CS 582: Distributed Systems

NTP and Logical Clocks

Dr. Zafar Ayyub Qazi

Fall 2024

Acknowledgements: Slides use some material from Martin Kleppmann (University of Cambridge), Indranil Gupta (UIUC), and Radhika Mittal (UIUC

Recap: Clock Synchronization

- Computers track physical time with a quartz clock
- Due to clock drift, clock error gradually increases
- Clock skew: different between two clocks at a point in time
- Solution: Periodically get the current time from a server that has a more time source (e.g., atomic clock)

Recap: How do we synchronize time?

- Synchronization in synchronous settings
  - $T_c = T_s + \frac{\min + max}{2}$
- Cristian's algorithm
  - $T_c = T_s + \frac{RTT}{2}$
- Berkeley algorithm
  - Use Cristian's algorithm to estimate time at each client
  - Average all local times
  - Send offsets

Today’s Agenda

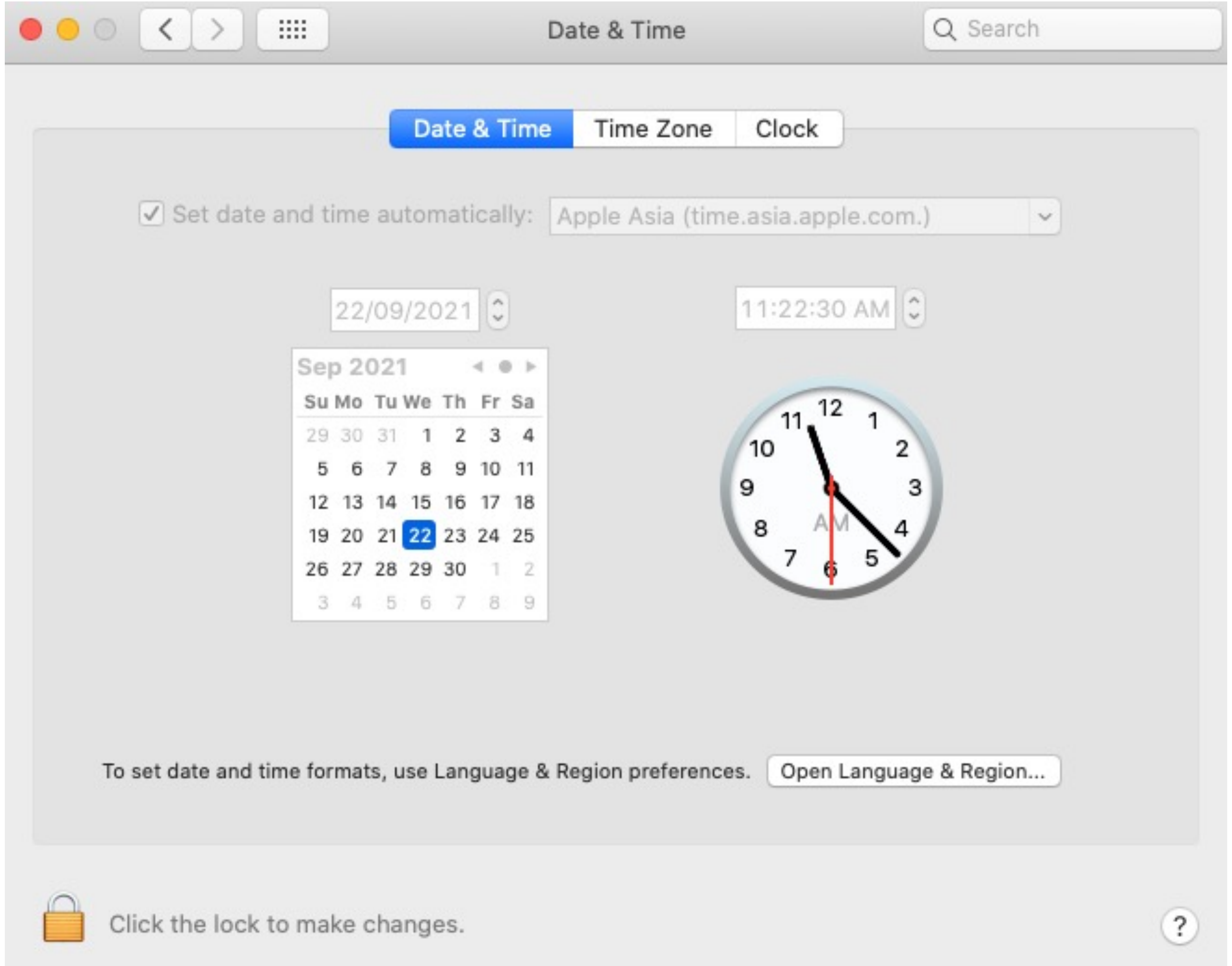
- Network Time Protocol (NTP)
- Logical Clock
  - Lamport Clock

Specific learning outcomes

By the end of today's lecture, you should be able to:

- ❑ Describe how NTP works
- ❑ Analyze when and why NTP time can cause correctness issues in a program
- ❑ Explain the *happens-before* relation
- ❑ Analyze why *happens-before* can only guarantee a partial order
- ❑ Explain what is a causal order
- ❑ Describe how the Lamport clock algorithm works
- ❑ Correctly apply Lamport clock timestamps to a given set of events in a distributed system
- ❑ Explain how the Lamport clock algorithm can be extended to implement a total order

Network Time Protocol (NTP)

