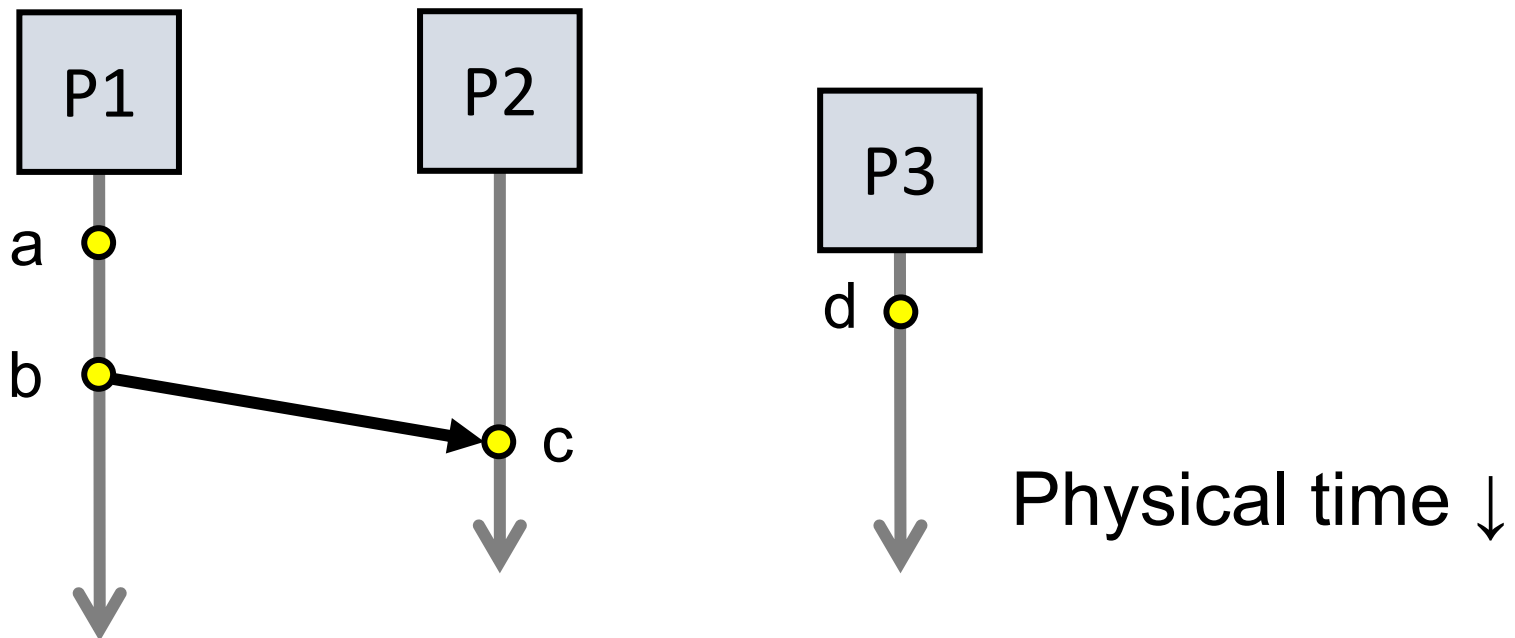
# Defining “happens-before” ( $\rightarrow$ )

1. If same process and  $a$  occurs before  $b$ , then  $a \rightarrow b$
2. If  $c$  is a message receipt of  $b$ , then  $b \rightarrow c$
3. If  $a \rightarrow b$  and  $b \rightarrow c$ , then  $a \rightarrow c$



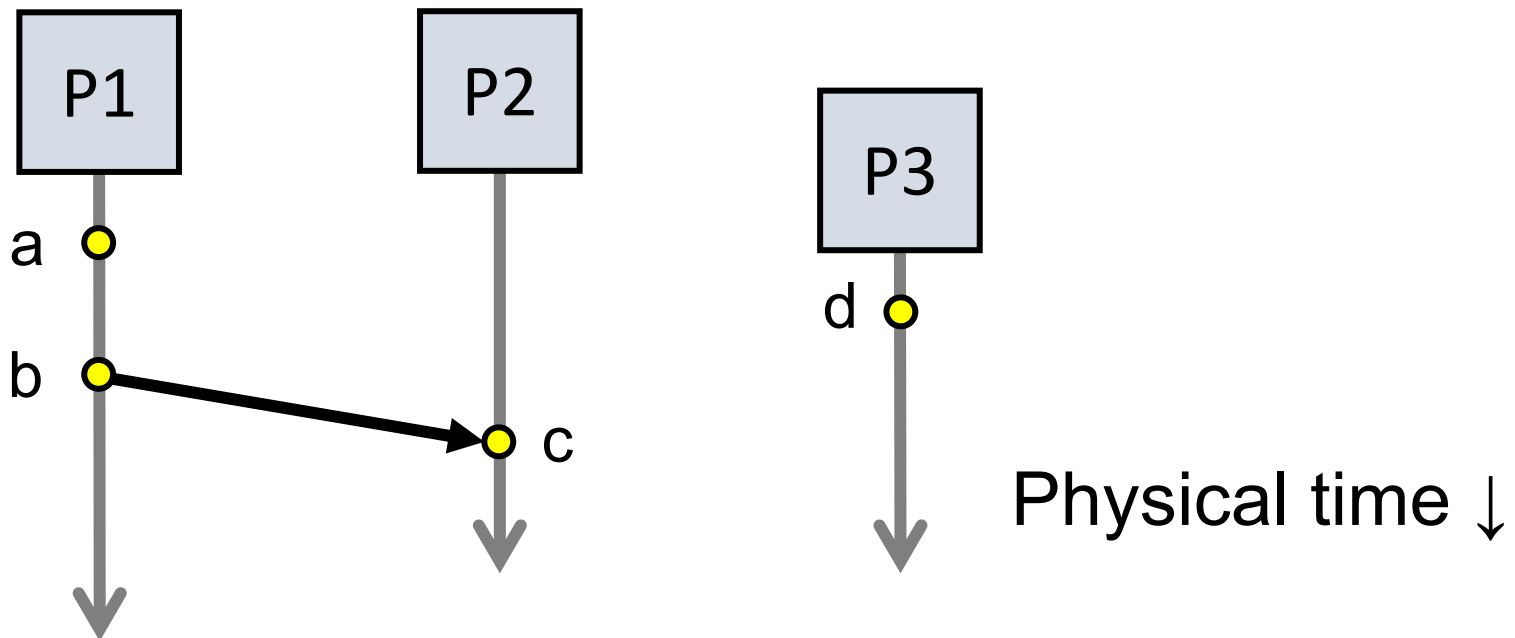
# Defining “happens-before” ( $\rightarrow$ )

1. Not all events are related by  $\rightarrow$



# Defining “happens-before” ( $\rightarrow$ )

1. Not all events are related by  $\rightarrow$
2.  $a, d$  not related by  $\rightarrow$  so concurrent, written as  $a \parallel d$



