Time & Clocks

CS 4740: Cloud Computing

Fall 2024

Lecture 8

Yue Cheng

UNIVERSITY *of* VIRGINIA

Outline

- **The need for time synchronization**

- "Wall clock time" synchronization

- Logical Time: Lamport Clocks

- Vector clocks

A cloud edit-compile workflow

Computer on which compiler runs — Time according to local clock

2144   2145   2146   2147

output.o created

Computer on which editor runs — Time according to local clock

2142   2143   2144   2145

output.c created

Physical time →

Compiler over cloud

RPC

Editor

A cloud edit-compile workflow

Computer on which compiler runs

2144   2145   2146   2147 ← Time according to local clock

output.o created

Computer on which editor runs

2142   2143   2144   2145 ← Time according to local clock

output.c created

Physical time →

Compiler over cloud

RPC

Editor

- 2143 < 2144 ➔ `make` doesn't call compiler

A cloud edit-compile workflow

	2144	2145	2146	2147	← Time according to local clock

Computer on which compiler runs

output.o created

	2142	2143	2144	2145	← Time according to local clock

Computer on which editor runs

output.c created

Physical time →

Compiler over cloud

RPC

Editor
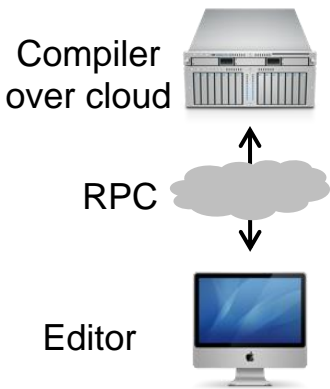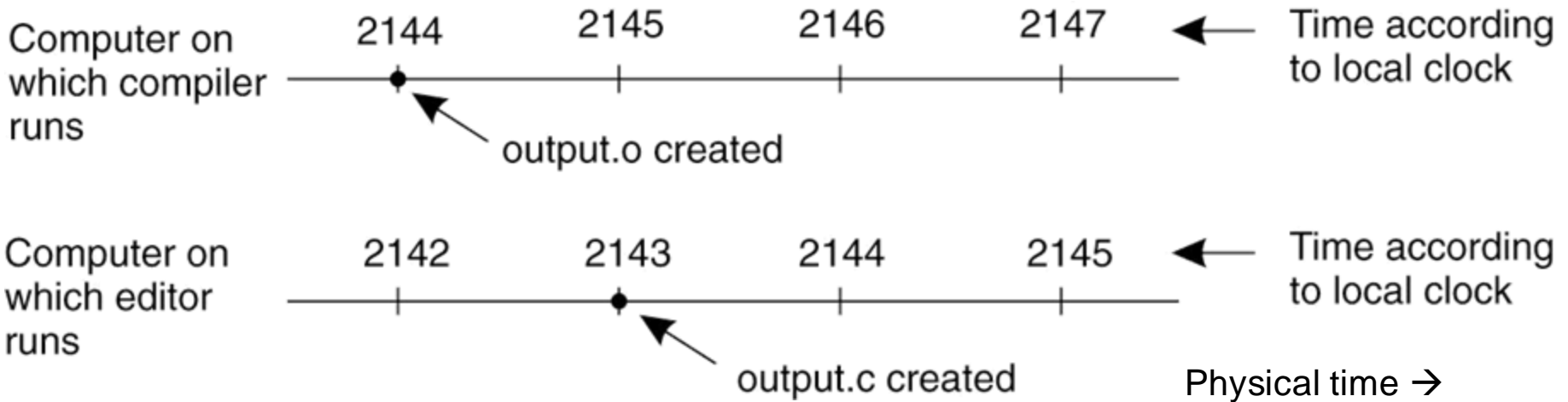
- 2143 < 2144 ➔ `make` doesn't call compiler

Lack of time synchronization – 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

Outline

- The need for time synchronization

- "Wall clock time" synchronization
  - Cristian's algorithm

- 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:

Client

Server

Time of day?

2:50 PM

Time ↓

Synchronization to a time server

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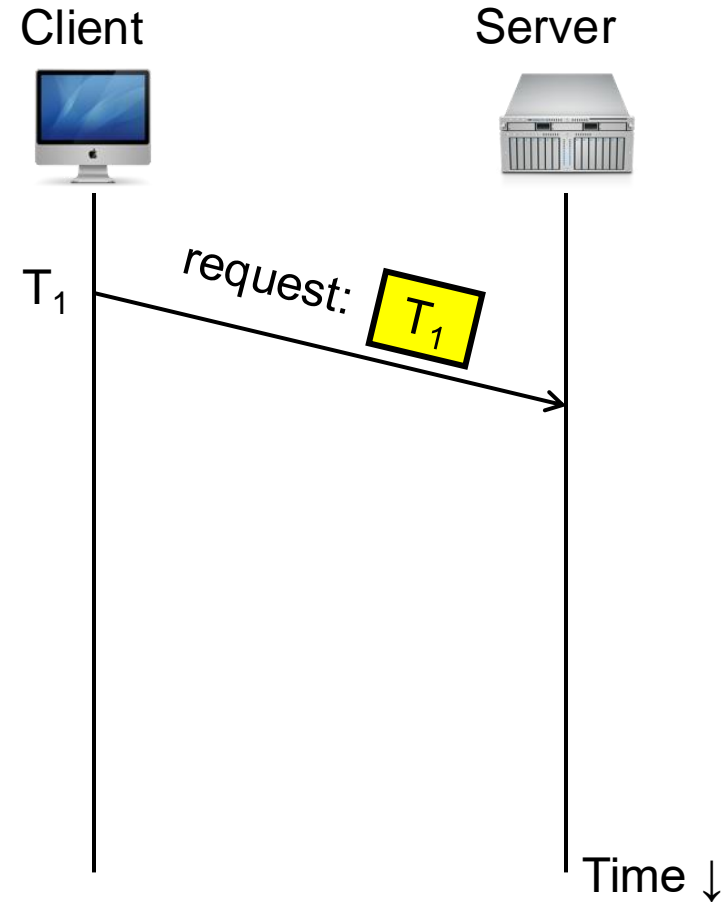 - Could simply issue an RPC to obtain the time:

Client   Server

Time of day?

2:50 PM
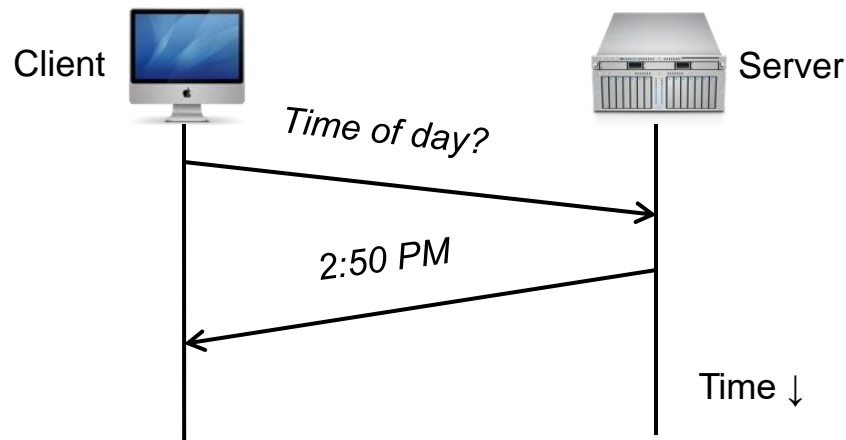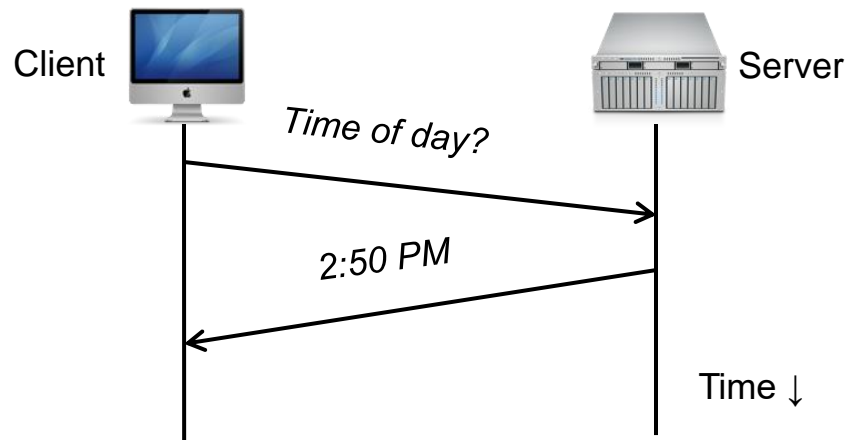
Time ↓