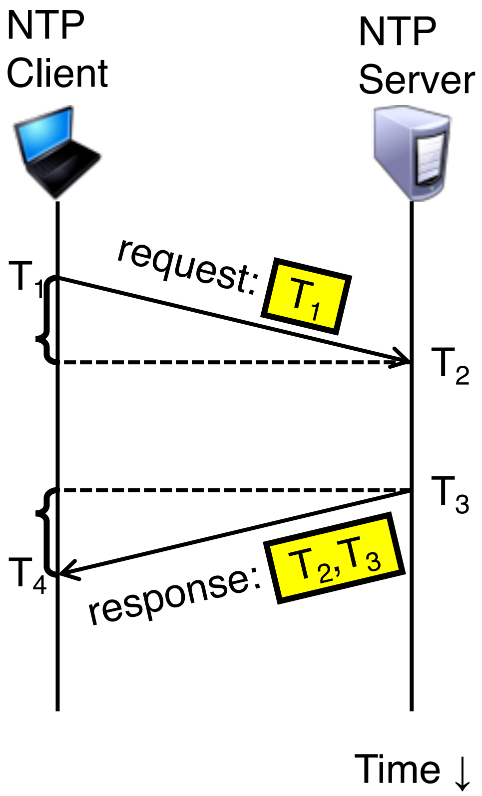


Network Time Protocol (NTP)

- Used by default in many operating systems
- Hierarchy of clock servers arranged into strata:
  - Stratum 0: atomic clock or GPS receiver
  - Stratum 1: synced directly with stratum 0 device
  - Stratum 2: servers that sync with stratum 1, etc
  - …
- May contact multiple servers, discard outliers, average rest
- Makes multiple requests to the same servers, uses min RTT to reduce errors due to network delay variations

Estimating time over the network

- Round-trip network delay, $\alpha = (T_4 - T_1) - (T_3 - T_2)$
- Offset = $\alpha/2$
- Estimated server time when client receives response = $T_3 + \alpha/2$
- Estimated clock skew = $\theta = T_3 + \alpha/2 - T_4 = (T_2 - T_1 + T_3 - T_4)/2$



NTP: Apply correction to clock

- Once the client has estimated clock θ, it needs to apply that correction to its clock
  - If μ < 125ms, slew the clock:
    - Slightly speed up or slow it down
      - Brings clocks in sync within ~5mins
  - If 125ms <= μ < 1000s, step the clock
    - Suddenly reset client clock to estimated server timestamp
  - If μ > 1000s, panic and do nothing
    - Leave the problem for a human operator to resolve

- Systems that rely on clock sync need to monitor clock skew

Time-of-day clocks and monotonic time

- Time of day clock
  - Time since a fixed date (e.g., 1 Jan 1970)
  - May suddenly move forwards or backwards (e.g., because of NTP step)
  - Timestamps can be compared across nodes (if synced)
  - Linux: clock_gettime(CLOCK_REALTIME)
- Monotonic clock
  - Time since an arbitrary point (e.g., when machine booted)
  - Always moves forward at a constant rate
  - Good for measuring elapsed time on a single node
  - Linux: clock_gettime (CLOCK_MONOTONIC)

Real synchronization is imperfect

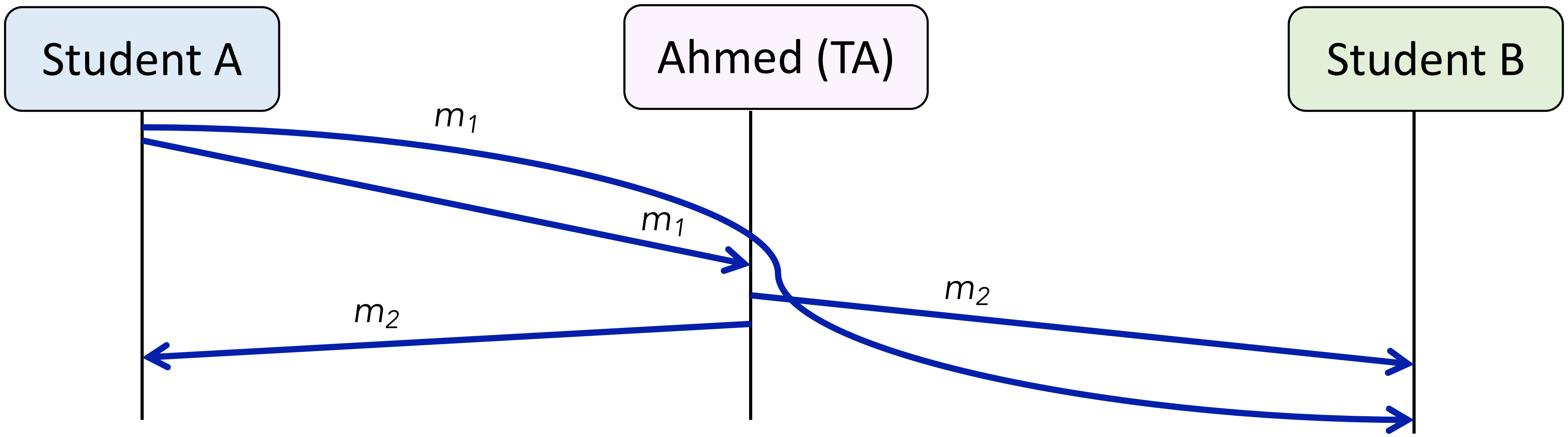
- Clocks may never be exactly synchronized

- Q: Can we always use real time stamps to order events across nodes in a distributed system?

