Introduction to Concurrency & Logical Time

COS 316: Principles of Computer System Design

Lecture 15

Amit Levy & Ravi Netravali

Concurrency

- Multiple things happening at the same time

- Primary benefit is better performance
  - Do more work in the same amount of time
  - Complete fixed amount work in less time
  - Better utilize resources

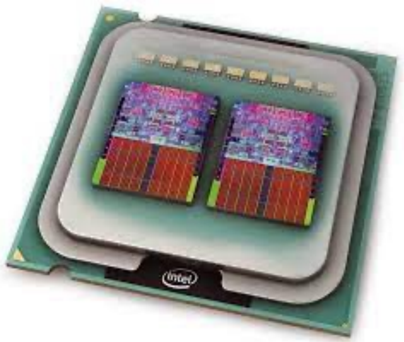
- Primary cost is complexity
  - Hard to reason about
  - Hard to get right
  - (Systems deal with it, not applications, … to some extent)

Concurrency Examples

Run a computation on all cores in a machine

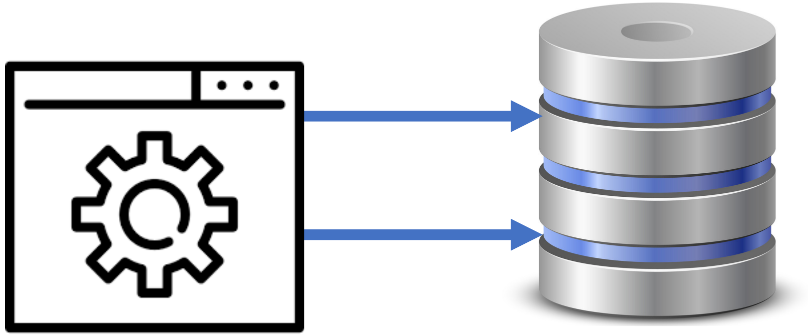
Run a computation on all cores on 100K machines!

Concurrency Examples

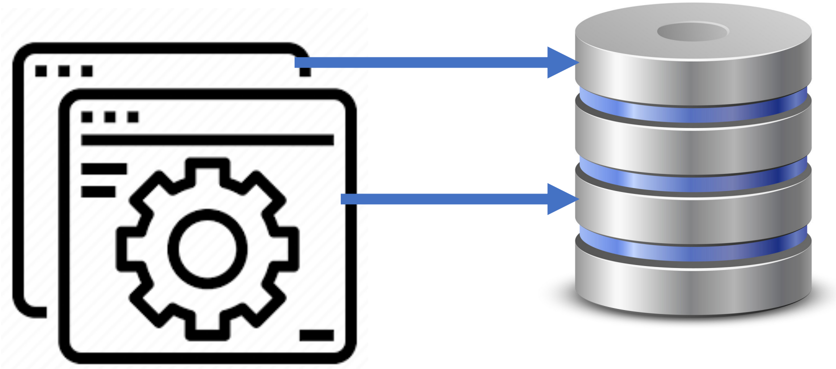


Run a computation on all cores in a machine

Run a computation on all cores on 100K machines!