# Lamport clocks: Objective

- We seek a clock time  $C(a)$  for every event  $a$

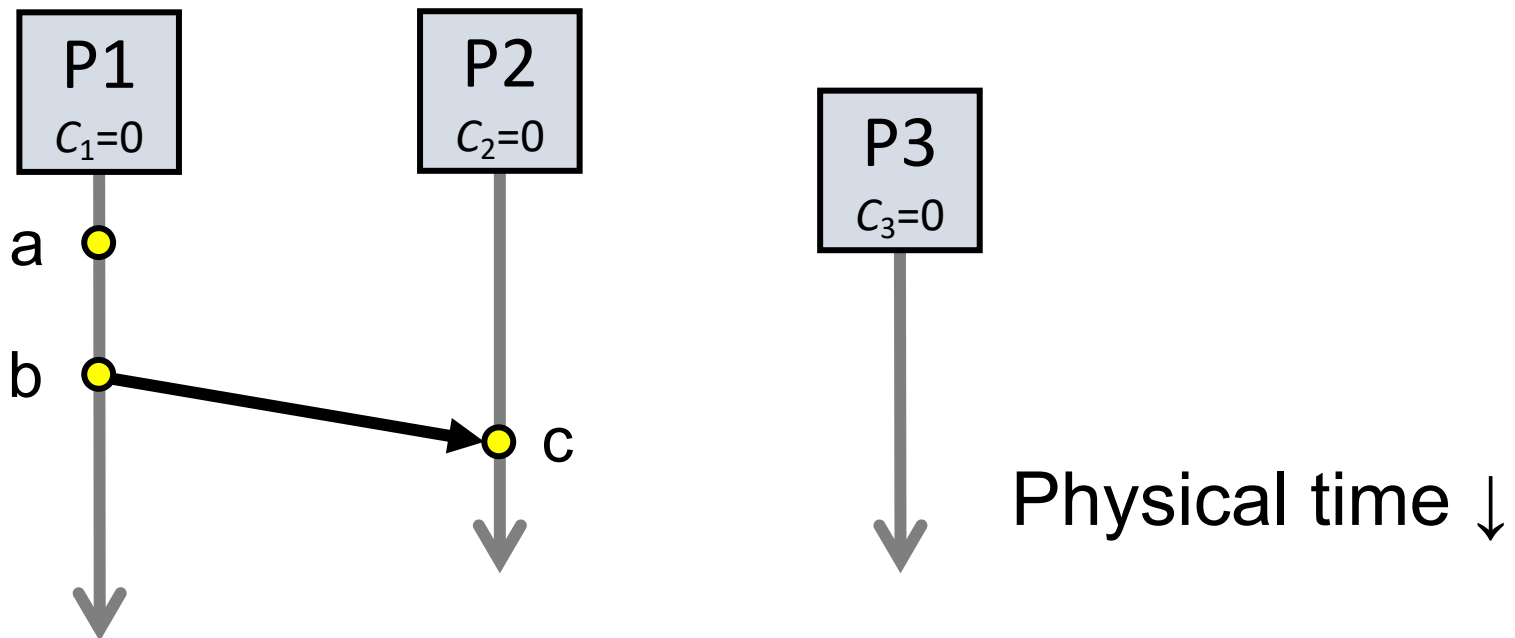
Plan: Tag events with clock times; use clock times to make distributed system correct