Ordering of Events

- Let’s revisit our Slack example in which we wanted to order messages using timestamps

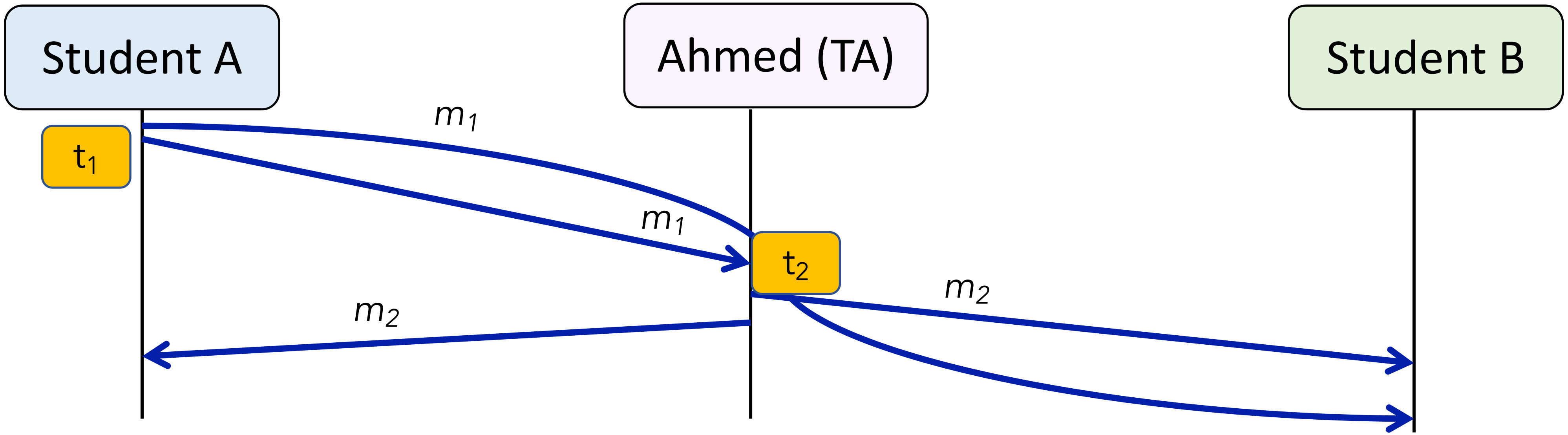
Example: Slack like Application



m_1: "Do we have a quiz on Mon?"
m_2: "No"

Student B sees m_2 first, m_1 second, even though logically m_1 happened before m_2

Ordering of messages using timestamps?



m_1: (t_1, "Do we have a quiz on Mon?")
m_2: (t_2, "No")

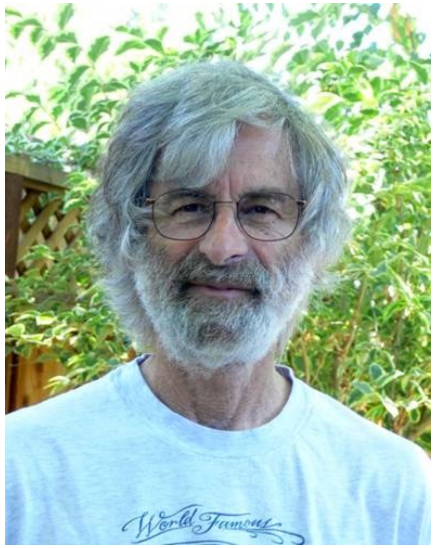
Problem: If the clocks are not synchronized, it is possible $t_2 < t_1$, which will result in an ordering that is inconsistent with the expected ordering

Driving question of the lecture

- Can we avoid synchronizing physical time altogether and order events in a distributed system?

Idea: Logical clocks

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



Idea: Disregard the physical clock time

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

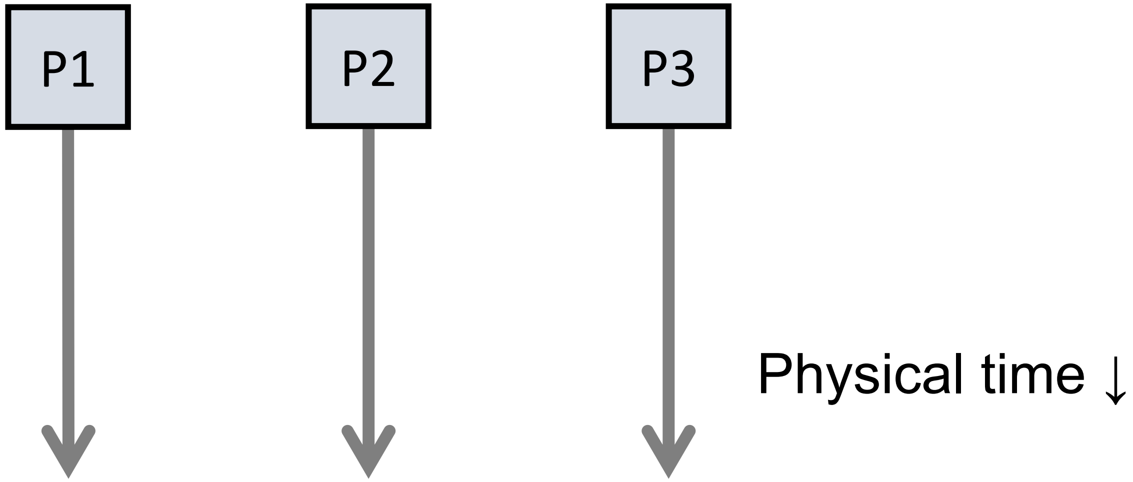
* **"Time, Clocks and the Ordering of Events in Distributed Systems"** *by Lamport*
* *Lamport was awarded the* **Turing award in 2013** *for his contributions in distributed systems*

Defining "happens-before" relation (→)

- An event is something happening at one node (sending or receiving a message, or a local execution step)
- If event a happens before event b, we write it as $\boldsymbol{a \rightarrow b}$