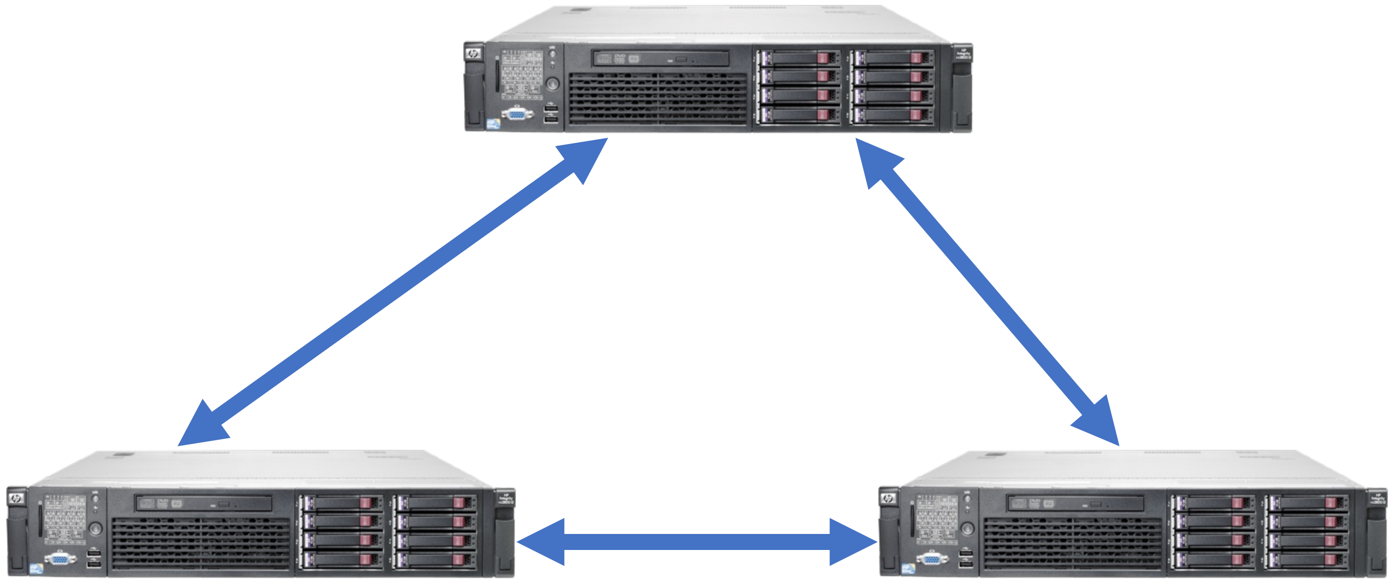


Application with multiple outstanding writes to disk



Many applications write to the disk

Distributed Systems



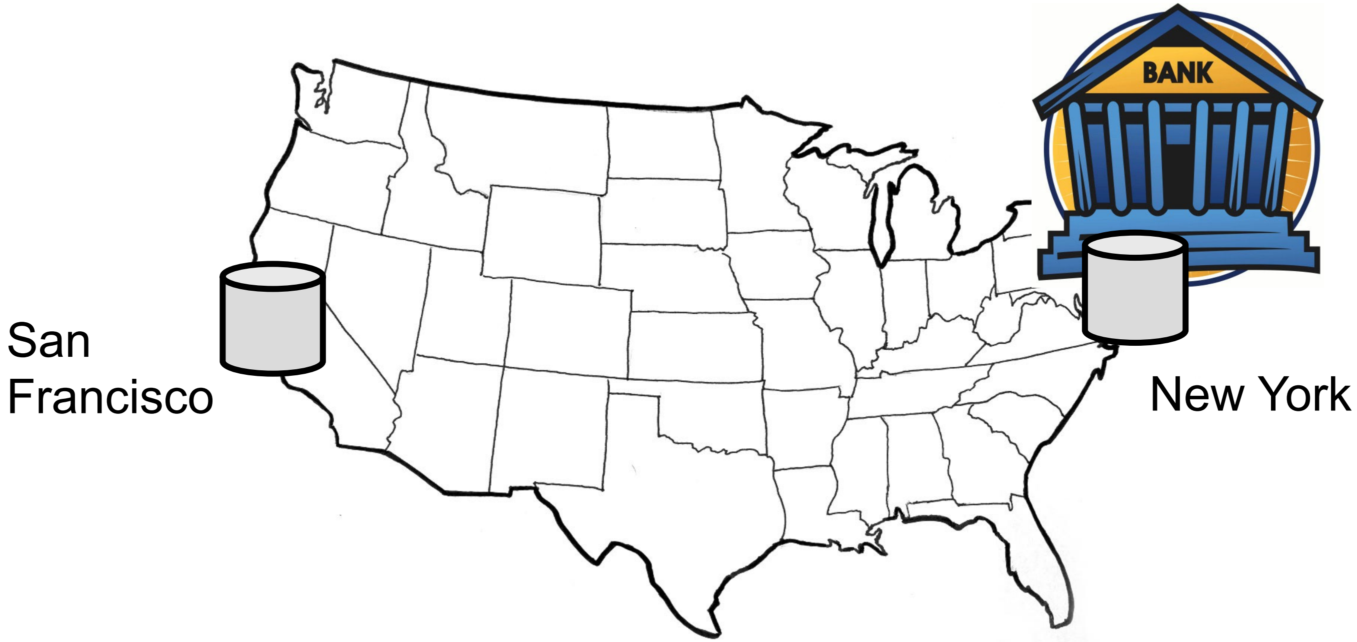
1) Multiple computers
2) Connected by a network
3) Doing something together

Concurrency is Inevitable!