- Clock condition: If  $a \rightarrow b$ , then  $C(a) < C(b)$

# The Lamport Clock algorithm

- Each process  $P_i$  maintains a local clock  $C_i$

1. Before executing an event,  $C_i \leftarrow C_i + 1$ :

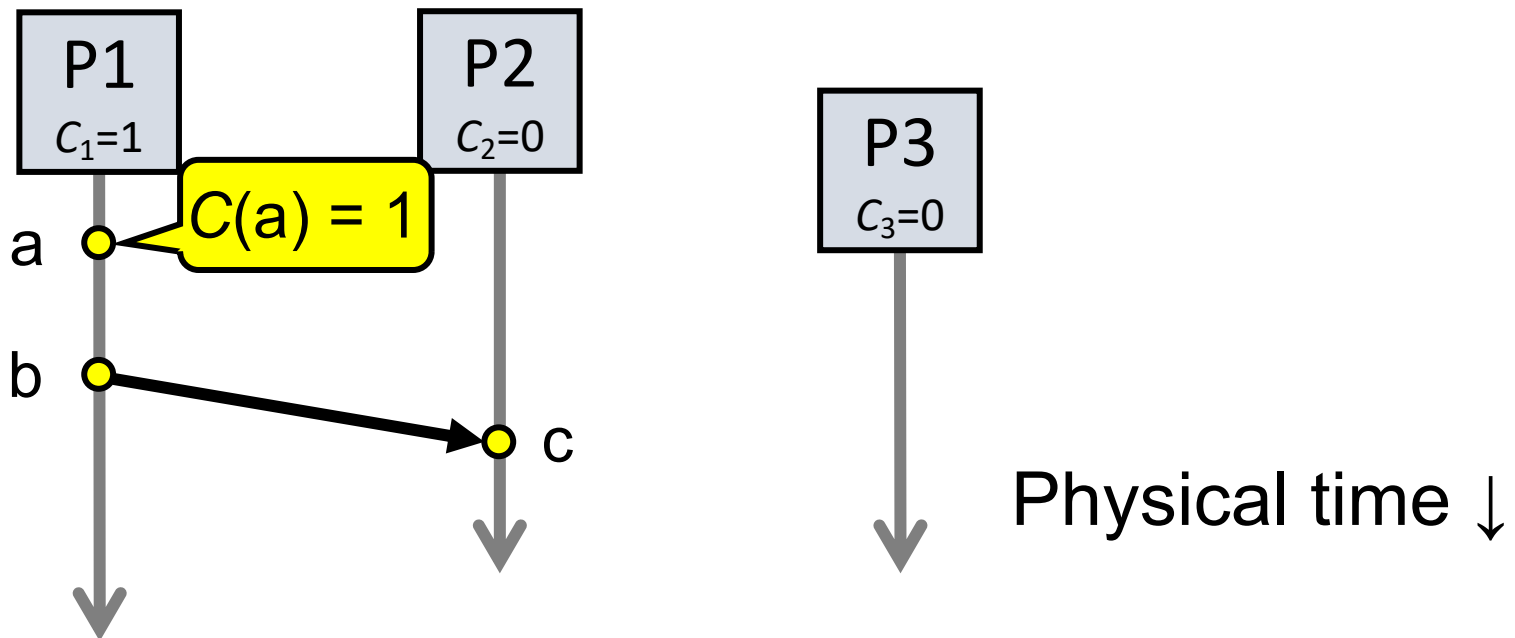




# The Lamport Clock algorithm

1. Before executing an event  $a$ ,  $C_i \leftarrow C_i + 1$ :

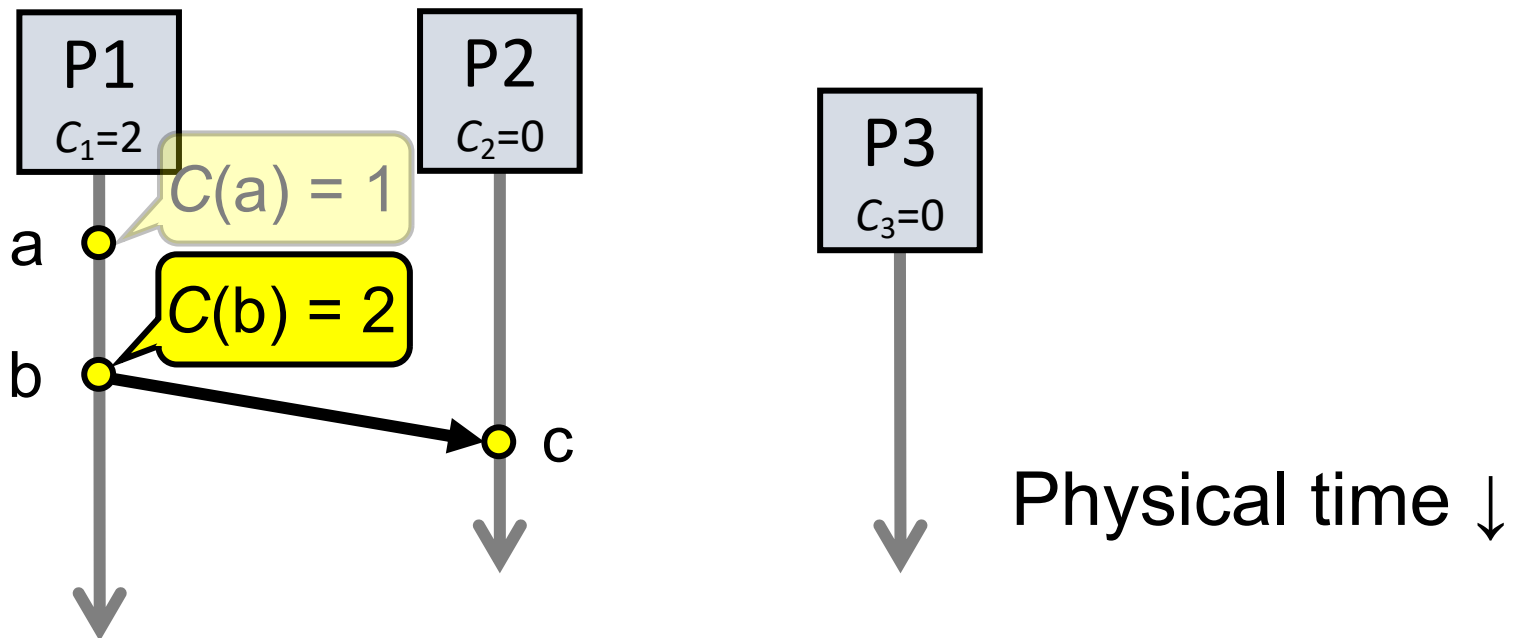
- Set event time  $C(a) \leftarrow C_i$



# The Lamport Clock algorithm

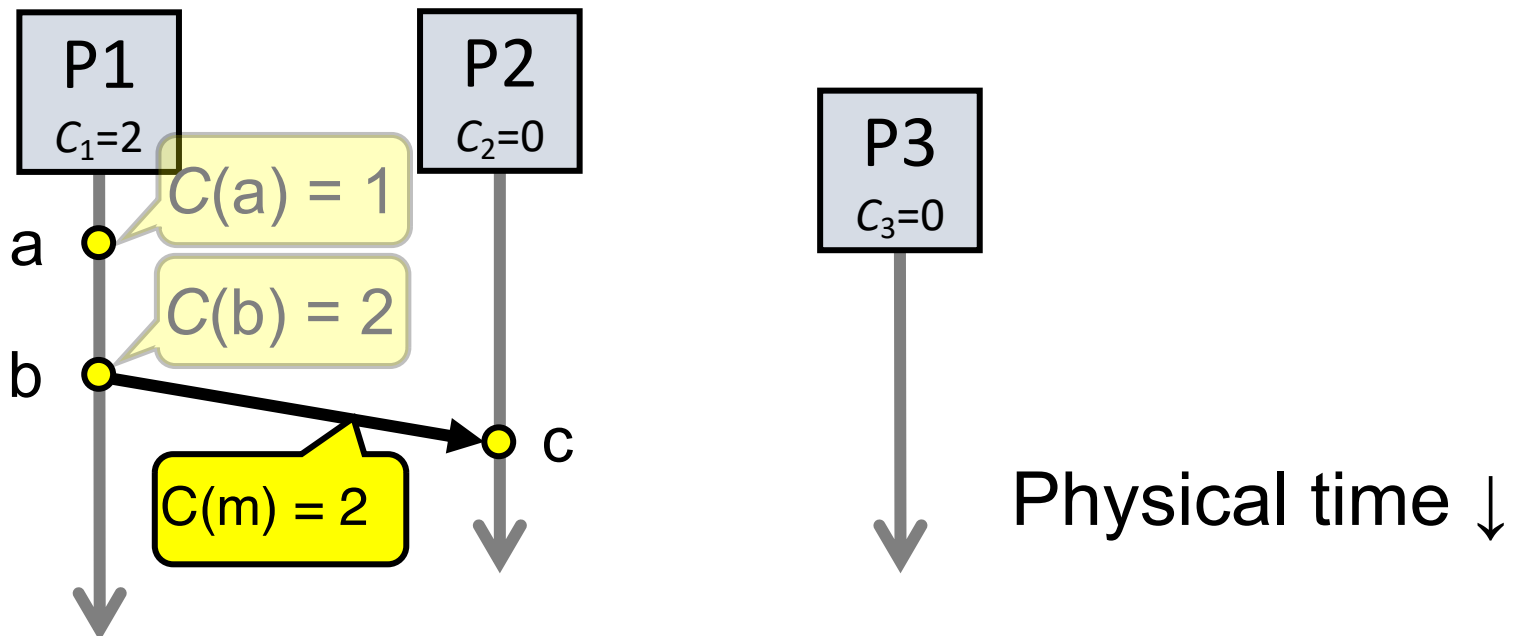
1. Before executing an event  $b$ ,  $C_i \leftarrow C_i + 1$ :

- Set event time  $C(b) \leftarrow C_i$



# The Lamport Clock algorithm

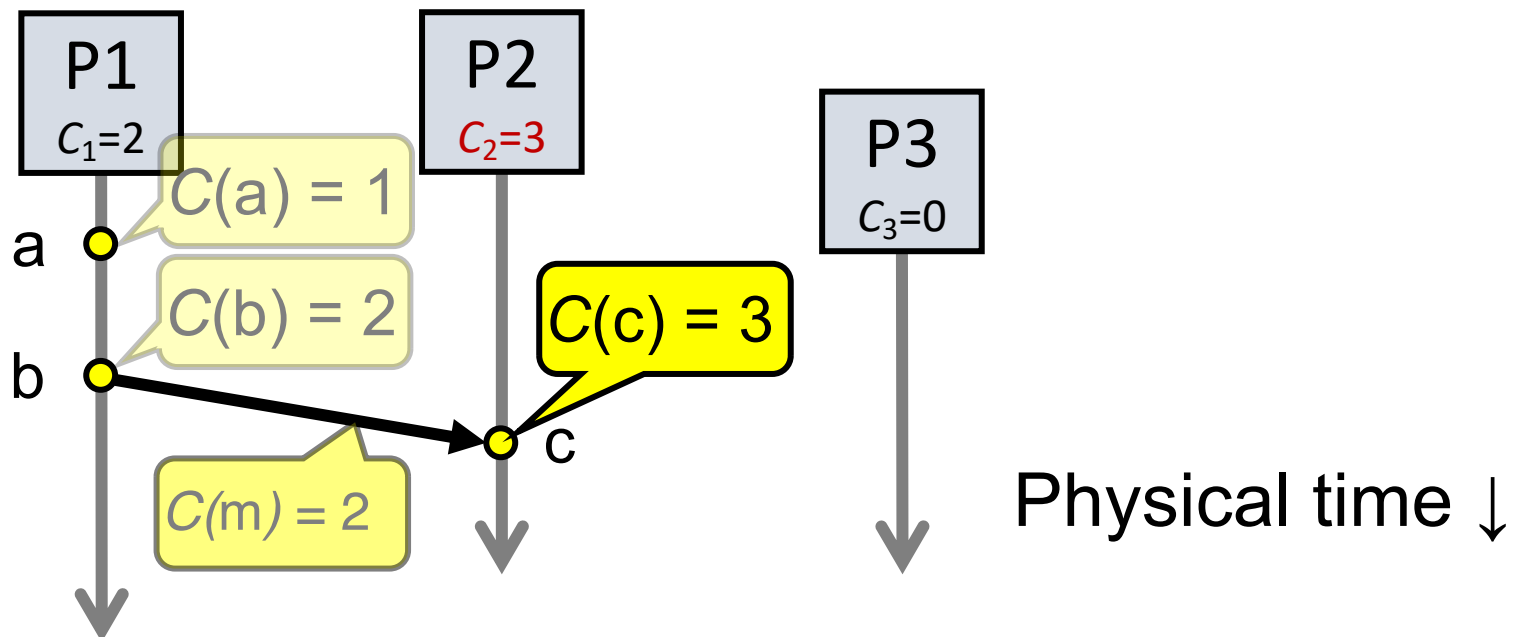
1. Before executing an event  $b$ ,  $C_i \leftarrow C_i + 1$
2. Send the local clock in the message  $m$