- 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

Client   Server

T_1 request: T_1

Time ↓

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

Client   Server

T_1 request: T_1

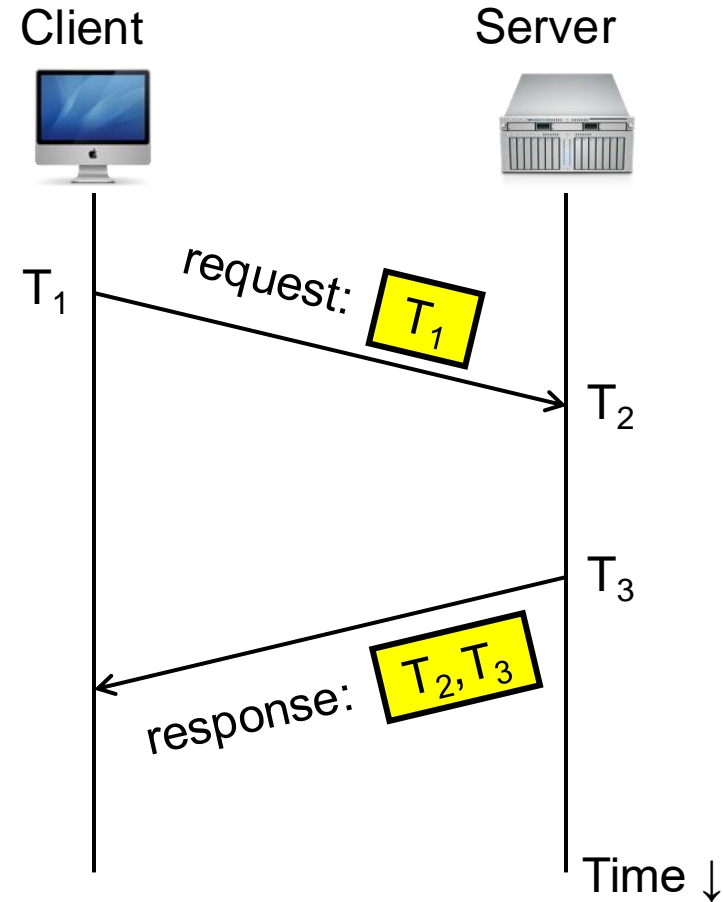 T_2

Time ↓

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

Client   Server

T_1 → request: T_1 → T_2
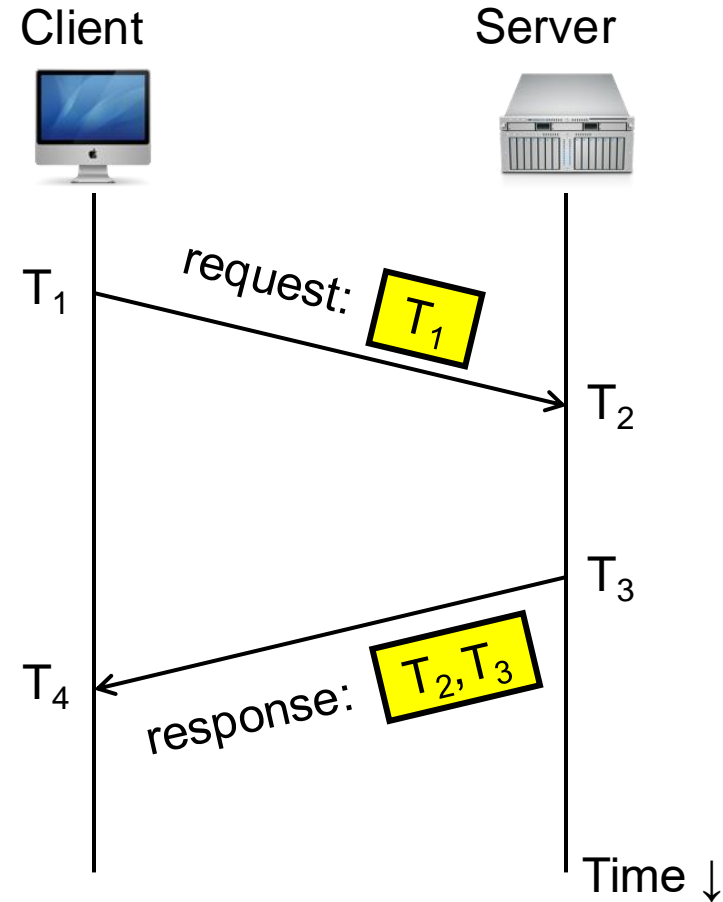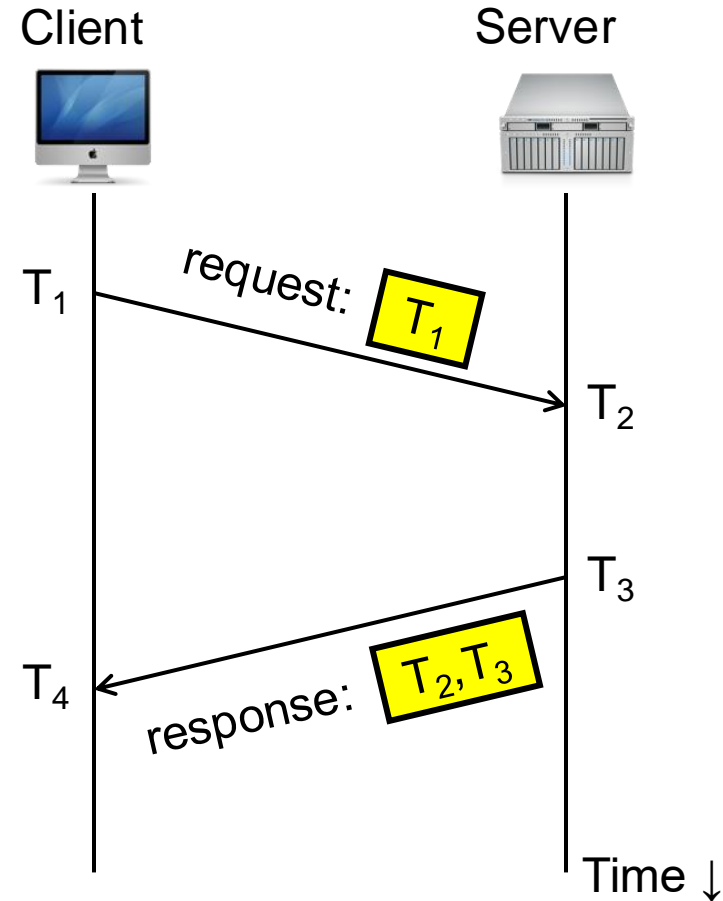
T_3

response: T_2, T_3

Time ↓

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**

Client   Server

T_1 request: T_1

T_2

T_3

T_4 response: T_2,T_3

Time ↓

Cristian's algorithm: Outline

Client   **Server**

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

T_1 — request: T_1 → T_2

T_3

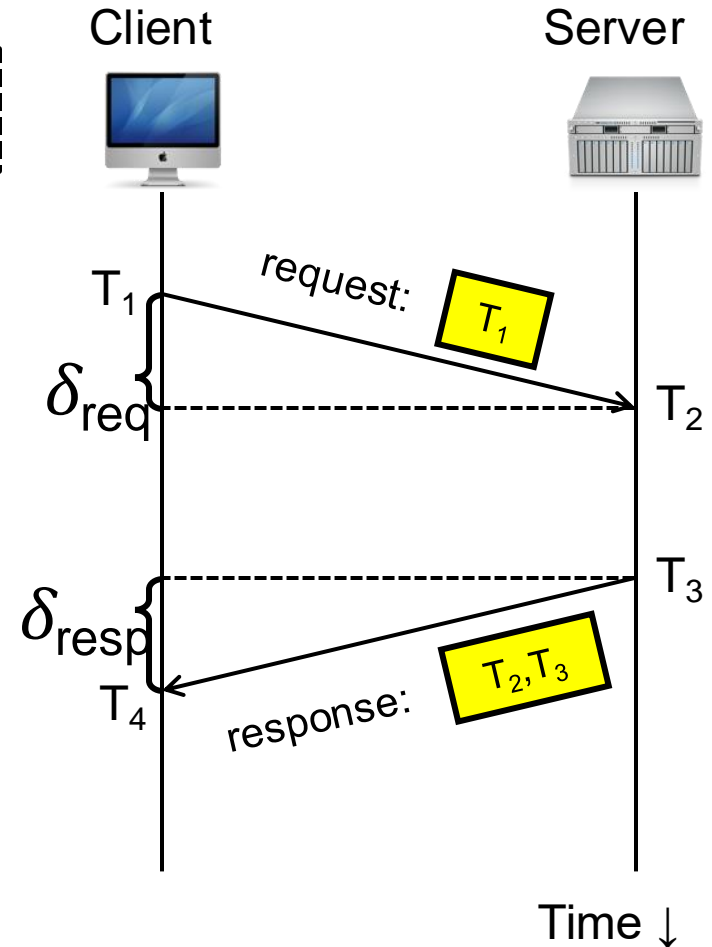T_4 ← response: T_2, T_3

Time ↓

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 $\leftarrow T_3 + \delta_{resp}$

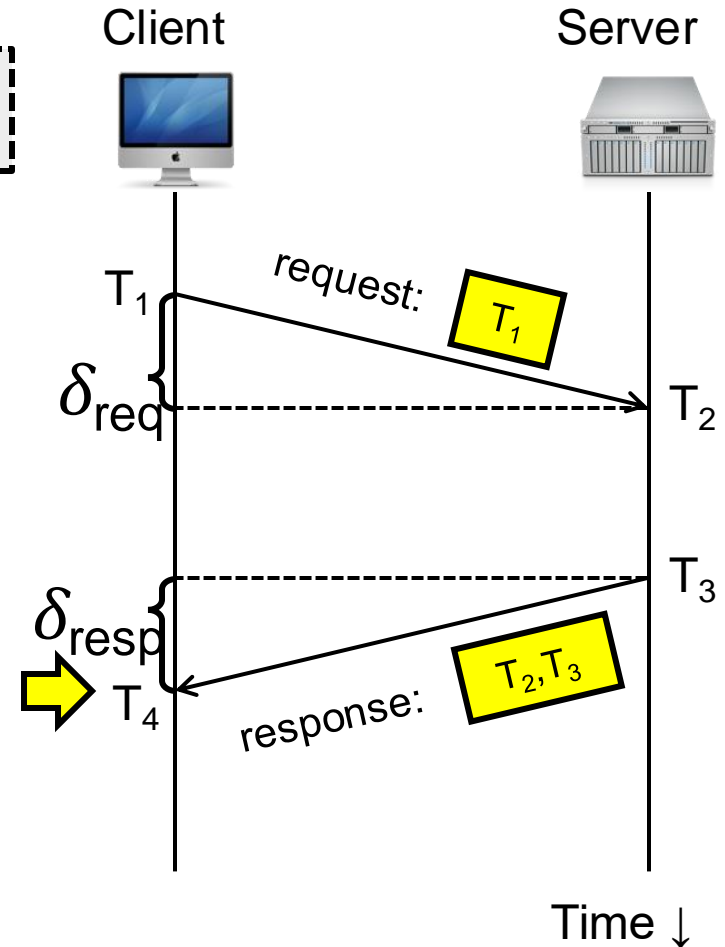- Client samples round trip time δ = $\delta_{req} + \delta_{resp} = (T_4 - T_1) - (T_3 - T_2)$

Client   Server

T_1 request: T_1

δ_{req} T_2

δ_{resp} T_3

T_4 response: T_2, T_3

Time ↓

Cristian's algorithm: Offset sample calculation

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

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

Client   Server

T_1

request: T_1

δ_{req}

T_2

T_3

δ_{resp}

T_2, T_3

T_4

response:

Time ↓

Cristian's algorithm: Offset sample calculation

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

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

- But client knows δ, not δ_{resp}

Client

Server

T_1

request: T_1

δ_{req} → T_2
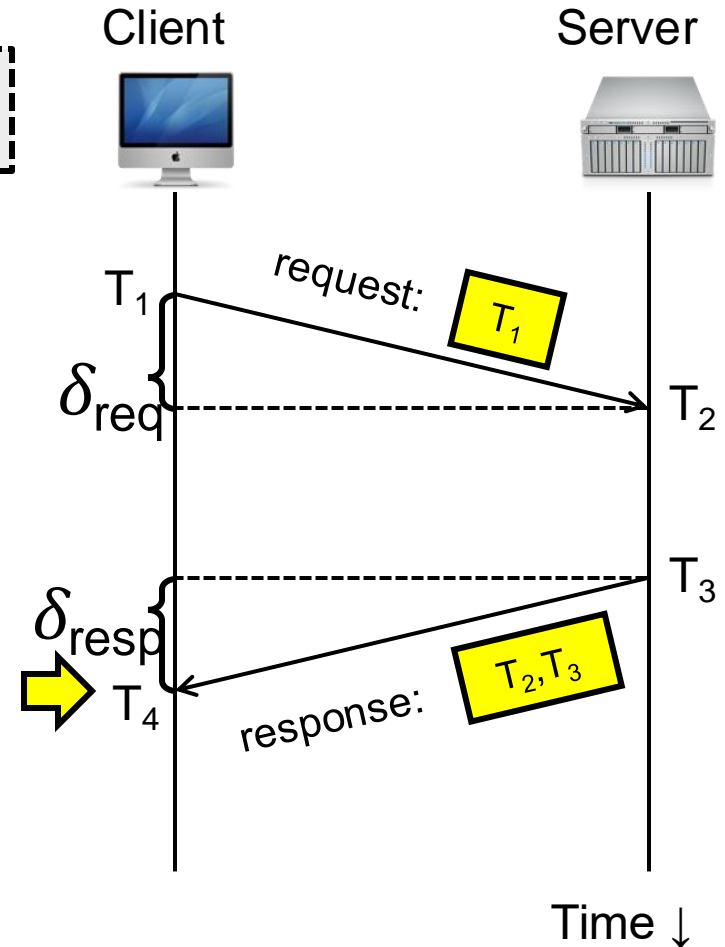
T_3

δ_{resp} → T_4

response: T_2, T_3

Time ↓

Cristian's algorithm: Offset sample calculation

> Goal: Client sets clock ← $T_3 + \delta_{resp}$

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

- But client knows δ, not δ_{resp}

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

Client   Server

T_1

request: T_1

δ_{req} T_2

T_3

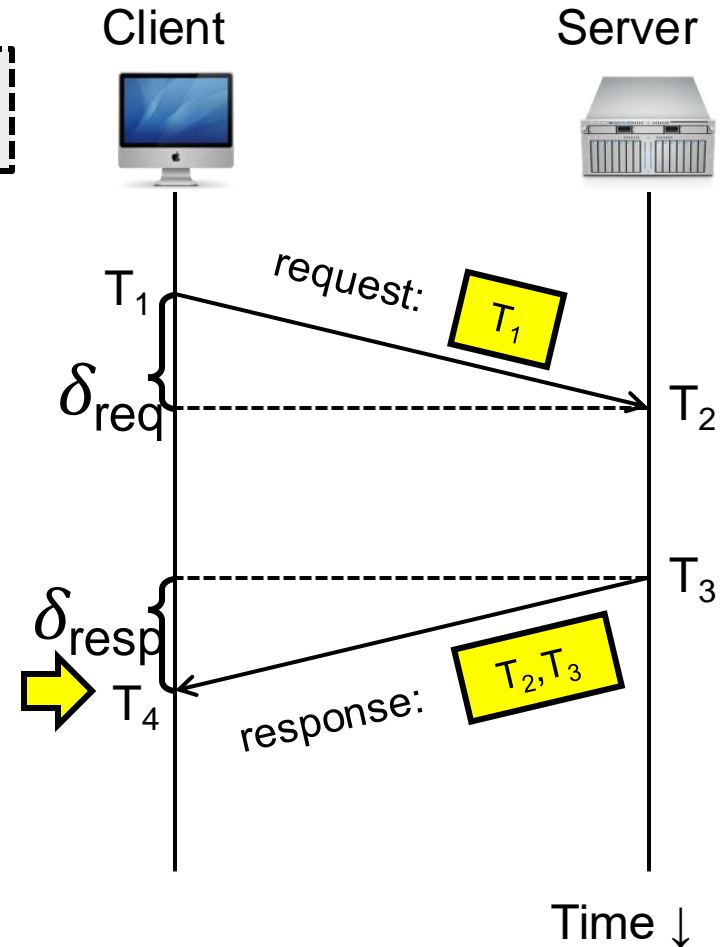δ_{resp}

T_4 response: T_2, T_3

Time ↓

Cristian's algorithm: Offset sample calculation

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

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

- But client knows δ, not δ_{resp}

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

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

Client

Server

T_1

request: $\boxed{T_1}$

δ_{req}

T_2

T_3

δ_{resp}

T_4

response: $\boxed{T_2, T_3}$

Time $\downarrow$

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

- Clock synchronization algorithms
  - Rely on timestamps to estimate network delays
  - 100s μs−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