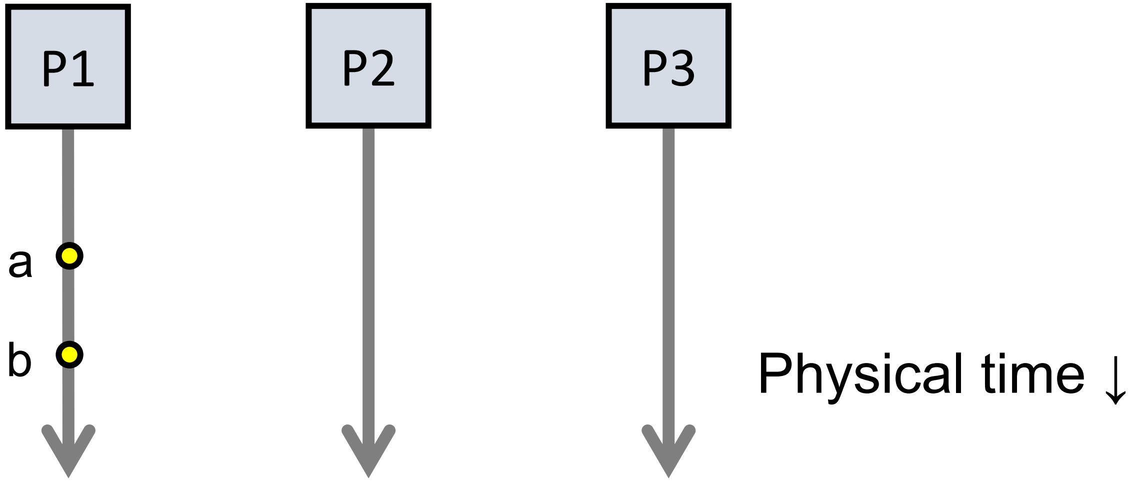
Defining "happens-before" relation (→)

- Consider three processes (on different nodes): P1, P2, and P3



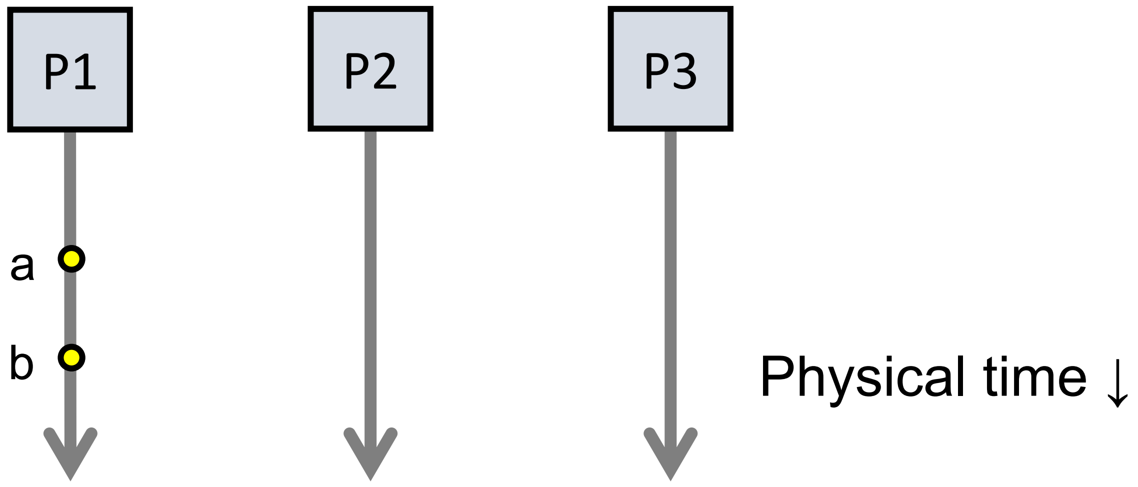
Defining "happens-before" relation (→)

1. Can observe event order at a single process



Defining "happens-before" relation (→)

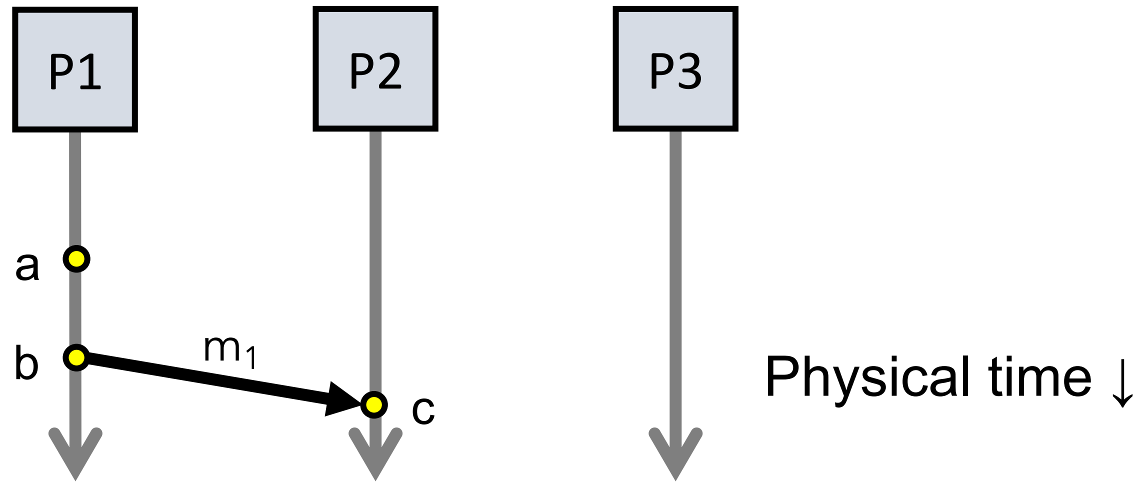
1. If same process and $\boldsymbol{a}$ occurs before $\boldsymbol{b}$, then $\boldsymbol{a} \rightarrow \boldsymbol{b}$



Defining "happens-before" relation (→)