# The Lamport Clock algorithm

3. On process  $P_j$  receiving a message  $m$ :

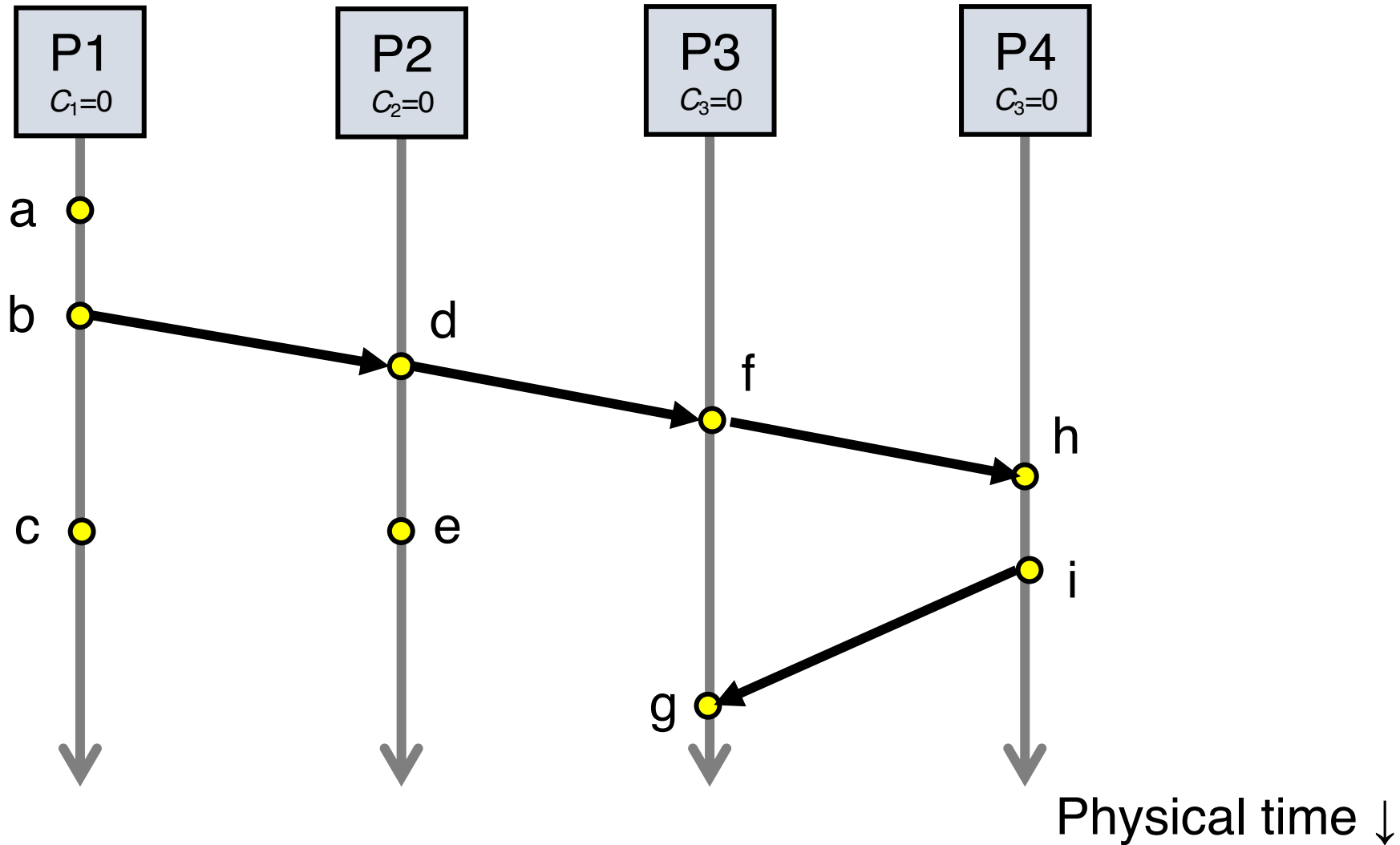
- Set  $C_j$  **and** receive event time  $C(c) \leftarrow 1 + \max\{ C_j, C(m) \}$



# Lamport Timestamps: Ordering all events

- Break ties by appending the process number to each event:
  1. Process  $P_i$  timestamps event  $e$  with  $C_i(e).i$
  2.  $C(a).i < C(b).j$  when:
    - $C(a) < C(b)$ , or  $C(a) = C(b)$  and  $i < j$
- Now, for any two events  $a$  and  $b$ ,  $C(a) < C(b)$  or  $C(b) < C(a)$ 
  - This is called a total ordering of events

# Order all these events



# Totally-Ordered Multicast

Goal: All sites apply updates in (same) Lamport clock order

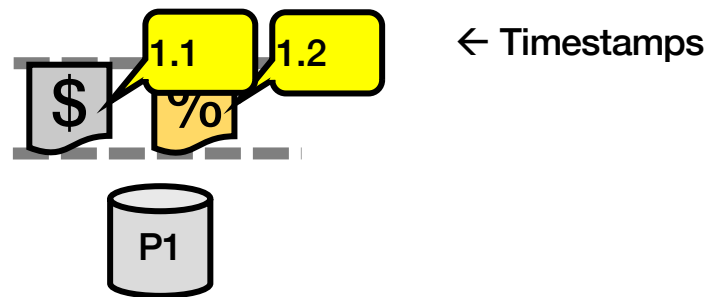
- Client sends update to one replica site  $j$ 
  - Replica assigns it Lamport timestamp  $C_j.j$

# Totally-Ordered Multicast

Goal: All sites apply updates in (same) Lamport clock order

- Client sends update to one replica site  $j$ 
  - Replica assigns it Lamport timestamp  $C_j . j$
- Key idea: Place events into a sorted **local queue**
  - **Sorted** by increasing Lamport timestamps

Example: P1's  
local queue:





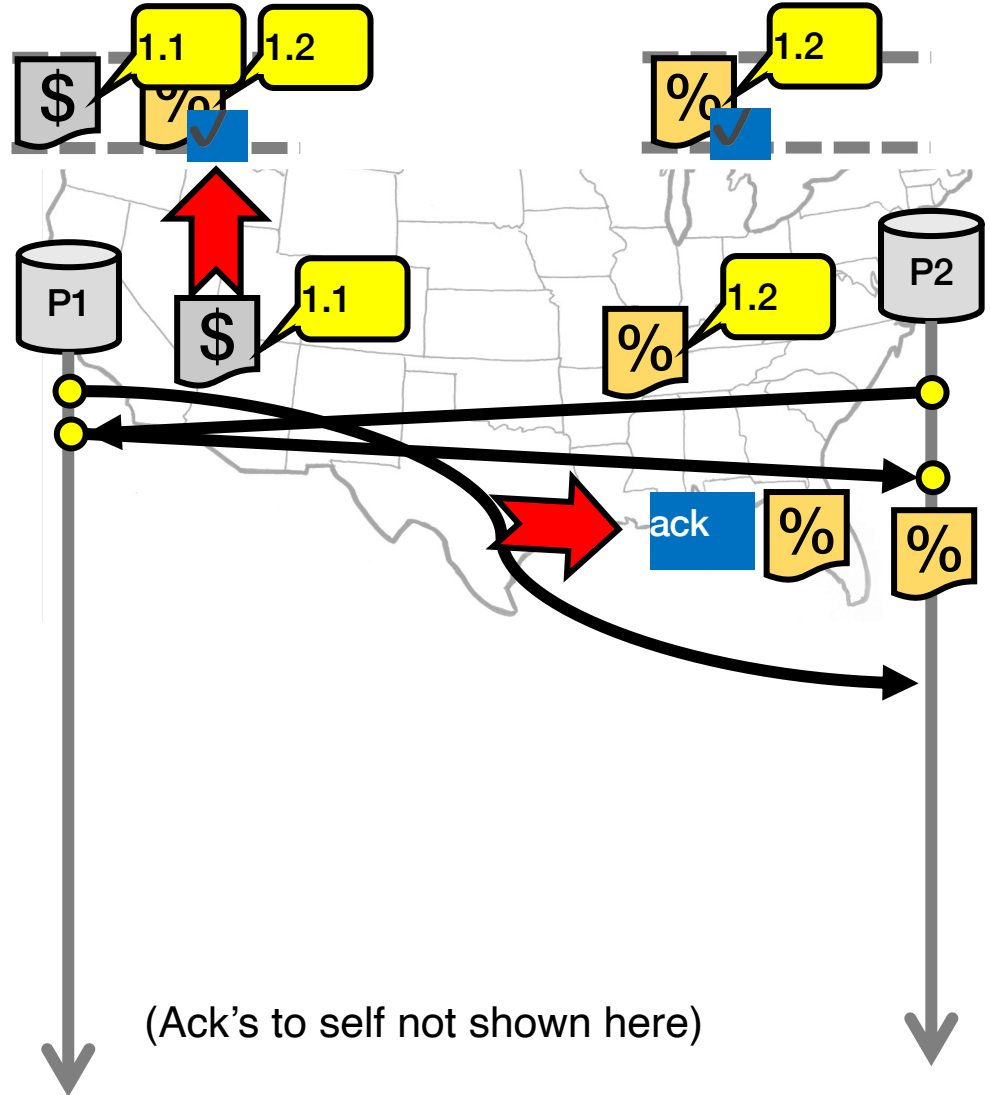
# Totally-Ordered Multicast (Almost correct)