- Clock synchronization algorithms
  - Rely on timestamps to estimate network delays
  - 100s μs−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

Outline

- The need for time synchronization

- "Wall clock time" synchronization
  - Cristian's algorithm

- **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
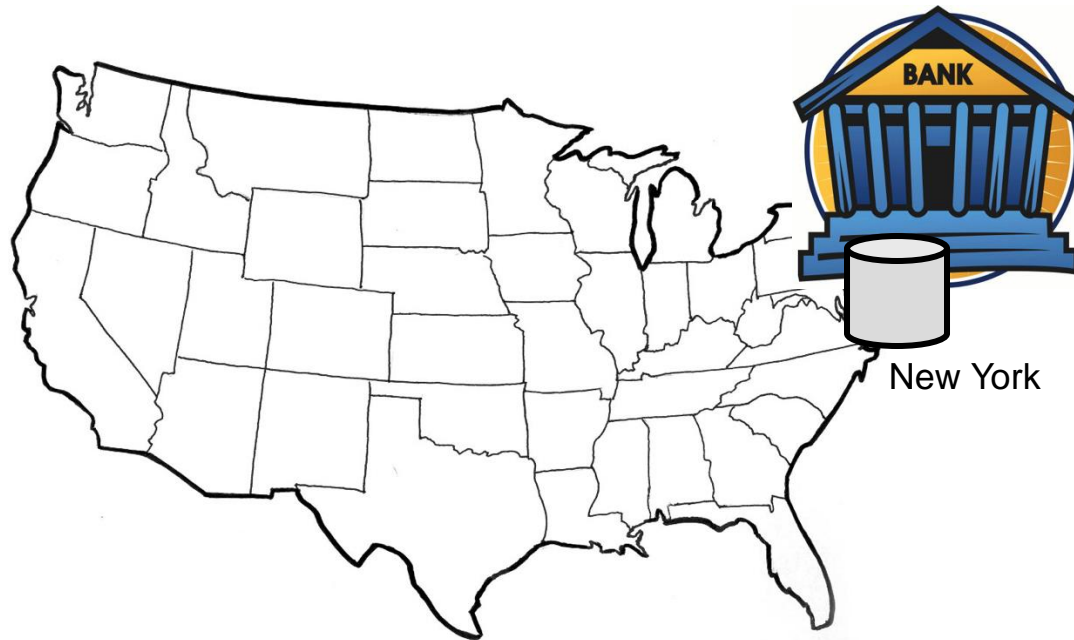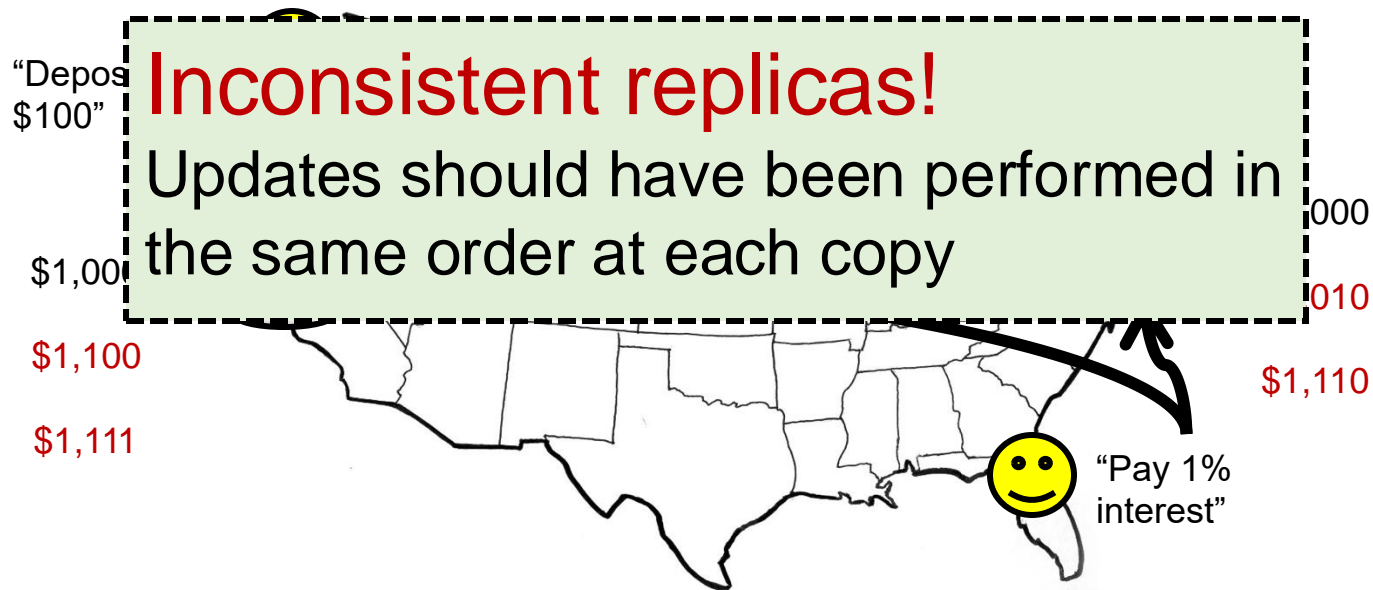
New York

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
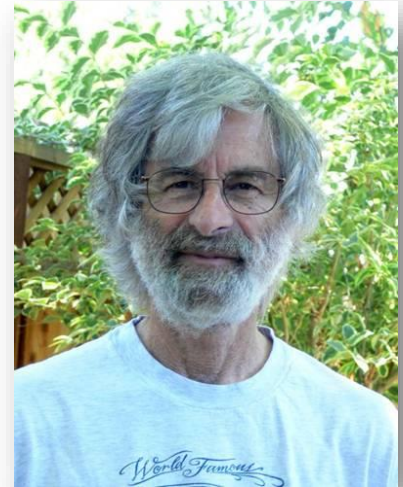

San Francisco

New York

The consequences of concurrent updates

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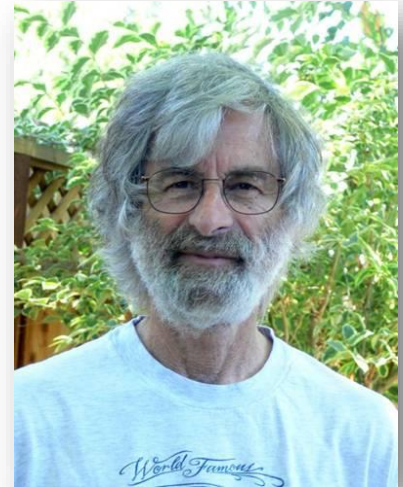 - Client sends update to both copies

"Depos
$100"

Inconsistent replicas!
Updates should have been performed in the same order at each copy

$1,00

000

010

$1,100

$1,110

$1,111

"Pay 1% interest"

Idea: Logical clocks

- Landmark 1978 paper by **Leslie Lamport**

Idea: Logical clocks

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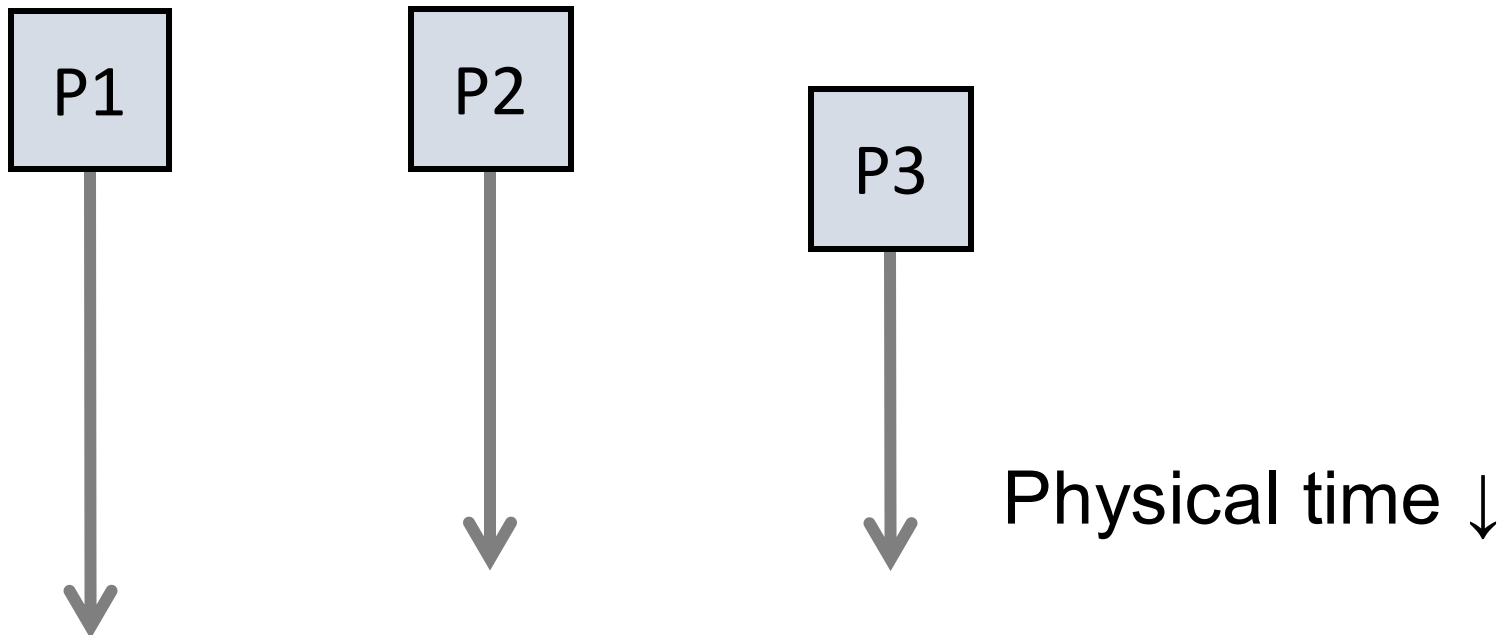
- **Insights:** only the events themselves matter

Idea: Disregard the precise clock time

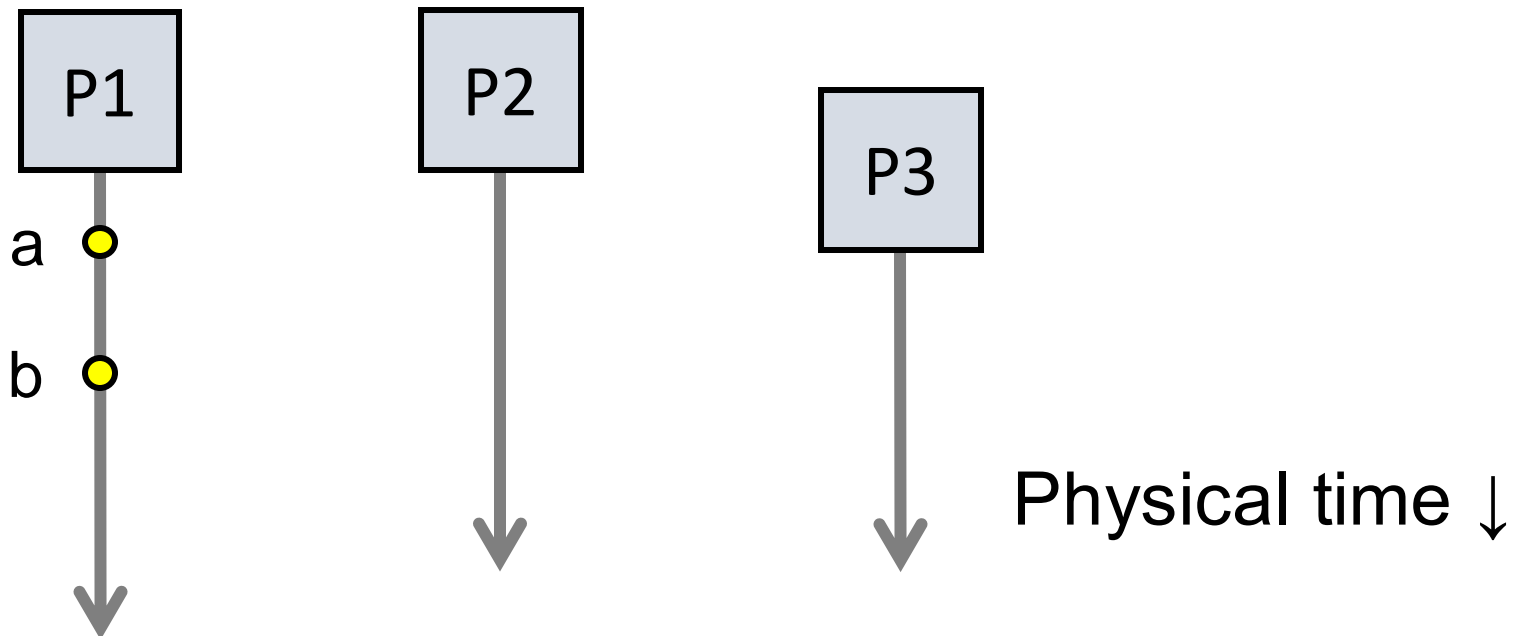Instead, capture just a "**happens before**" relationship between a pair of events