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

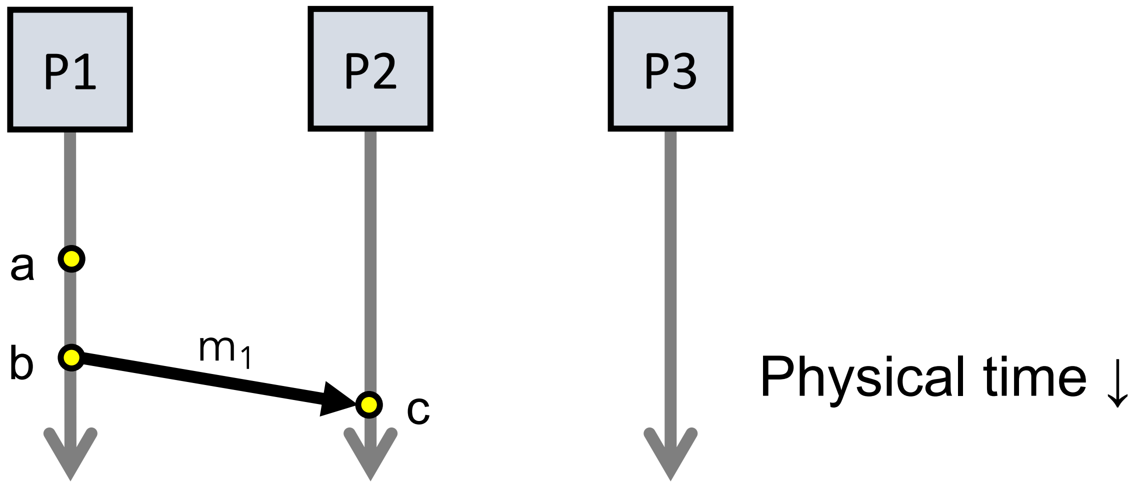
2. Can observe ordering when processes communicate



Defining "happens-before" relation (→)

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

2. If $\boldsymbol{c}$ is a message receipt of $\boldsymbol{b}$, then $\boldsymbol{b} \rightarrow \boldsymbol{c}$

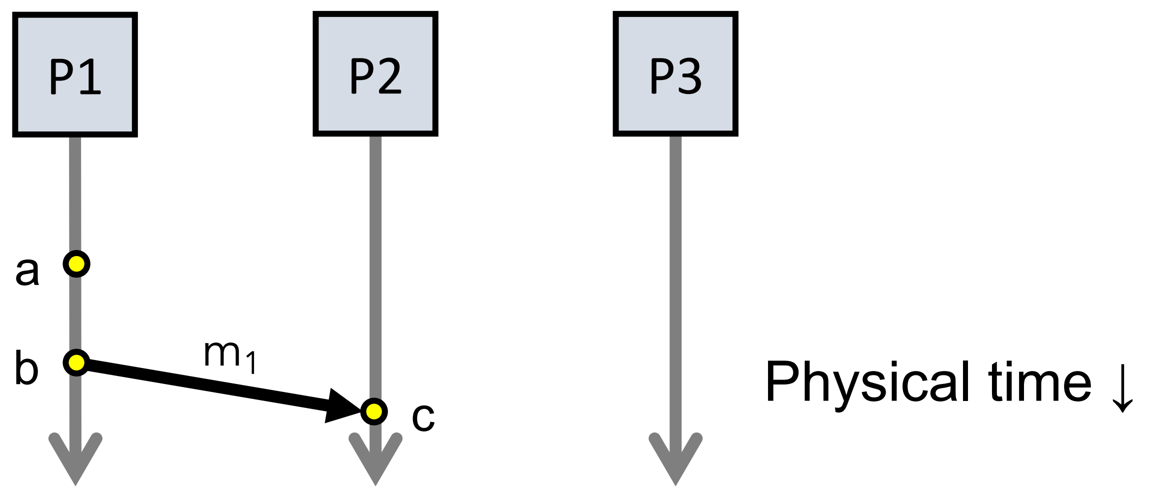


Defining "happens-before" relation (→)

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

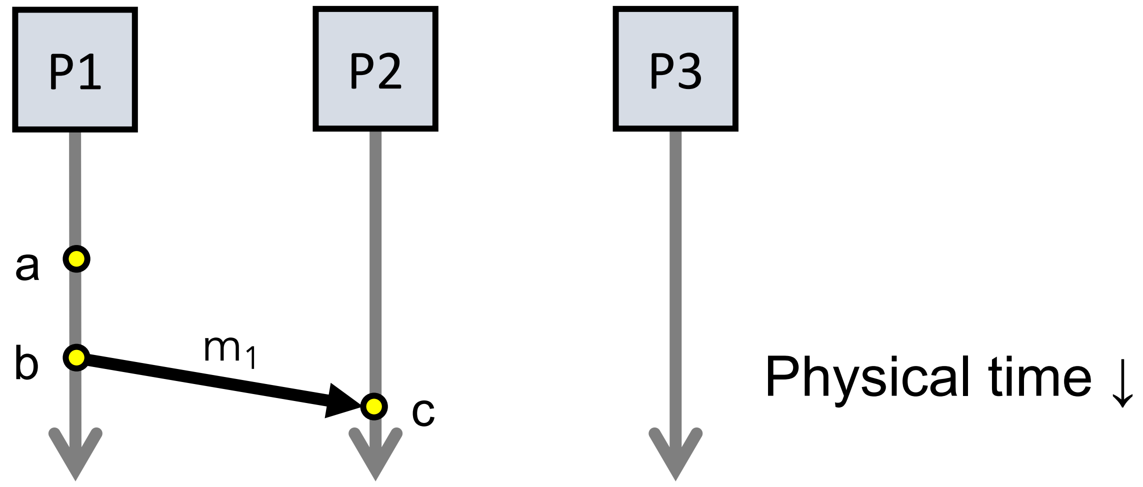


Defining "happens-before" relation (→)