1. On receiving an update from client, broadcast to others (including yourself)
2. On receiving an update from replica:
  - a) Add it to your local queue
  - b) Broadcast an **acknowledgement message** to every replica (including yourself)
3. On receiving an acknowledgement:
  - Mark corresponding update **acknowledged** in your queue
4. **Remove and process** updates everyone has ack'ed from head of queue

# Totally-Ordered Multicast (Almost correct)

- P1 queues \$, P2 queues %
- P1 queues and ack's %
  - P1 marks % fully ack'ed
- P2 marks % fully ack'ed

X P2 processes %



# Totally-Ordered Multicast (Correct version)

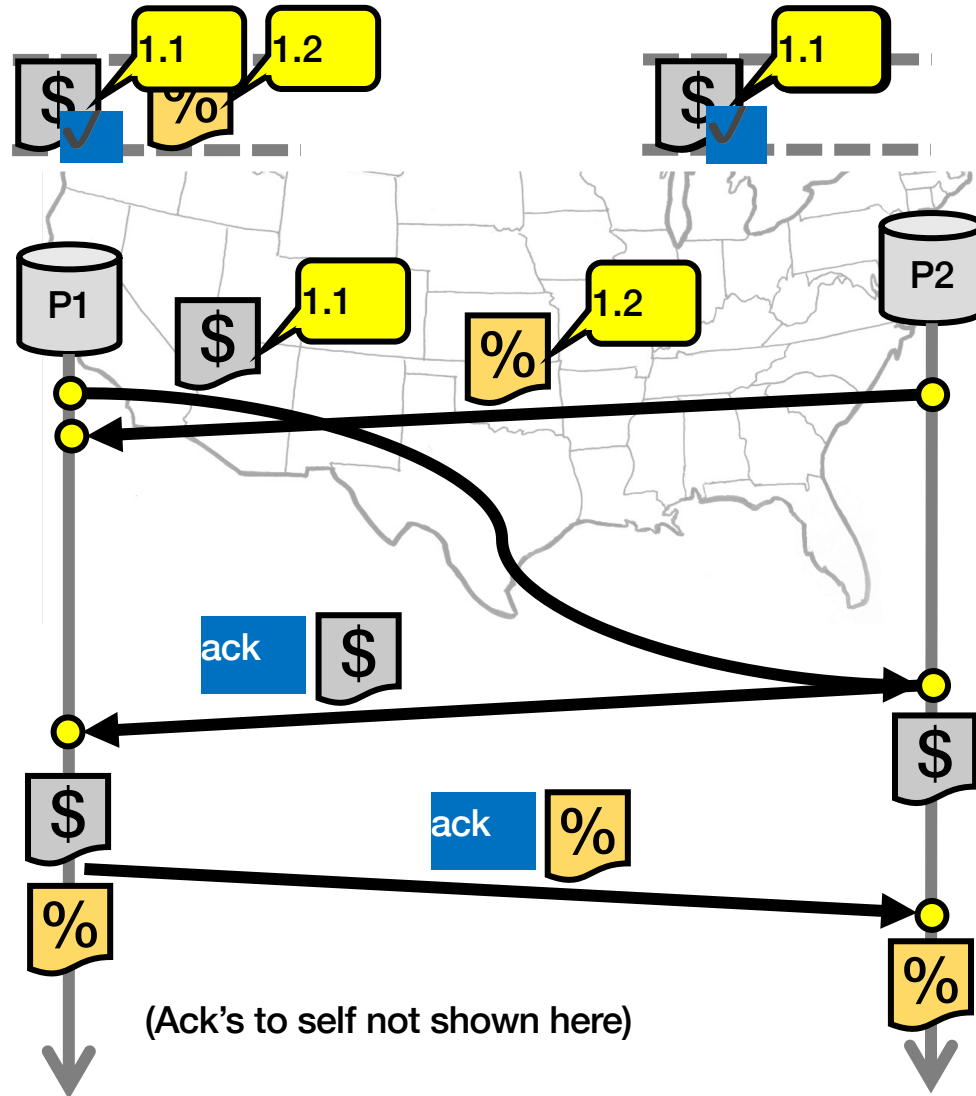
1. On receiving an update from client, broadcast to others (including yourself)

2. On receiving **or processing** an update:
  - a) Add it to your local queue
  - b) Broadcast an **acknowledgement message** to every replica (including yourself) **only from head of queue**

3. On receiving an acknowledgement:
  - Mark corresponding update **acknowledged** in your queue

4. **Remove and process** updates everyone has ack'ed from head of queue

# Totally-Ordered Multicast (Correct version)



# So, are we done?

- *Does totally-ordered multicast solve the problem of multi-site replication in general?*

# So, are we done?

- *Does totally-ordered multicast solve the problem of multi-site replication in general?*
- Not by a long shot!
- 1. Our protocol **assumed:**
  - No node failures
  - No message loss
  - No message corruption

# So, are we done?

- *Does totally-ordered multicast solve the problem of multi-site replication in general?*
- Not by a long shot!
- 1. Our protocol **assumed**:
  - No node failures
  - No message loss
  - No message corruption
- 2. All-to-all communication **does not scale**

# So, are we done?

- *Does totally-ordered multicast solve the problem of multi-site replication in general?*
- Not by a long shot!
- 1. Our protocol **assumed:**
  - No node failures
  - No message loss
  - No message corruption
- 2. All-to-all communication **does not scale**
- 3. **Waits forever** for message delays (performance?)



# Lamport Clocks: Takeaway points

- Can totally-order events in a distributed system: that's useful!
  - We saw an application of Lamport clocks for totally-ordered multicast

# Lamport Clocks: Takeaway points

- Can totally-order events in a distributed system: that's useful!
  - We saw an application of Lamport clocks for totally-ordered multicast
- But: while by construction,
$$a \rightarrow b \text{ implies } C(a) < C(b),$$
  - The converse is not necessarily true:
    - $C(a) < C(b)$  does not imply  $a \rightarrow b$  (possibly,  $a \parallel b$ )

# Lamport Clocks: Takeaway points

- Can totally-order events in a distributed system: that's useful!
  - We saw an application of Lamport clocks for totally-ordered multicast
- But: while by construction,
$$a \rightarrow b \text{ implies } C(a) < C(b),$$
  - The converse is not necessarily true:
    - $C(a) < C(b)$  does not imply  $a \rightarrow b$  (possibly,  $a \parallel b$ )

**Can't** use Lamport timestamps to infer **causal relationships** between events

# Today's outline

- The need for time synchronization
- “Wall clock time” synchronization
  - Cristian's algorithm, NTP
- Logical Time: Lamport Clocks
- **Vector clocks**

# Lamport Clocks and causality

- Lamport clock timestamps do not capture causality
- Given two timestamps  $C(a)$  and  $C(z)$ , want to know whether there's a chain of events linking them:

$$a \rightarrow b \rightarrow \dots \rightarrow y \rightarrow z$$

# Vector clock: Introduction

- One integer can't order events in more than one process
- So, a **Vector Clock (VC)** is a vector of integers, one entry for each process in the entire distributed system
  - Label event  $e$  with  $VC(e) = [c_1, c_2, \dots, c_n]$ 
    - Each entry  $c_k$  is a count of events in process  $k$  that causally precede  $e$

# Vector clock: Update rules

- Initially, all vectors are  $[0, 0, \dots, 0]$
- Two update rules:
  1. For each local event on process  $i$ , increment local entry  $c_i$