Defining "happens-before" ($\rightarrow$)

- Consider three processes: P1, P2, and P3

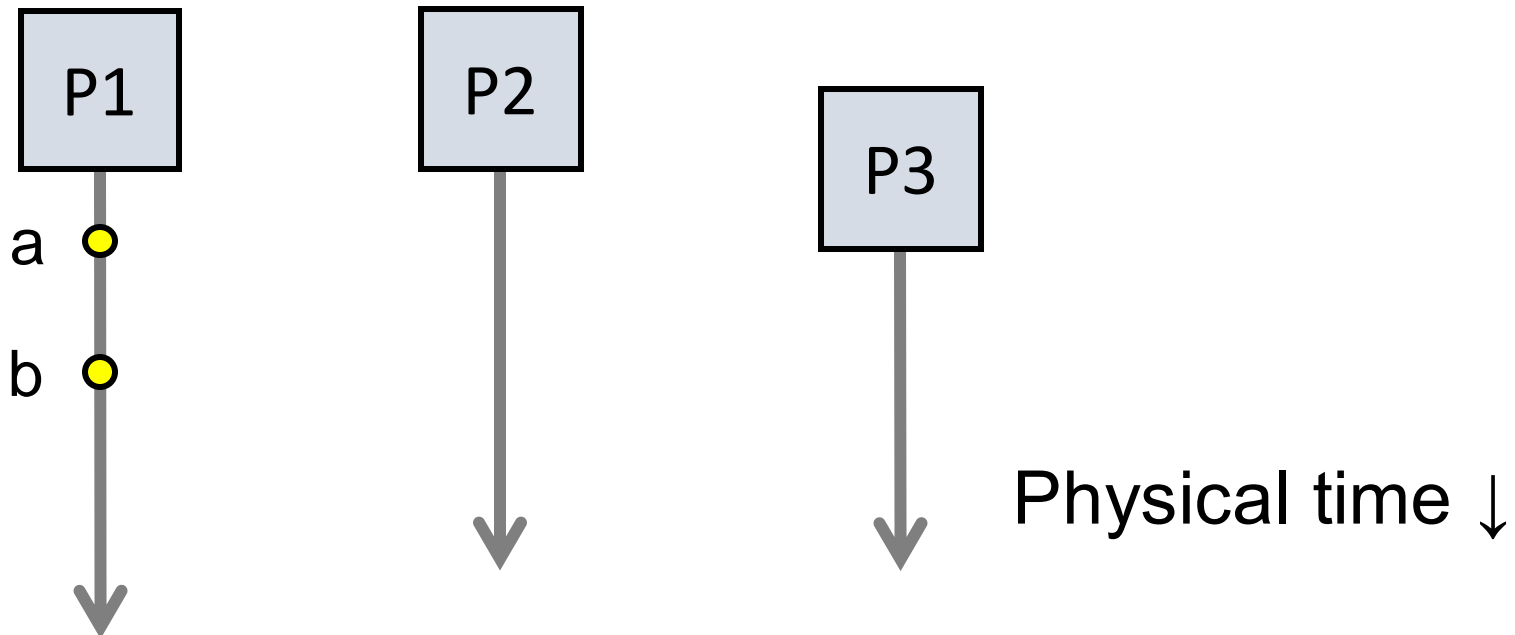- Notation: Event a happens before event b (a $\rightarrow$ b)

P1

P2

P3

Physical time ↓

Defining "happens-before" ($\rightarrow$)

- Can observe event order at a single process

P1

P2

P3

a

b

Physical time ↓

Defining "happens-before" ($\rightarrow$)

1. If **same process** and **a** occurs before **b**, then **a** $\rightarrow$ **b**
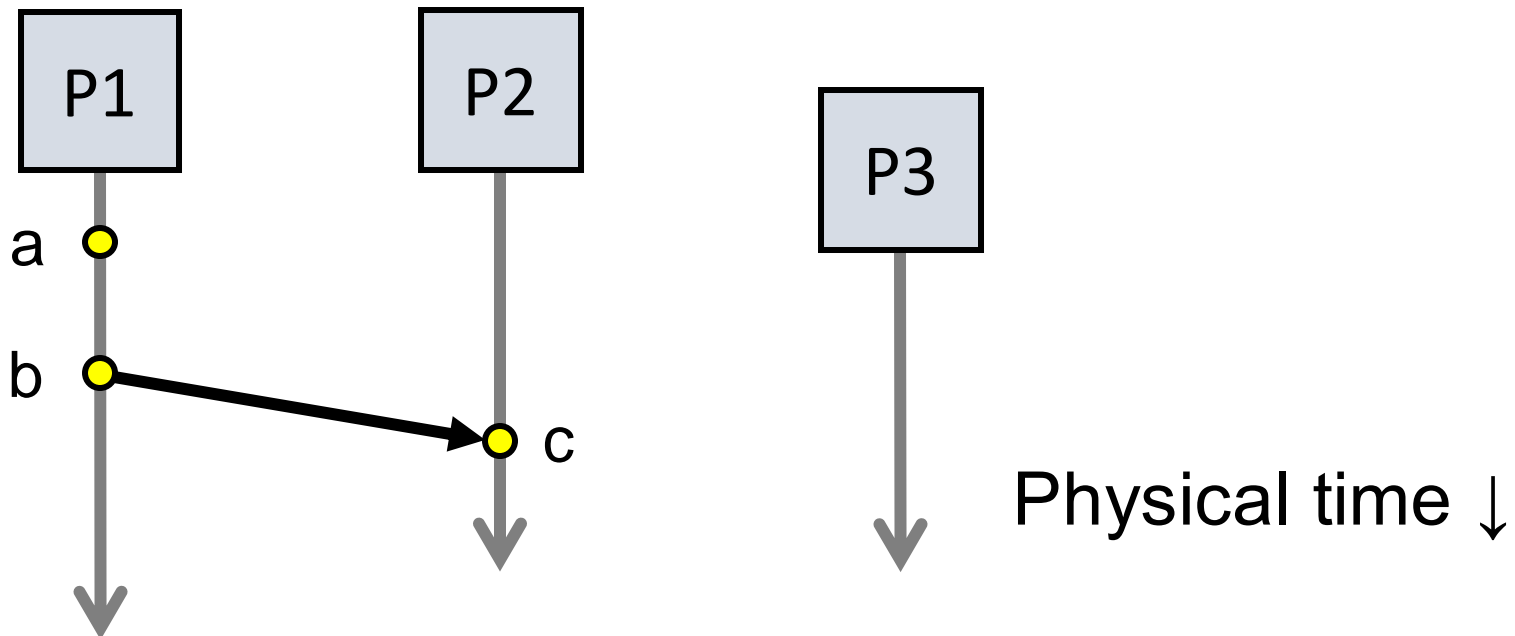
P1

P2

P3

a

b

Physical time ↓

Defining "happens-before" ($\rightarrow$)

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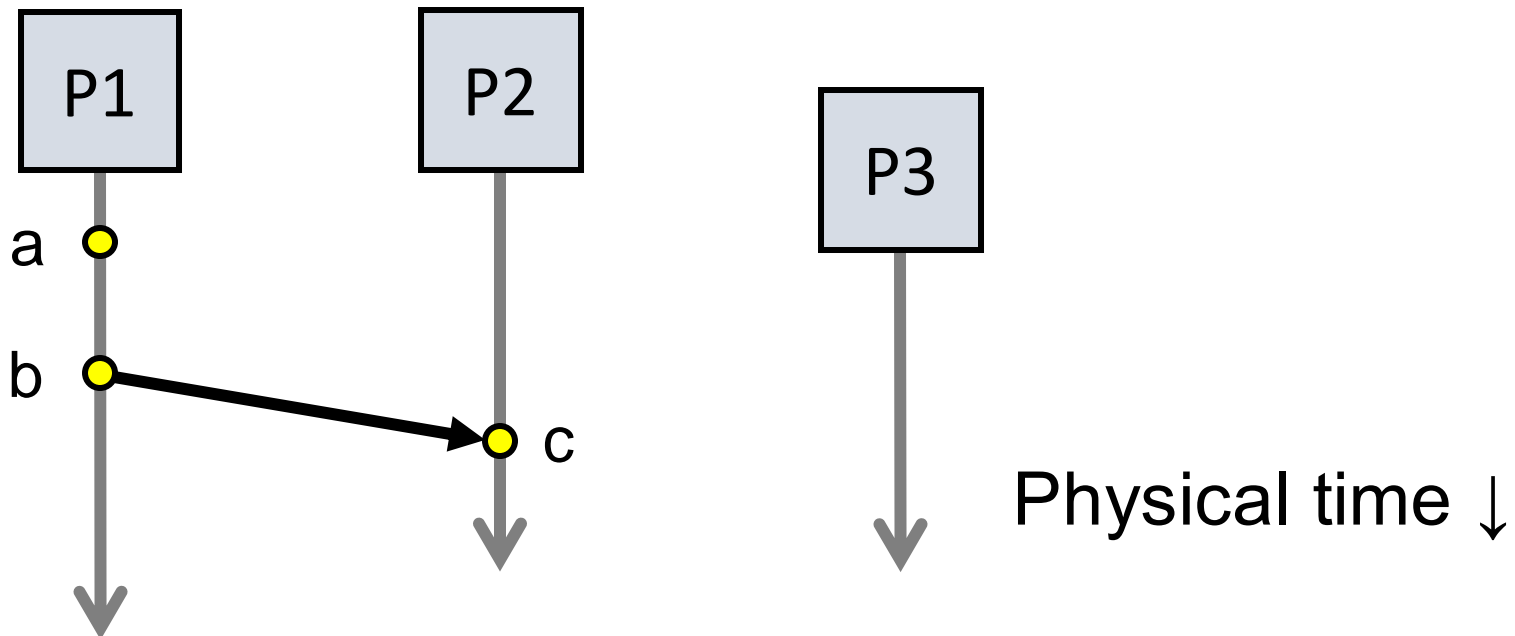
2. Can observe ordering when processes communicate
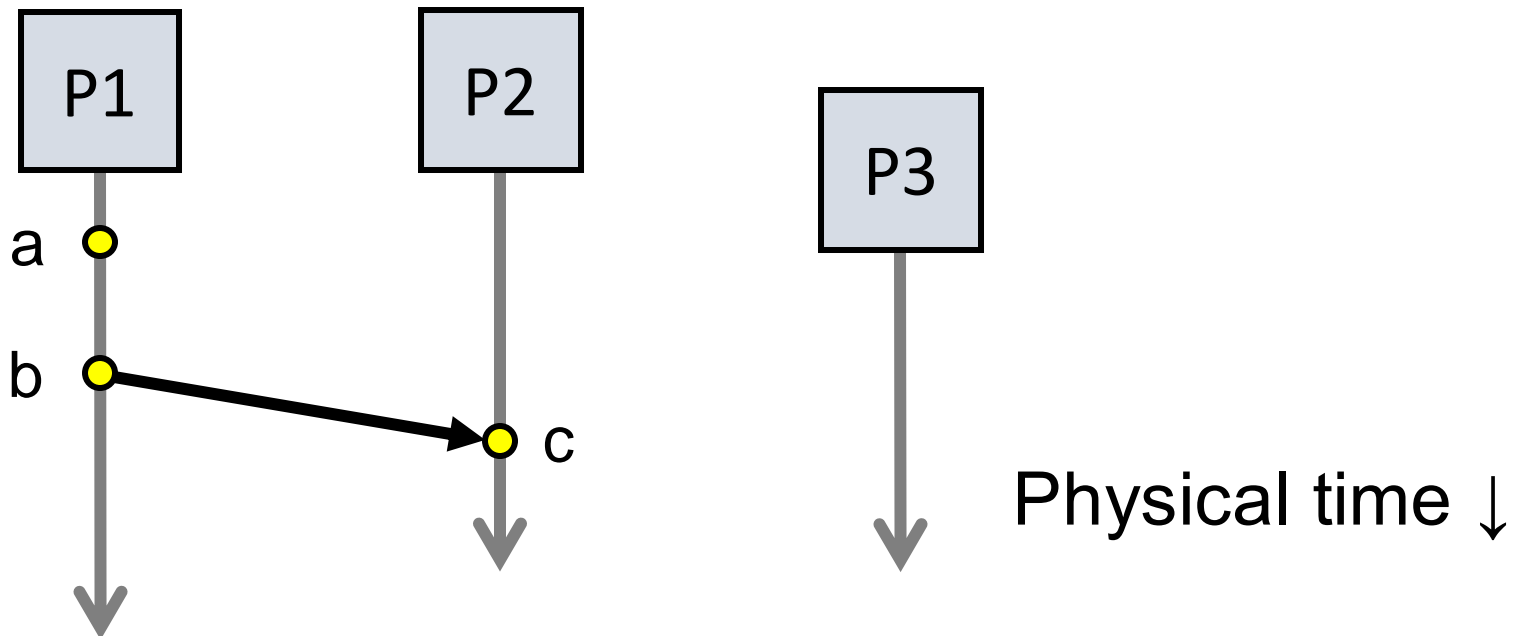