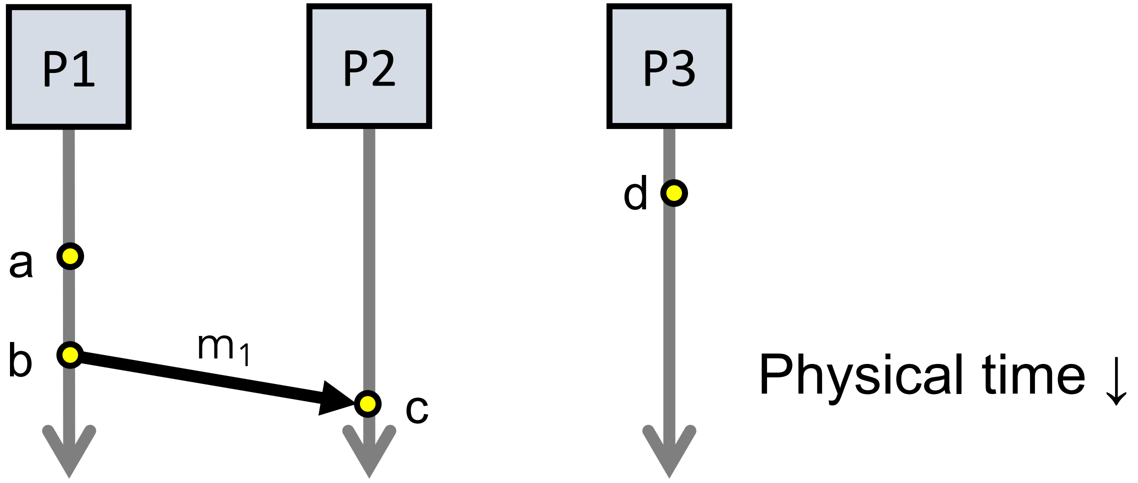
1. If same process and $\boldsymbol{a}$ occurs before $\boldsymbol{b}$, then $\boldsymbol{a} \rightarrow \boldsymbol{b}$

2. If $\boldsymbol{c}$ is a message receipt of $\boldsymbol{b}$, then $\boldsymbol{b} \rightarrow \boldsymbol{c}$

3. If $\boldsymbol{a} \rightarrow \boldsymbol{b}$ and $\boldsymbol{b} \rightarrow \boldsymbol{c}$, then $\boldsymbol{a} \rightarrow \boldsymbol{c}$



How are events a and d related?



Are all events related by → ?

Summary: "happens-before" relation (→)

- We say a happens before event b (written $a \rightarrow b$) iff:
  - a and b occurred at the same process, and a occurred before b in that process's local order, or
  - Event a is the sending of some message m, and event b is the receipt of the same message m, or
  - There exists an event c such that $a \rightarrow c$ and $c \rightarrow b$
    - Transitivity applies

- The happens-before relation is a <u>partial order</u>
  - It is possible that neither $a \rightarrow b$ nor $b \rightarrow a$
  - In that case, a and b are <u>**concurrent**</u> (written as $a \,||\, b$)

Total Order

Potential Causality

- When $a \rightarrow b$, then a might have caused b
- When $a \,||\, b$, we know that a could not have caused b
- Happens-before relation encodes potential causality

- Let be $\prec$ a strict total order on events
  - If $a \rightarrow b \;\; => \;\; (a \;\prec\; b)$ then $\prec$ is a causal order
    - Or $\prec$ is "consistent with causality"

Questions

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 $C(a) < C(b)$

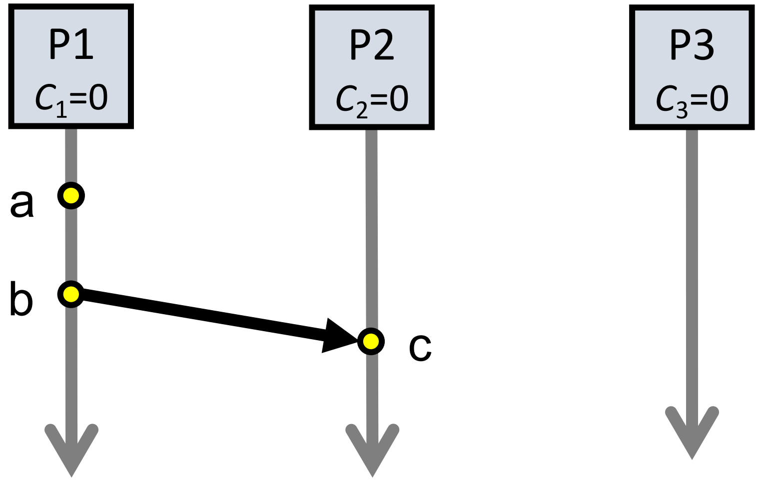
Lamport Clock Algorithm

- Each process maintain an event counter
  - We refer to this event counter as the process's Lamport/local clock

Lamport Clock Algorithm

- Each process P_i maintains a local clock C_i

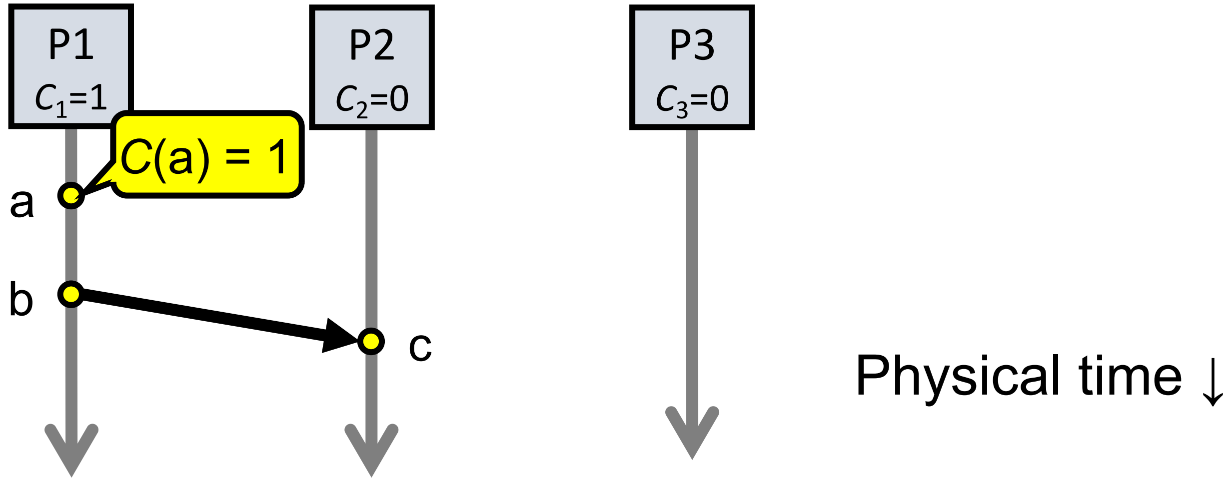
1. Before executing an event, $C_i \leftarrow Ci \;_{+\;1}$