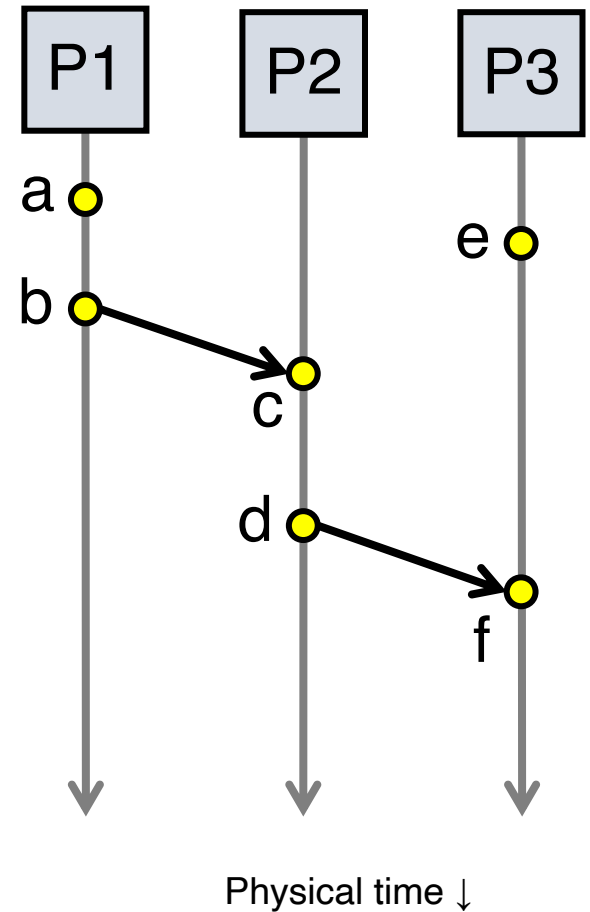
# Vector clock: Update rules

- Initially, all vectors are  $[0, 0, \dots, 0]$
- Two update rules:
  1. For each local event on process  $i$ , increment local entry  $c_i$
  2. If process  $j$  receives message with vector  $[d_1, d_2, \dots, d_n]$ :
    - Set each local entry  $c_k = \max\{c_k, d_k\}$
    - Increment local entry  $c_j$