P1   P2   P3

a

b

c

Physical time ↓

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

P1

P2

P3

a

b

c

Physical time ↓

Defining "happens-before" (→)

1. If **same process** and **a** occurs before **b**, then **a** → **b**

2. If **c** is a message receipt of **b**, then **b** → **c**

3. Can observe ordering transitively

P1

P2

P3

a

b

c

Physical time ↓

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**
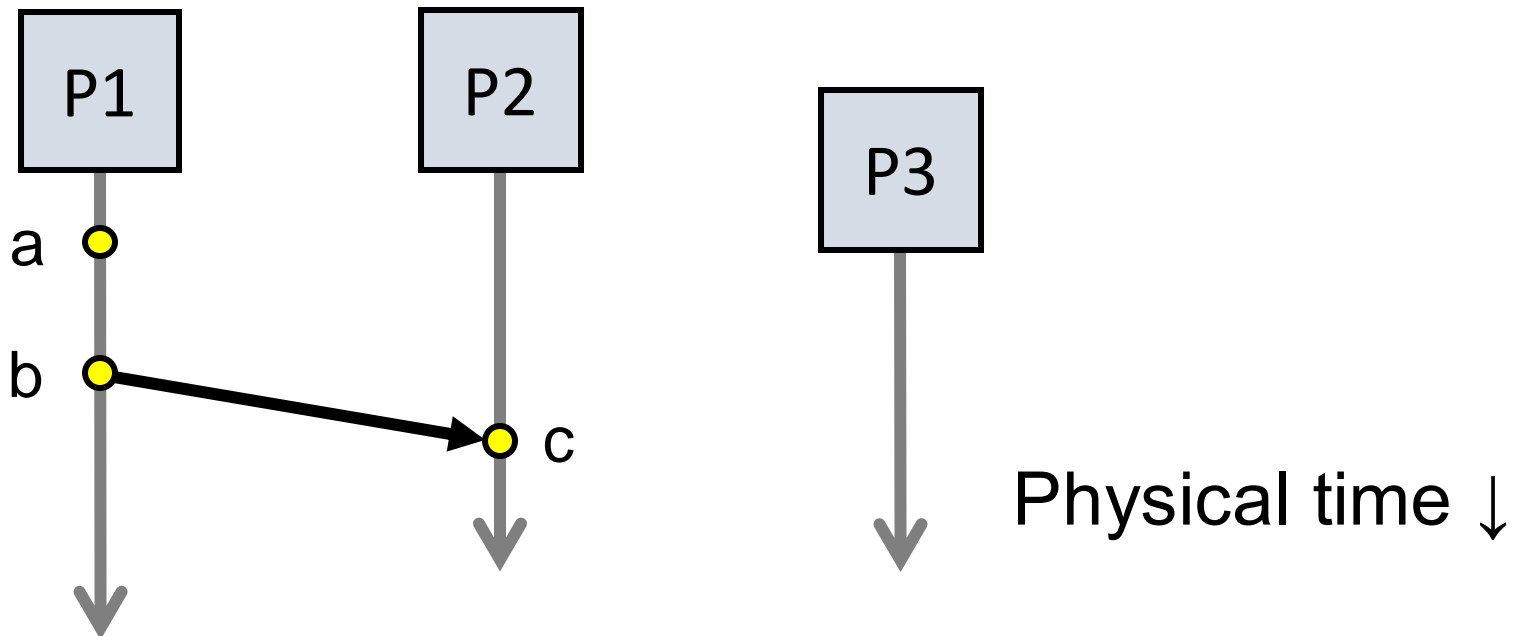
P1

P2

P3

a

b

c

Physical time ↓

Defining "happens-before" ($\rightarrow$)

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

P1
P2
P3

a
b
c
d

Physical time ↓
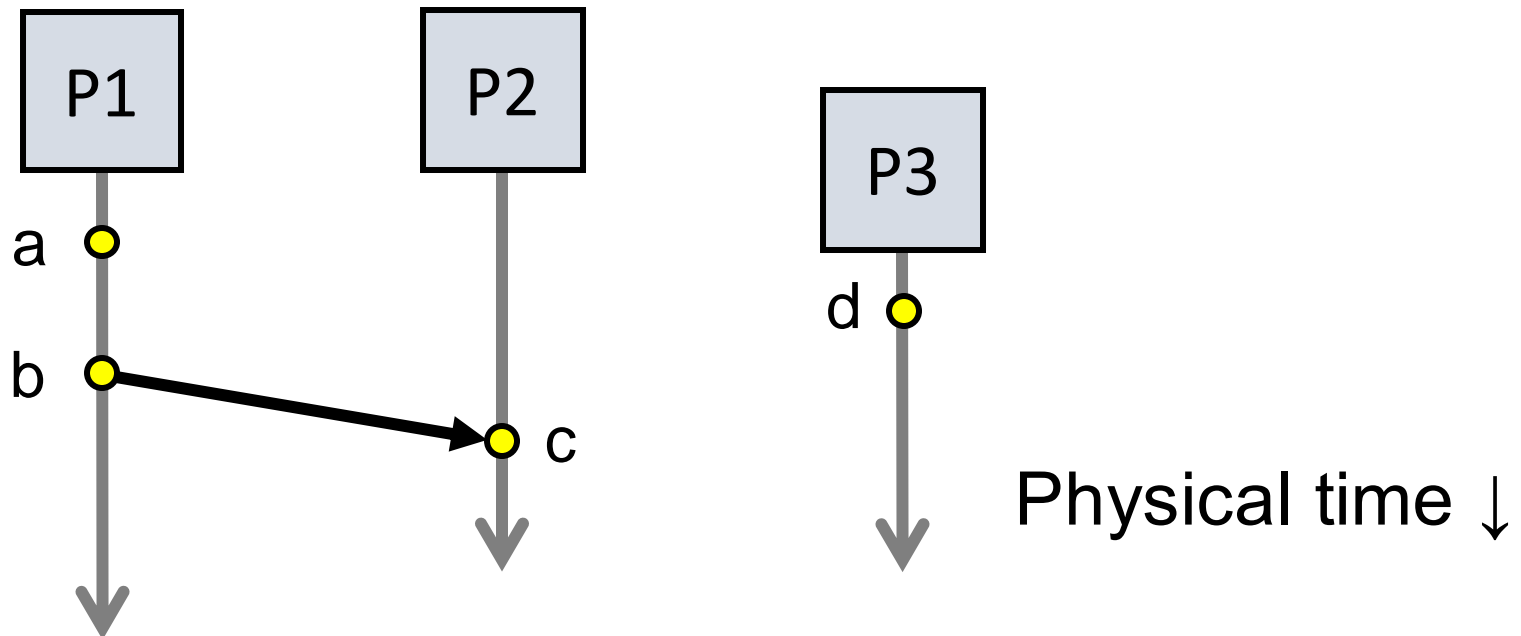
Defining "happens-before" ($\rightarrow$)

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

2. a, d not related by $\rightarrow$ so concurrent, written as **a || d**

P1   P2   P3

a

b

c

d

Physical time ↓
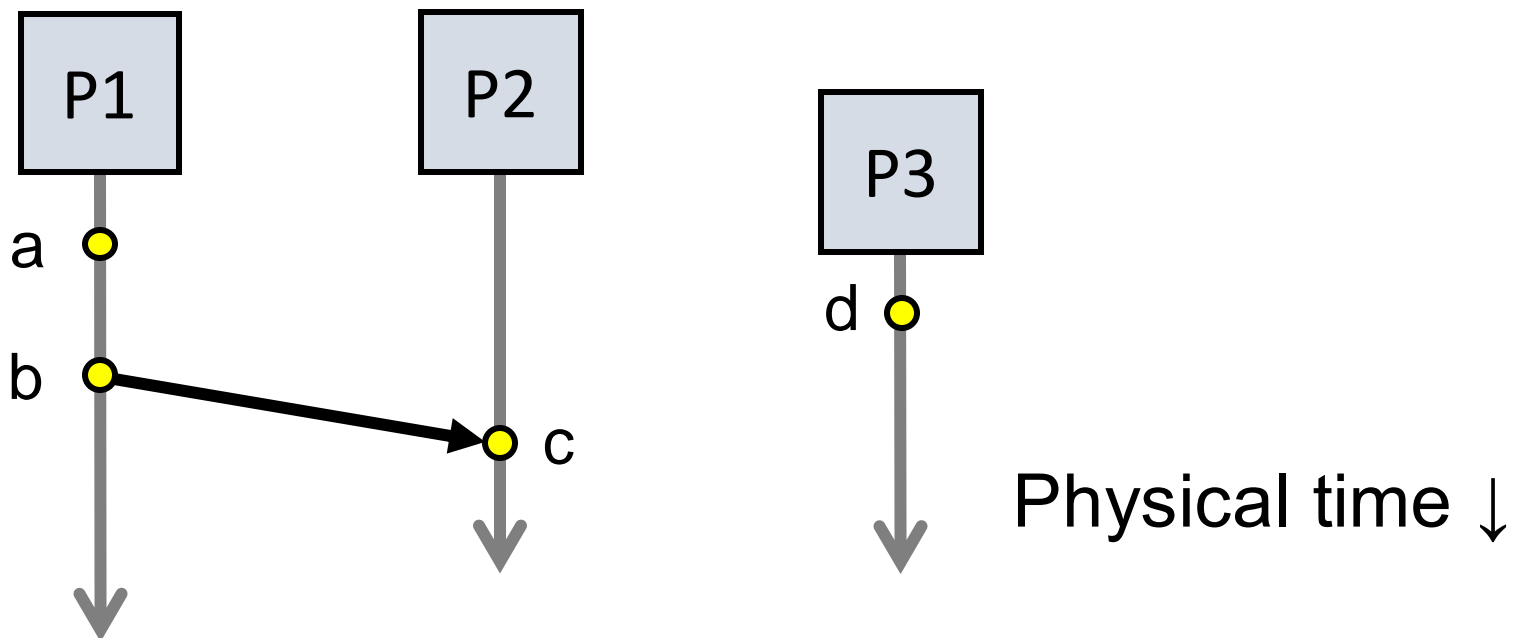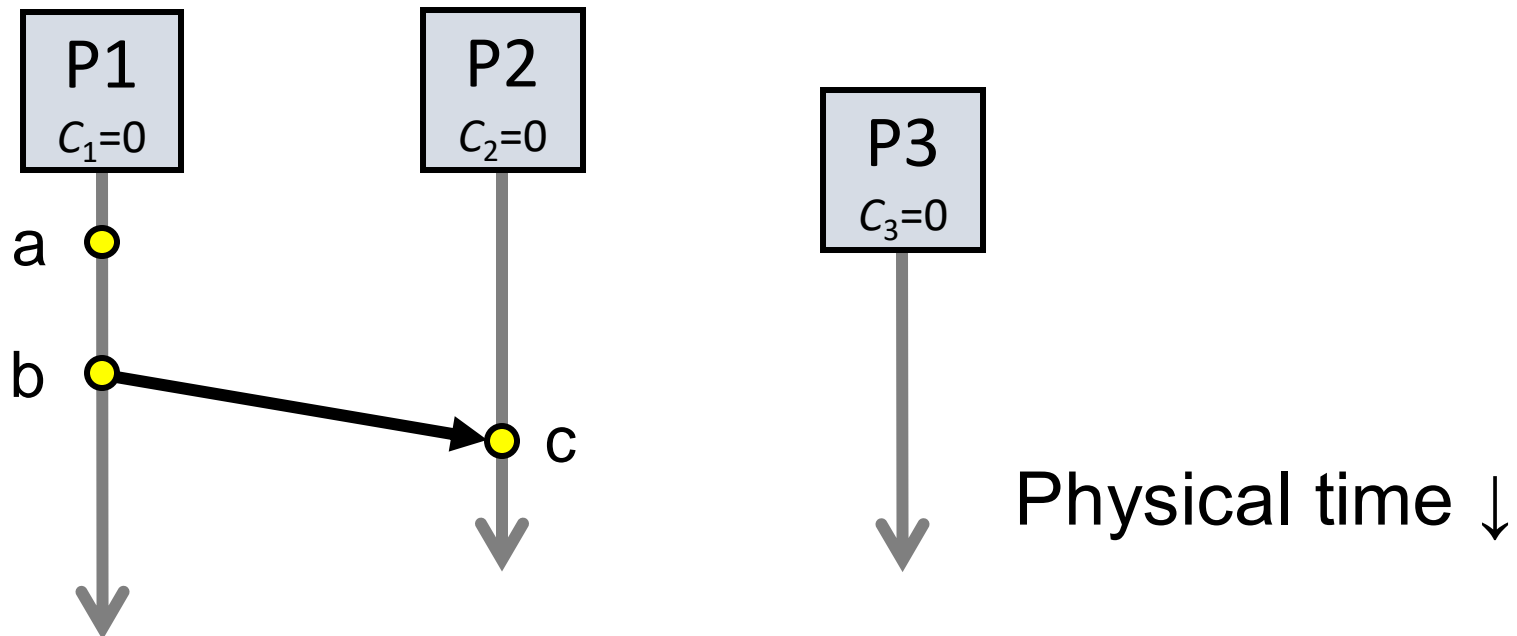
Lamport clocks: Objective