Physical time ↓

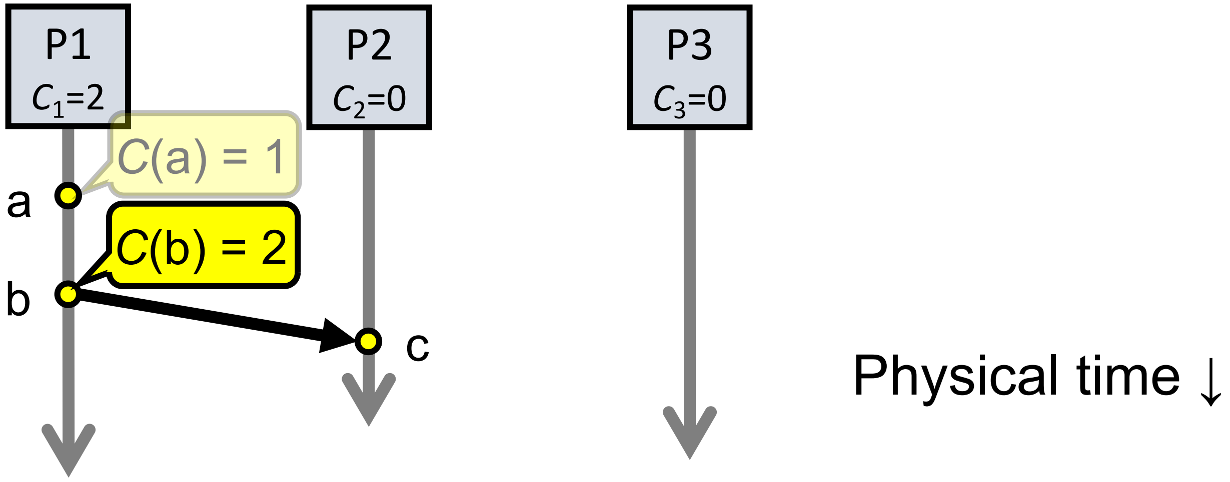
Lamport Clock Algorithm

1. Before executing an event a, $C_i \leftarrow C_i + 1$:
   - Set event time $C(a) \leftarrow C_i$



Lamport Clock Algorithm

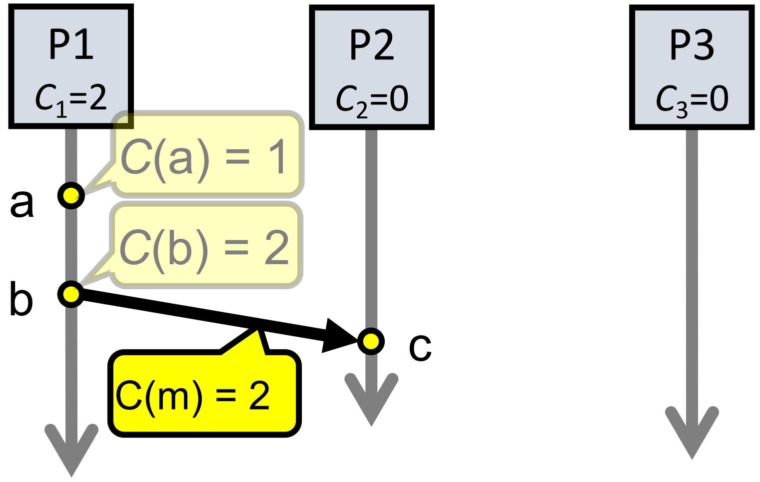
1. Before executing an event b, $C_i \leftarrow Ci + 1$:
   - Set event time $C(b) \leftarrow C_{\mathrm{i}}$