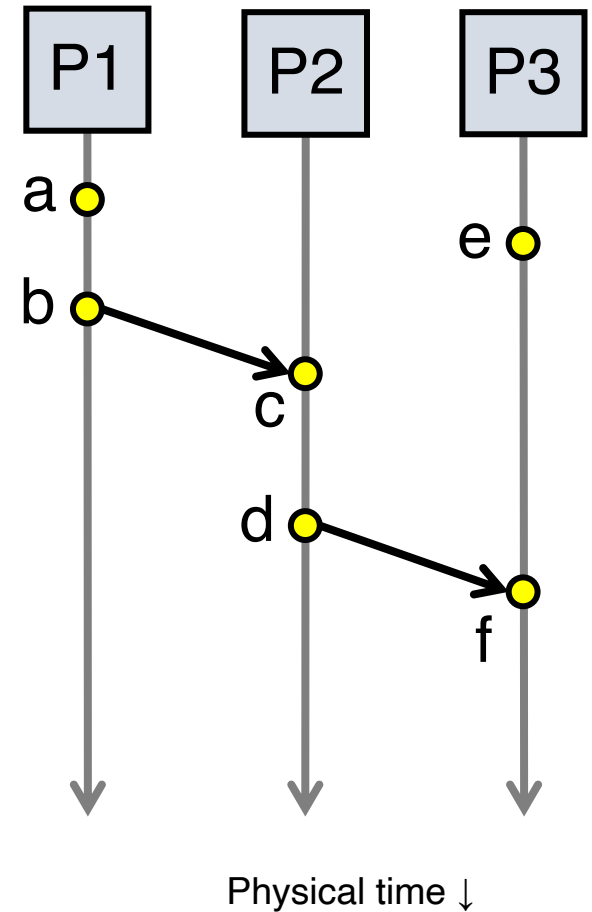
# Vector clock: Example

- All processes' VCs start at  $[0, 0, 0]$



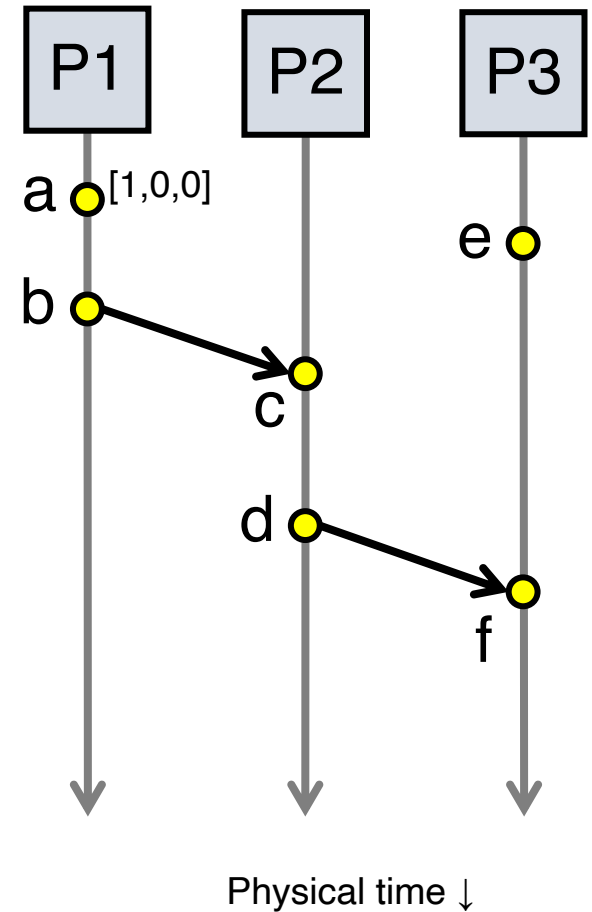
# Vector clock: Example

- All processes' VCs start at  $[0, 0, 0]$
- Applying local update rule



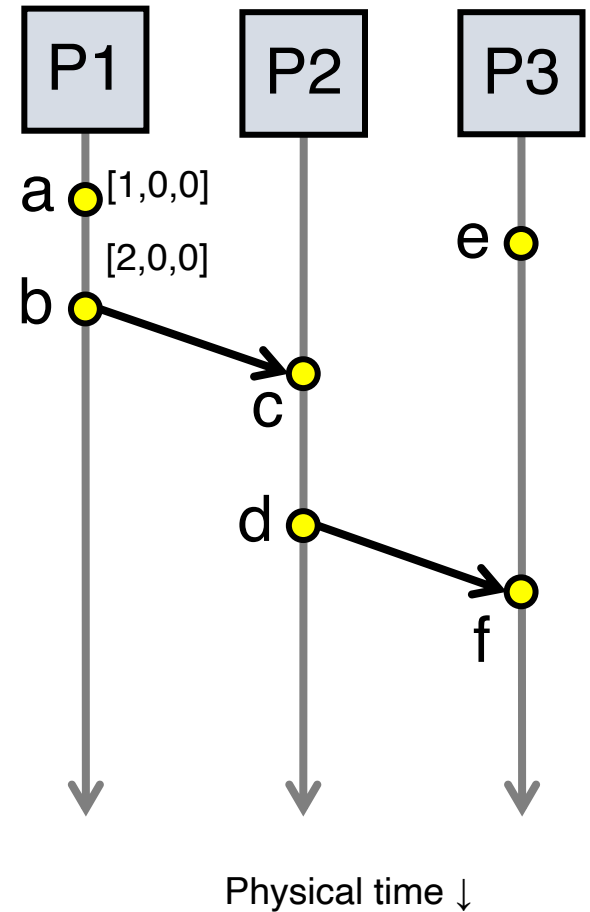
# Vector clock: Example

- All processes' VCs start at  $[0, 0, 0]$
- Applying local update rule



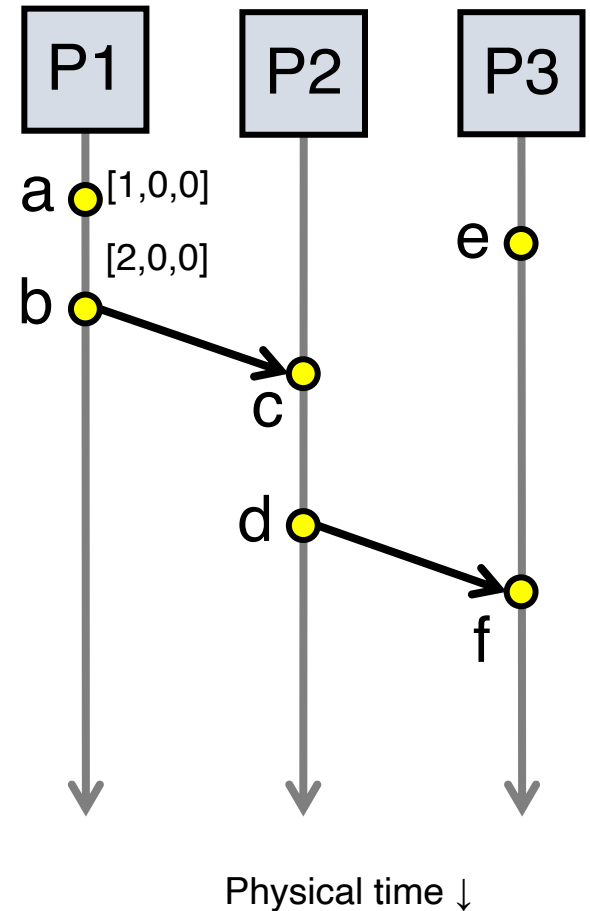
# Vector clock: Example

- All processes' VCs start at  $[0, 0, 0]$
- Applying local update rule



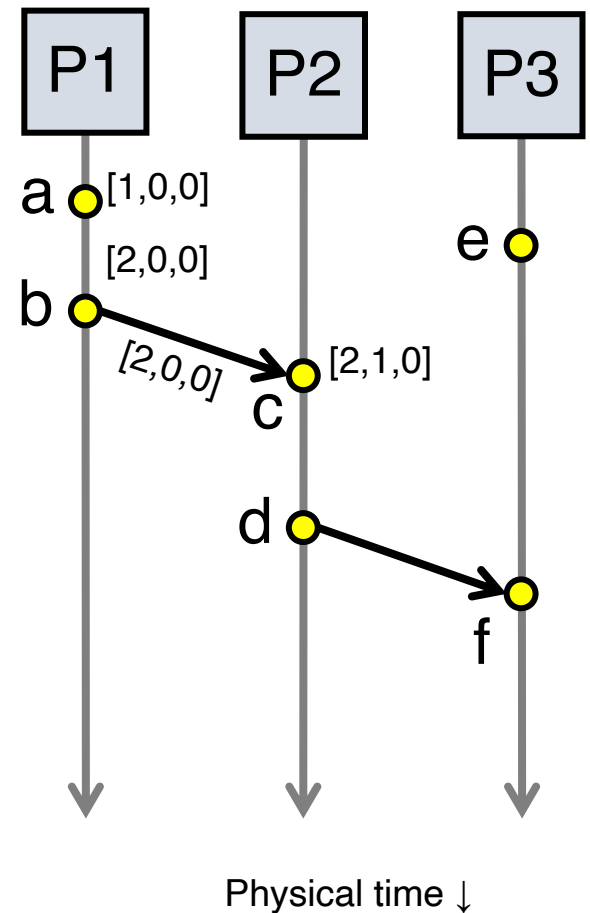
# Vector clock: Example

- All processes' VCs start at  $[0, 0, 0]$
- Applying local update rule
- Applying message rule
  - Local vector clock **piggybacks** on inter-process messages



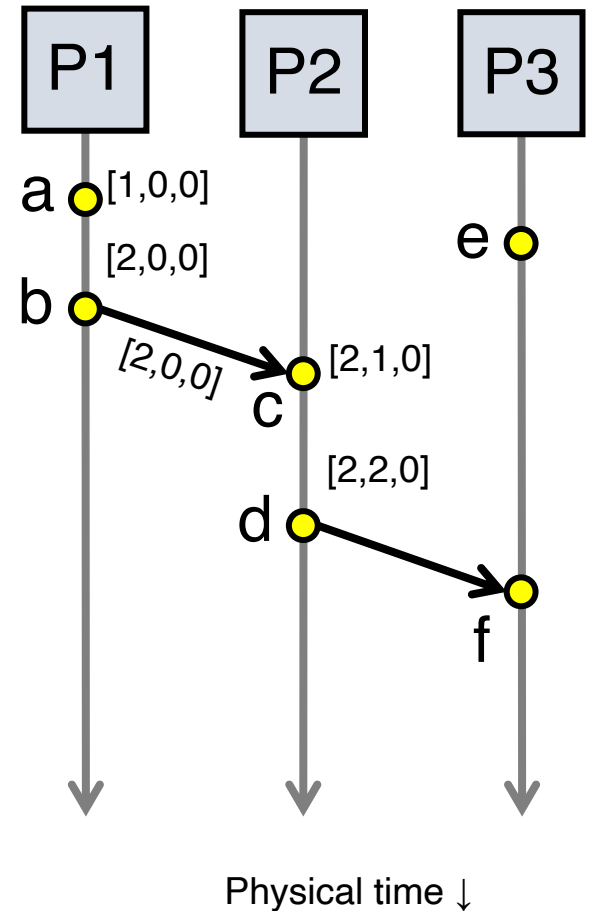
# Vector clock: Example

- All processes' VCs start at  $[0, 0, 0]$
- Applying local update rule
- Applying message rule
  - Local vector clock **piggybacks** on inter-process messages



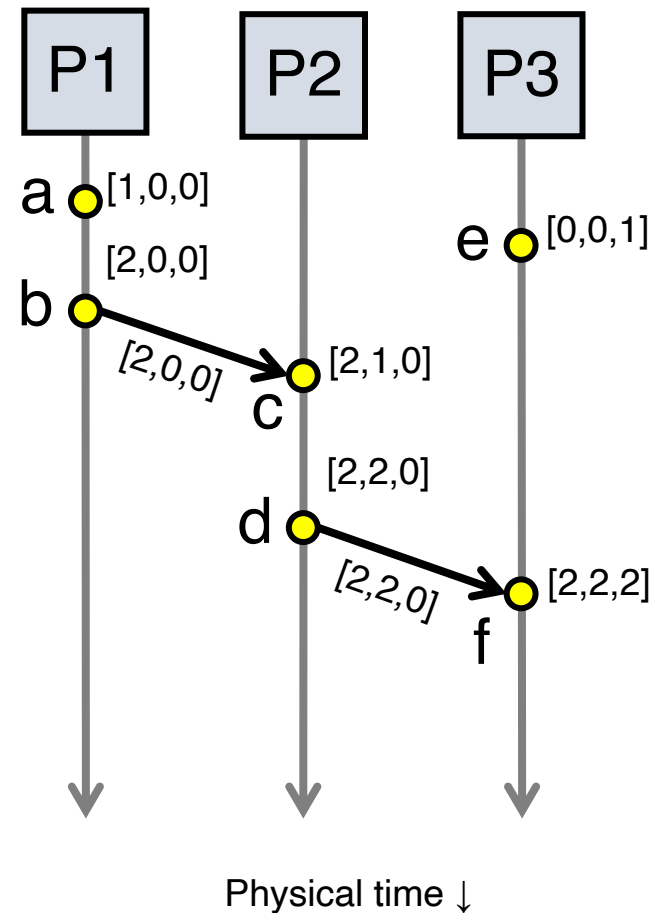
# Vector clock: Example

- All processes' VCs start at  $[0, 0, 0]$
- Applying local update rule
- Applying message rule
  - Local vector clock **piggybacks** on inter-process messages



# Vector clock: Example

- All processes' VCs start at  $[0, 0, 0]$
- Applying local update rule
- Applying message rule
  - Local vector clock **piggybacks** on inter-process messages