- 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 → 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, $C_i \leftarrow C_i + 1$:

P1
$C_1=0$

P2
$C_2=0$

P3
$C_3=0$

a

b

c

Physical time ↓

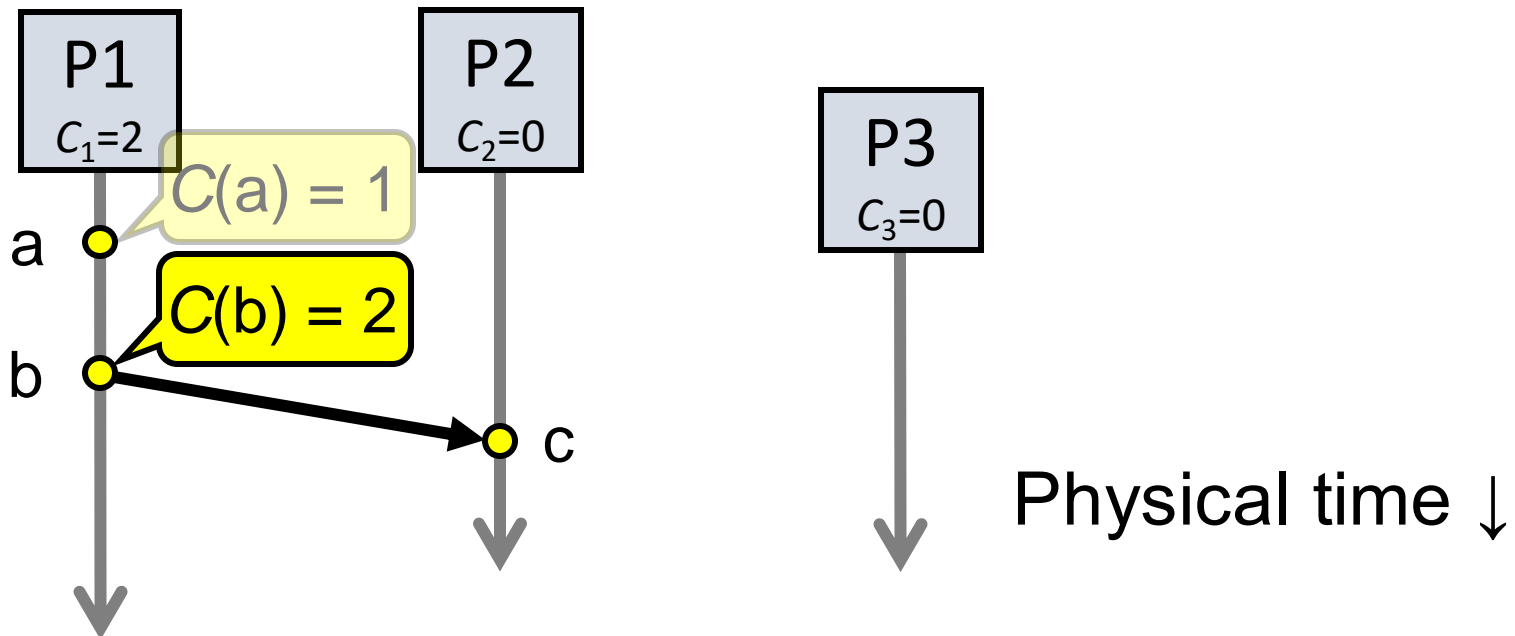# The Lamport Clock algorithm

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

   - Set event time $C(a) \leftarrow C_i$
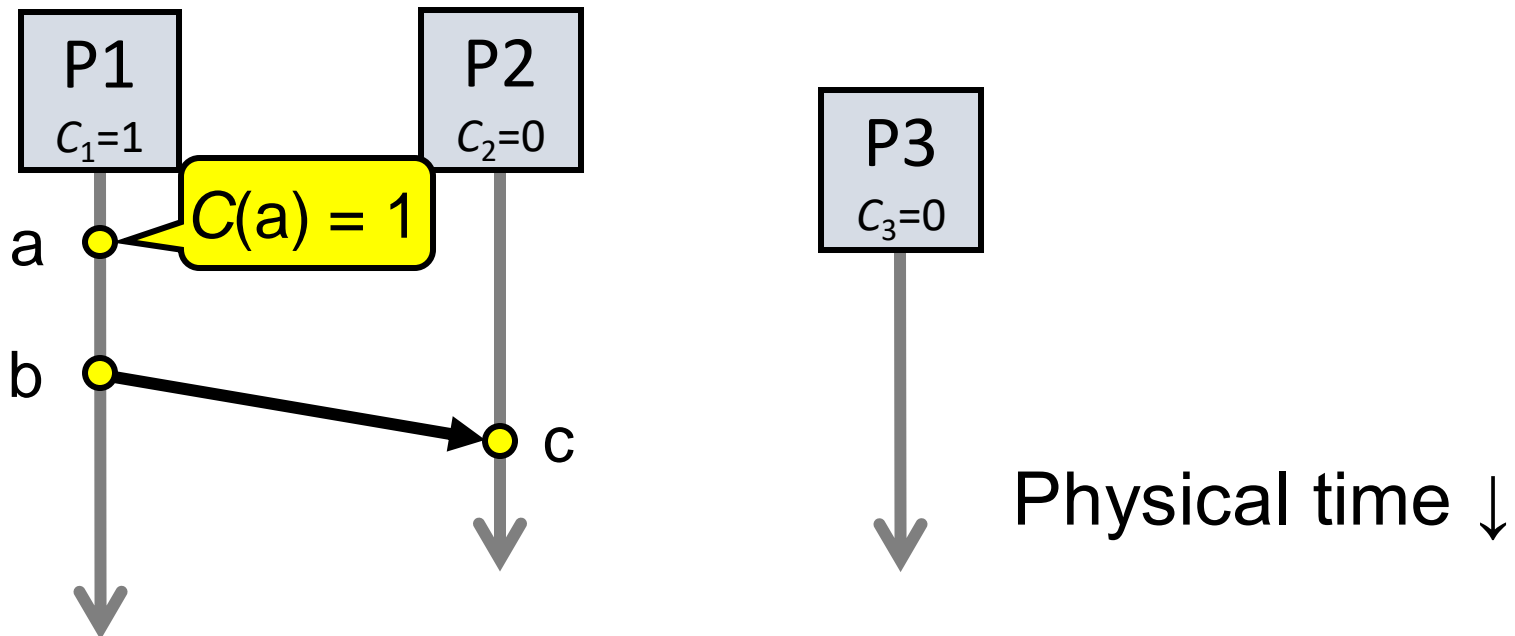
The Lamport Clock algorithm

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

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

Physical time ↓
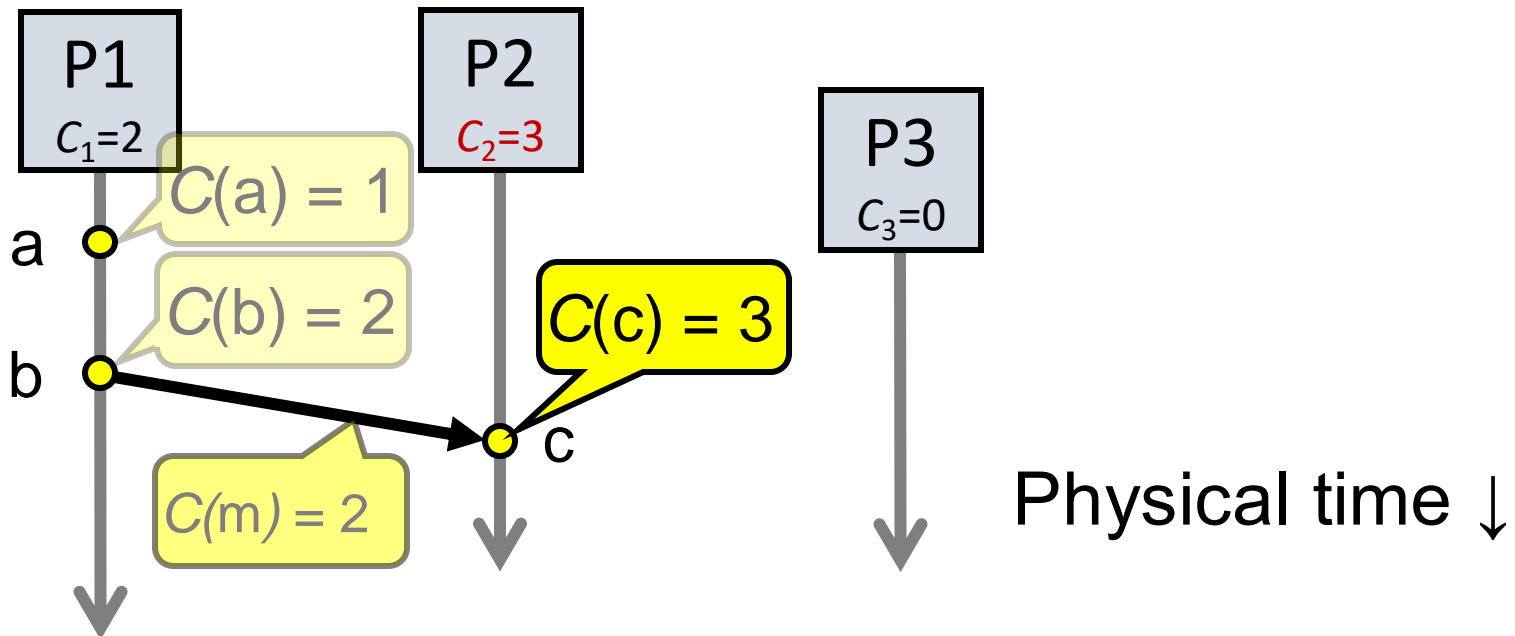
The Lamport Clock algorithm