Lamport Clock Algorithm

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

2. Send the local clock in the message m

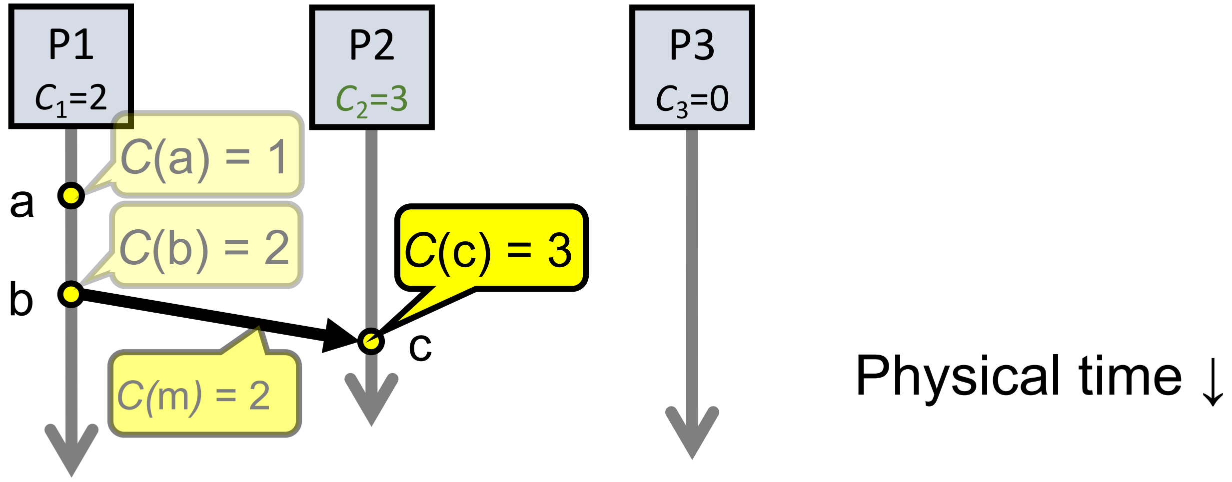


Physical time ↓

Lamport Clock Algorithm

3. On process P_j receiving a message m:

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



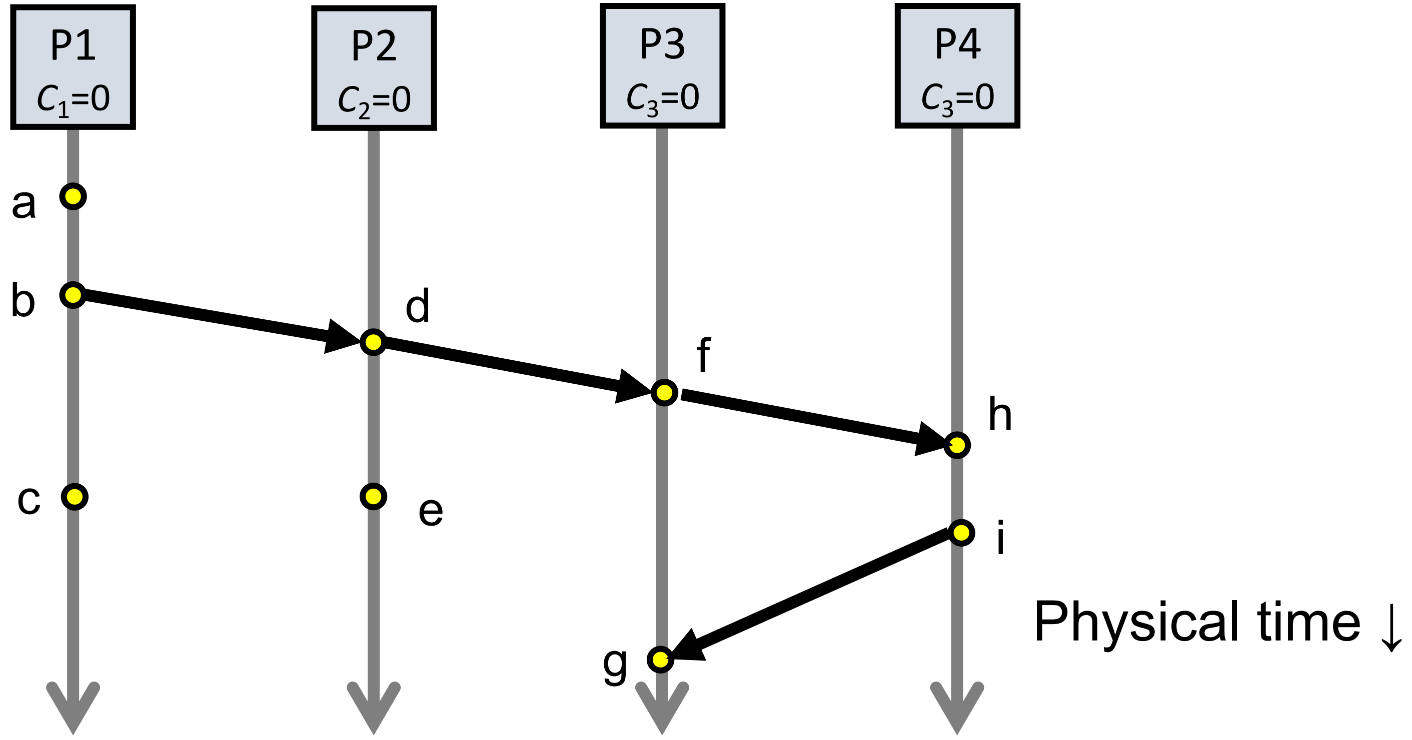
Summary: Lamport Clock Algorithm

- Each process maintains an event counter
- Before executing an event, increment the counter
- Whenever a process sends a message, include the counter
- When a message is received, set the counter to:
    - max(local_counter, received_counter) + 1

Lamport Clocks and Total Order

- How can we extend Lamport Clocks to guarantee total ordering of events?

Class Exercise: Order all these events



Summary: Happens-before and Lamport Clock

- Happens-before relation (a→ b)
  1. Local update: If same process and a occurs before b, then $\boldsymbol{a} \rightarrow \boldsymbol{b}$
  2. Message communication: If $\boldsymbol{b}$ is a message receipt of $\boldsymbol{a}$, then $\boldsymbol{a} \rightarrow \boldsymbol{b}$
  3. Transitivity: If $\boldsymbol{a} \rightarrow \boldsymbol{c}$ and $\boldsymbol{c} \rightarrow \boldsymbol{b}$, then $\boldsymbol{a} \rightarrow \boldsymbol{b}$

- Lamport Clock algorithm
  - Each process maintains an event counter
  - Before executing an event, increment the counter
  - Whenever a process sends a message, include the counter value
  - When a message is received, set the counter to:
    - $\max(local\_counter, received\_counter) + 1$

Take-away points: Lamport Clocks

- Can totally order events in a distributed system: that's useful!
- But: while by construction, a → b implies C(a) < C(b),
  - The converse is not necessarily true:
    - C(a) < C(b) does not imply a → b (possibly, a || b)

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

Next Lecture: Inferring potential causality

- Given two timestamps C(a) and C(z), want to know whether there's a chain of events linking them:

a → b → … → y → z

Next Lecture

- Application of Lamport clocks
  - Totally-ordered multicast for multi-site database replication
- Vector Clocks
  - Can be used to infer potential causal relationships from timestamps