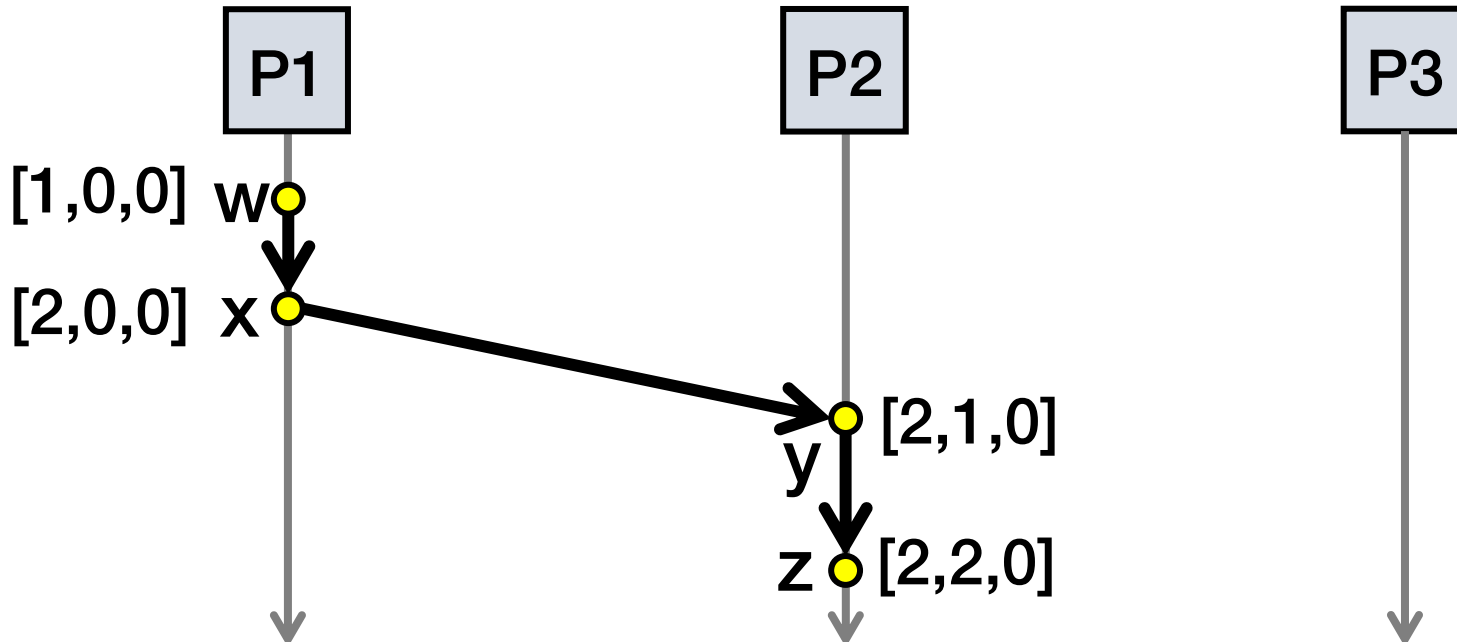


# Comparing vector timestamps

- Rule for comparing vector timestamps:
  - $V(a) = V(b)$  when  $a_k = b_k$  for all  $k$
  - $V(a) < V(b)$  when  $a_k \leq b_k$  for all  $k$  and  $V(a) \neq V(b)$
- Concurrency:
  - $V(a) \parallel V(b)$  if  $a_i < b_i$  and  $a_j > b_j$ , some  $i, j$

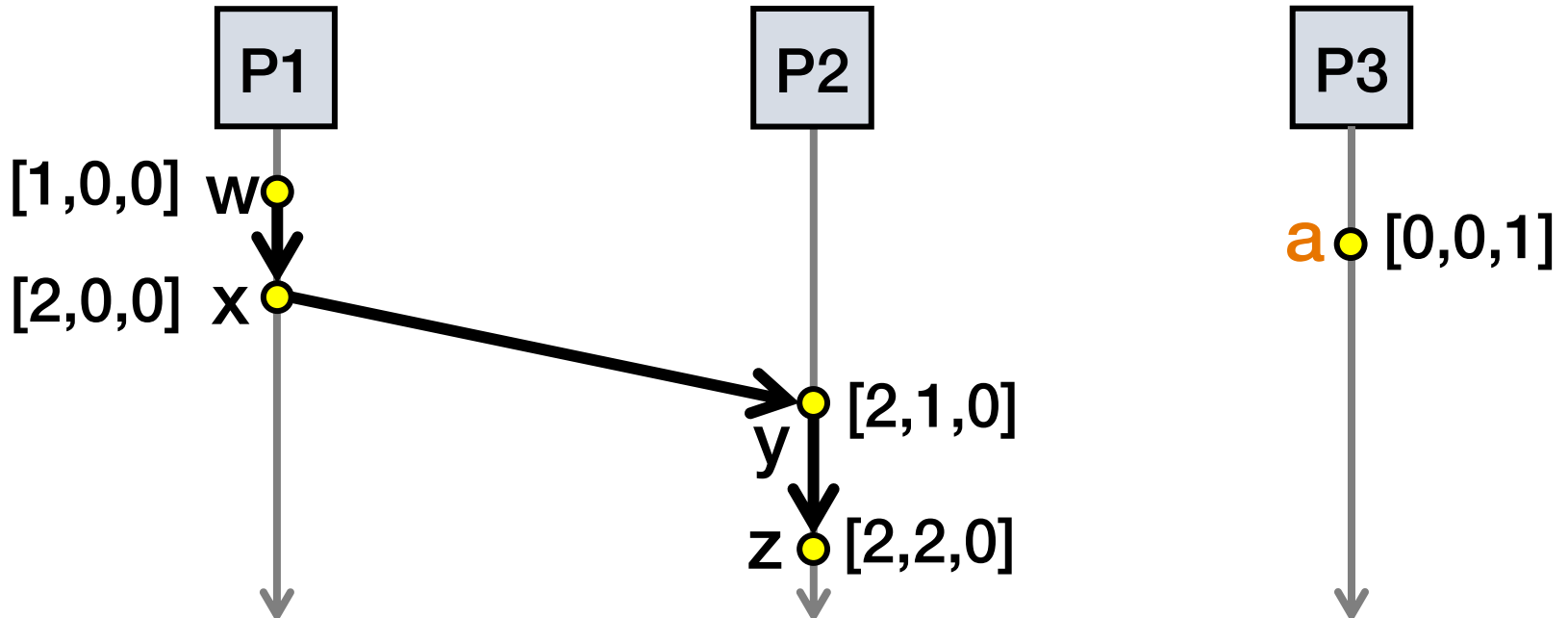
# Vector clocks capture causality

- $V(w) < V(z)$  then there is a chain of events linked by Happens-Before ( $\rightarrow$ ) between  $w$  and  $z$



# Vector clocks capture causality

- $V(w) < V(z)$  then there is a chain of events linked by Happens-Before ( $\rightarrow$ ) between  $w$  and  $z$
- $V(a) \parallel V(w)$  then there is **no** such chain of events between  $a$  and  $w$



# Comparing vector timestamps

- Rule for comparing vector timestamps:
  - $V(a) = V(b)$  when  $a_k = b_k$  for all  $k$ 
    - They are the same event
  - $V(a) < V(b)$  when  $a_k \leq b_k$  for all  $k$  and  $V(a) \neq V(b)$ 
    - $a \rightarrow b$
- Concurrency:
  - $V(a) \parallel V(b)$  if  $a_i < b_i$  and  $a_j > b_j$ , some  $i, j$ 
    - $a \parallel b$

Two events  $a, z$

Lamport clocks:  $C(a) < C(z)$

Conclusion:  $z \not\rightarrow a$ , i.e., either  $a \rightarrow z$  or  $a \parallel z$

Vector clocks:  $V(a) < V(z)$

Conclusion:  $a \rightarrow z$

Two events  $a, z$

Lamport clocks:  $C(a) < C(z)$

Conclusion:  $z \not\rightarrow a$ , i.e., either  $a \rightarrow z$  or  $a \parallel z$

Vector clocks:  $V(a) < V(z)$

Conclusion:  $a \rightarrow z$

Vector clock timestamps precisely capture happens-before relation (potential causality)