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

2. Send the local clock in the message m

Physical time ↓
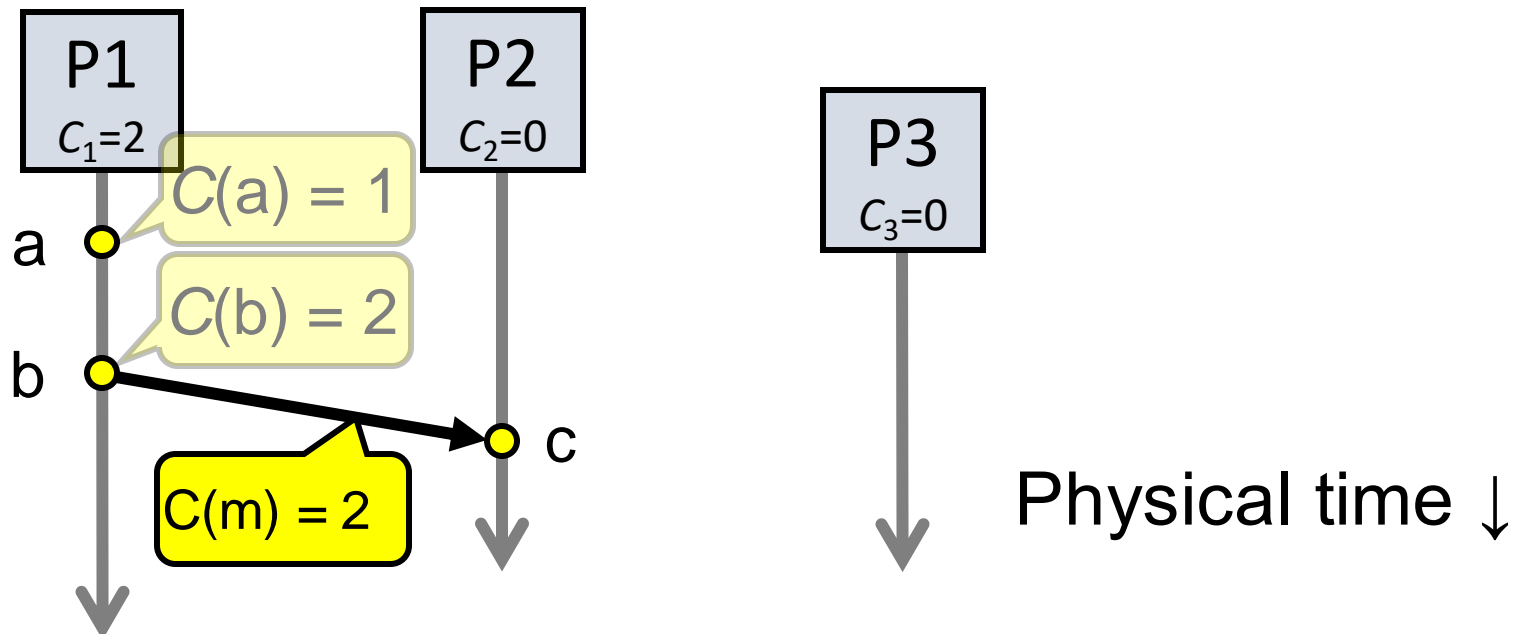
The Lamport Clock algorithm

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
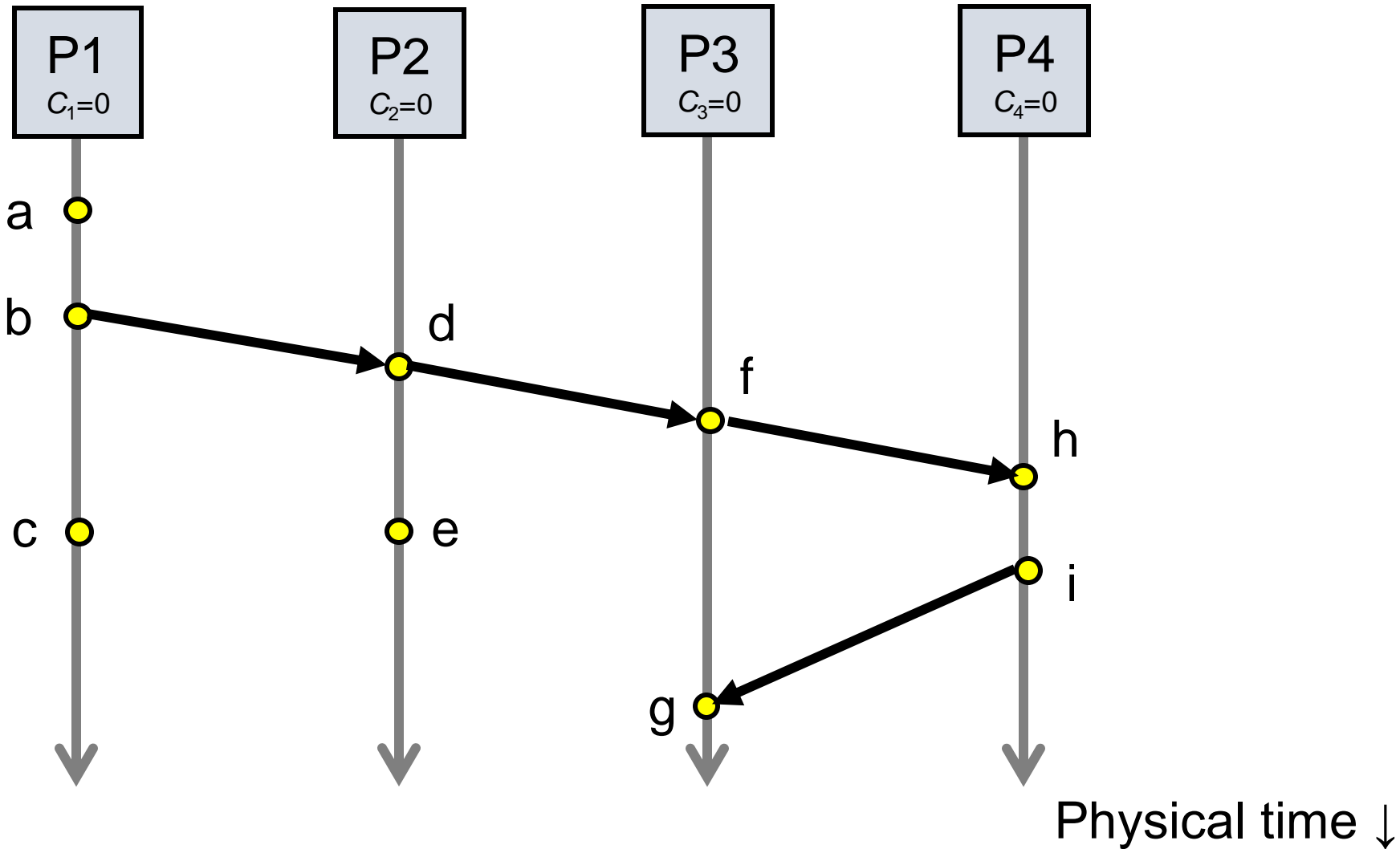
P1 $C_1=0$

P2 $C_2=0$

P3 $C_3=0$

P4 $C_4=0$

a

b

c

d

e

f

g

h

i

Physical time ↓

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 || 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

Outline

- The need for time synchronization

- "Wall clock time" synchronization
  - Cristian's algorithm

- 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:

$$a \rightarrow b \rightarrow \ldots \rightarrow y \rightarrow z$$

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) = $[c_1, c_2 \ldots, c_n]$
    - Each entry c_k is a count of events in process k that causally precede e
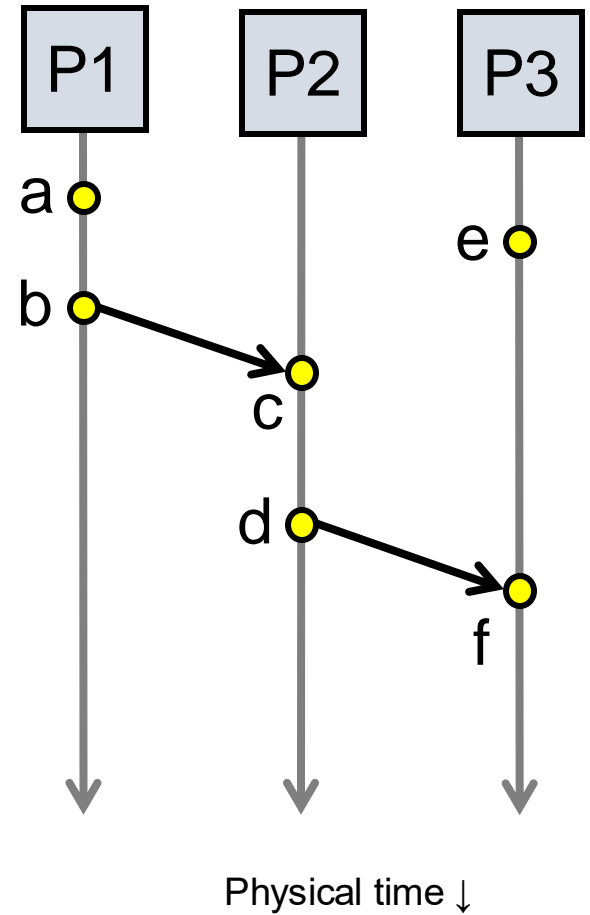
Vector clock: Update rules

- Initially, all vectors are [0, 0, …, 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, …, 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 $[d_1, d_2, …, d_n]$:
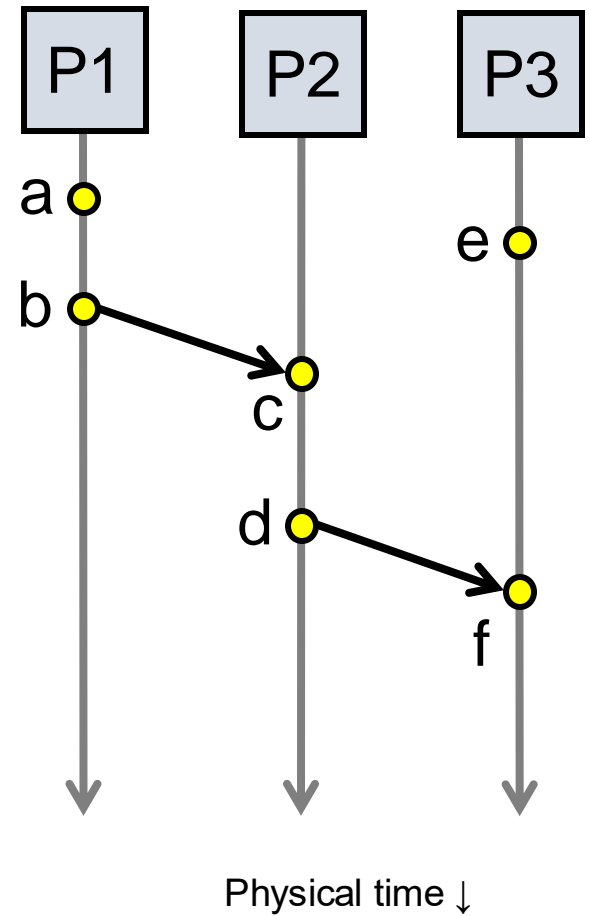   - Set each local entry $c_k = \max\{c_k, d_k\}$
   - Increment local entry c_j

Vector clock: Example
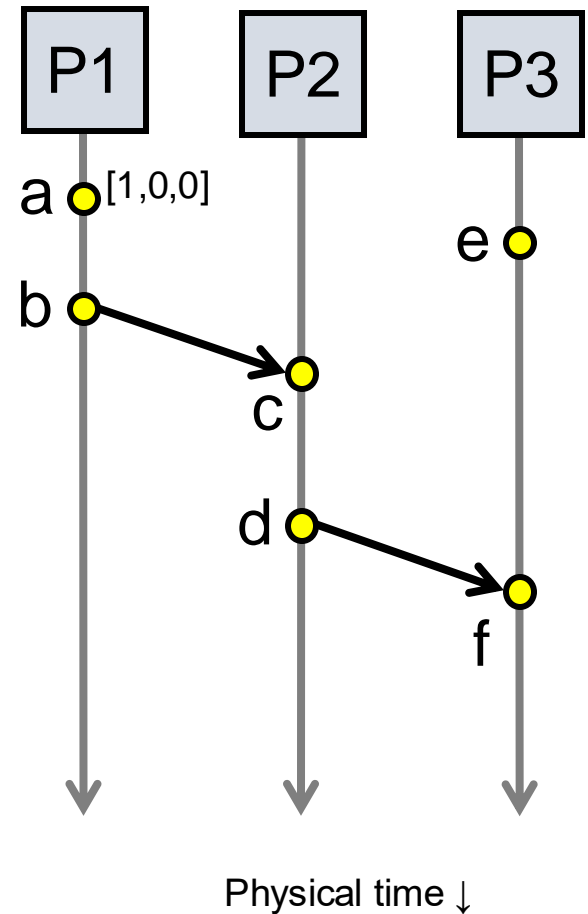
- All processes' VCs start at [0, 0, 0]

P1   P2   P3

a

e

b

c

d

f

Physical time ↓

Vector clock: Example

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

- Applying local update rule

P1   P2   P3

a

e

b

c

d

f

Physical time ↓

Vector clock: Example

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

- Applying local update rule

P1   P2   P3

a $\circ$ [1,0,0]

e $\circ$

b $\circ$

c $\circ$

d $\circ$

f $\circ$

Physical time ↓

Vector clock: Example

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

- Applying local update rule

P1   P2   P3

a [1,0,0]
   [2,0,0]   e
b
        c

     d
              f

Physical time ↓

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

P1   P2   P3

a [1,0,0]
              e
   [2,0,0]
b
              c

           d
                         f

Physical time ↓

Vector clock: Example

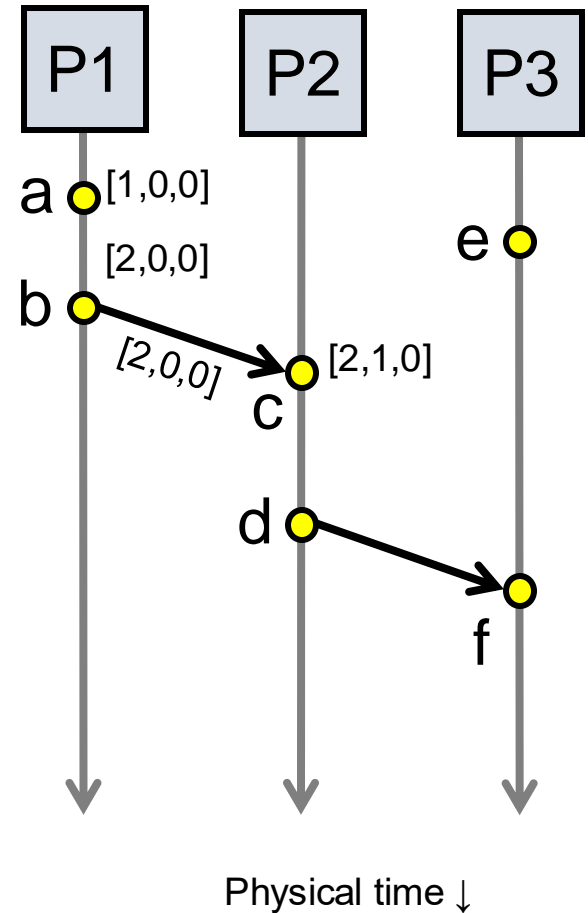- All processes' VCs start at [0, 0, 0]

- Applying local update rule

- Applying message rule
  - Local vector clock **piggybacks** on inter-process messages

P1   P2   P3

a [1,0,0]

e

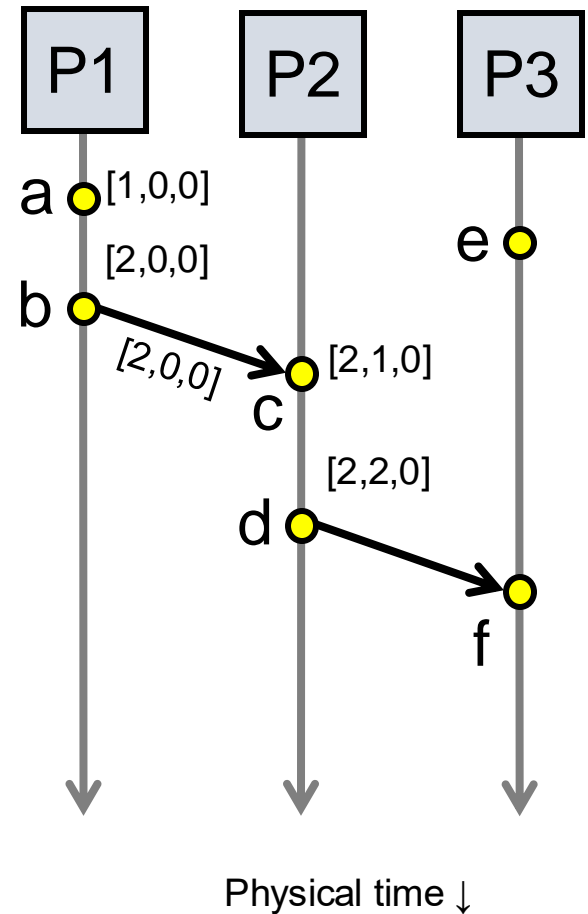[2,0,0]

b

[2,0,0]

c [2,1,0]

d

f

Physical time ↓

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

P1   P2   P3

a [1,0,0]

e

[2,0,0]

b

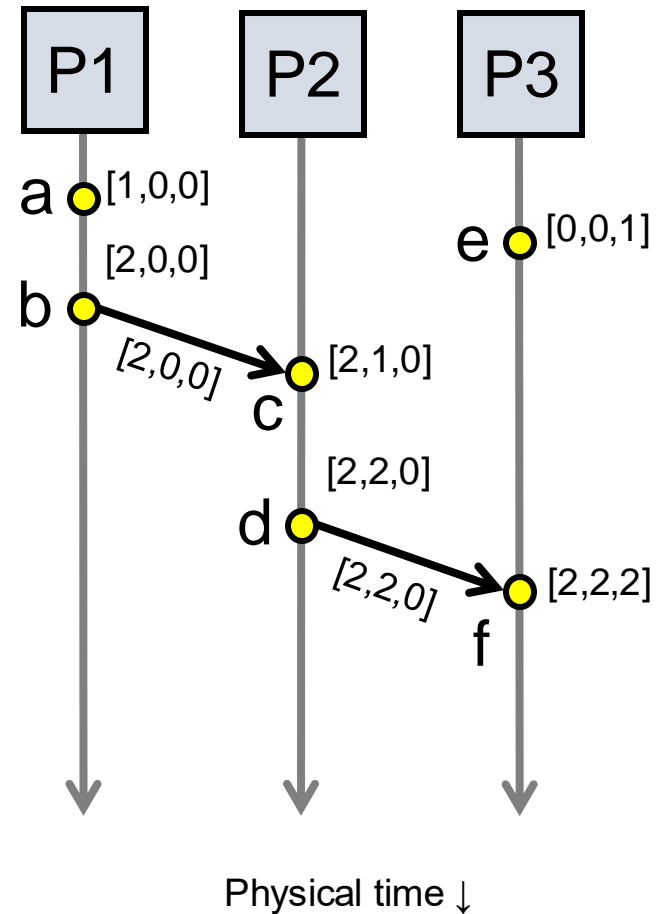[2,0,0]   c [2,1,0]

[2,2,0]

d

f

Physical time ↓

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

P1   P2   P3

a [1,0,0]

e [0,0,1]

[2,0,0]

b [2,0,0]

c [2,1,0]

[2,2,0]

d [2,2,0]

f [2,2,2]

Physical time ↓

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:

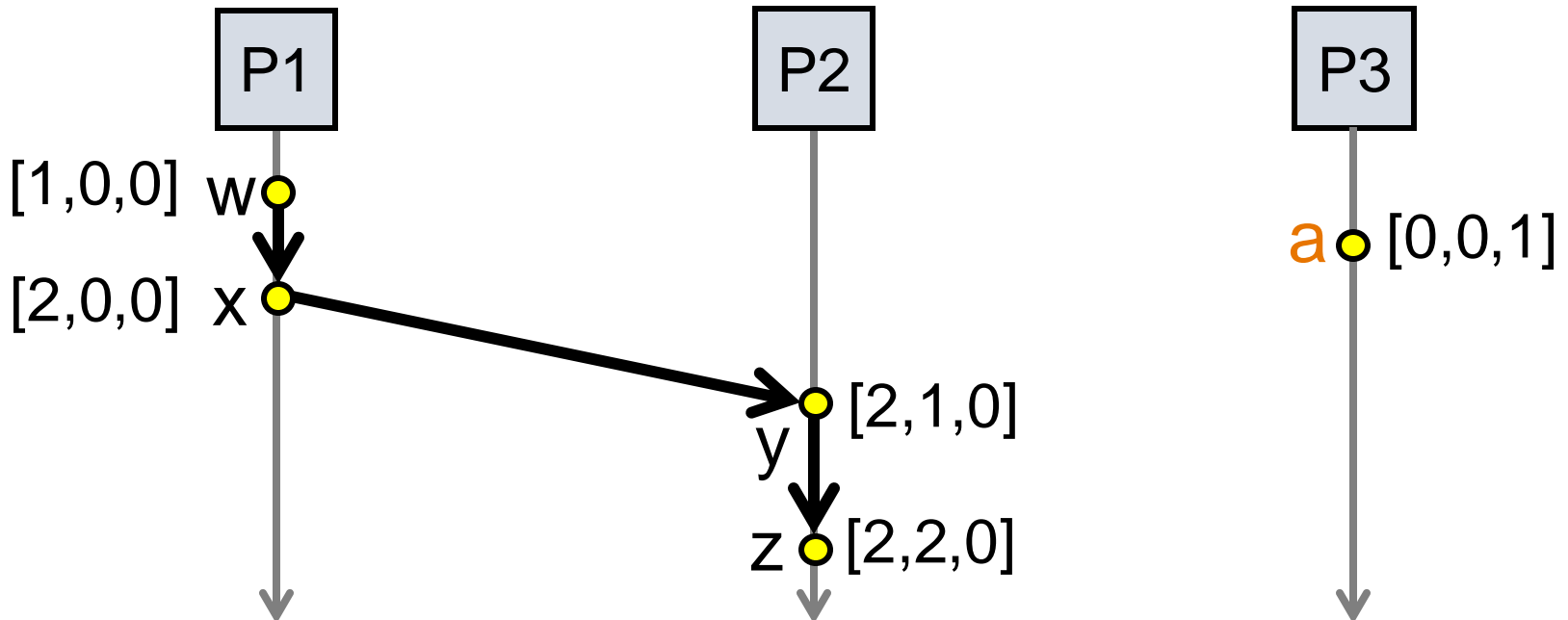   - $V(a) \parallel V(b)$ if $a_i < b_i$ and $a_j > b_j$, some i, j

Vector clocks capture causality

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

P1

P2

P3

[1,0,0] w

[2,0,0] x

y [2,1,0]

z [2,2,0]
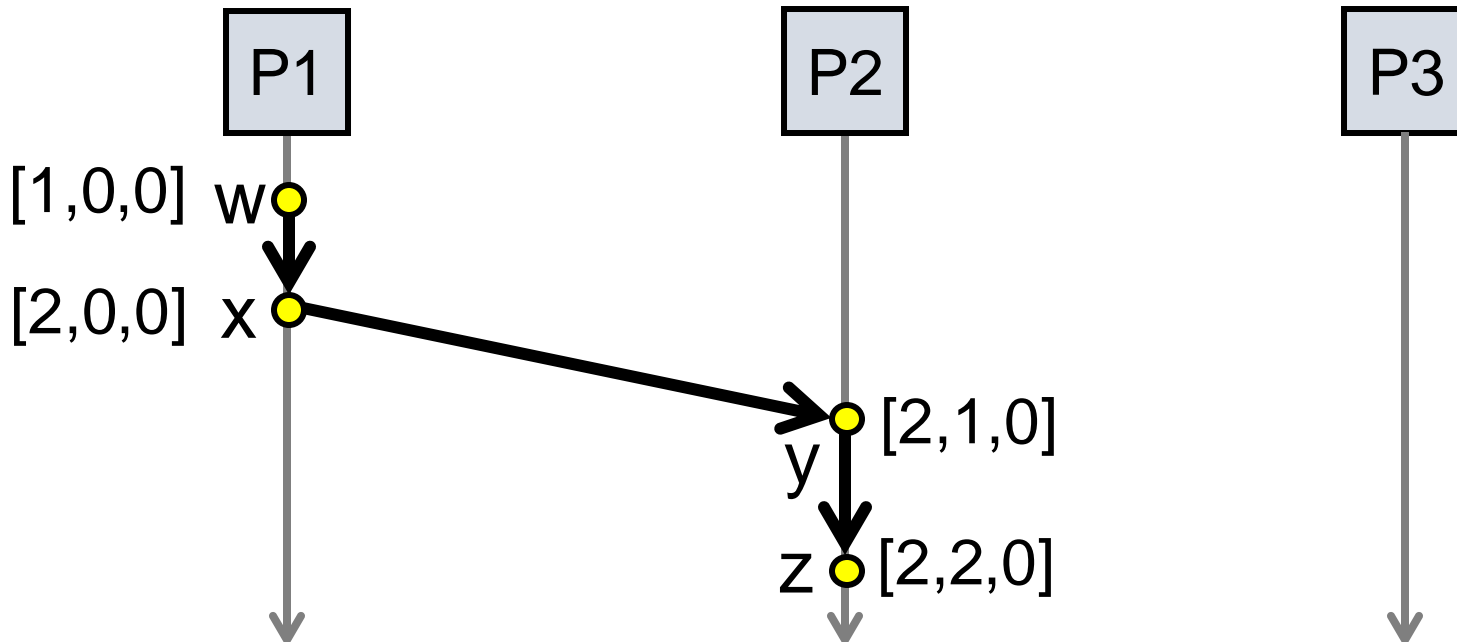
Vector clocks capture causality

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

- V(a) || V(w) then there is no such chain of events between a and w

P1   P2   P3

[1,0,0] w

[2,0,0] x

          [2,1,0]
          y

          [2,2,0]
          z

                        a [0,0,1]

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 -/-> a, i.e., either a $\rightarrow$ z or a || z

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

      Conclusion: a $\rightarrow$ z

Two events a, z

Lamport clocks: C(a) < C(z)
    Conclusion: z -/-> a, i.e., either a $\rightarrow$ z or a || z

Vector clocks: V(a) < V(z)
      Conclusion: a $\rightarrow$ z

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