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 query to the nearest copy
  - Client sends update to both copies

The consequences of concurrent updates

- Replicate the database, keep one copy in sf, one in nyc
  - Client sends query to the nearest copy
  - Client sends update to both copies

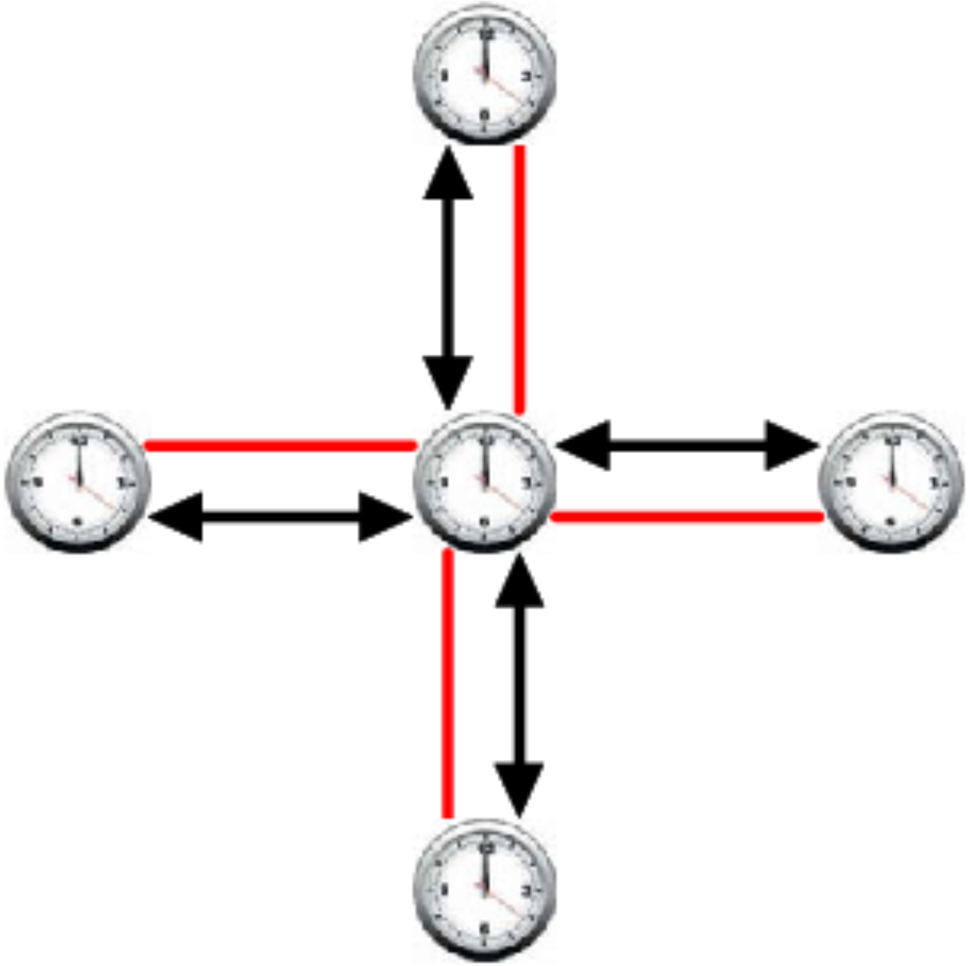
What to Do?

- Handle one client at a time?
  - Deposit $100 in both replicas, then pay 1% interest in both replicas
  - Pay 1% interest in both replicas, then deposit $100 in both replicas

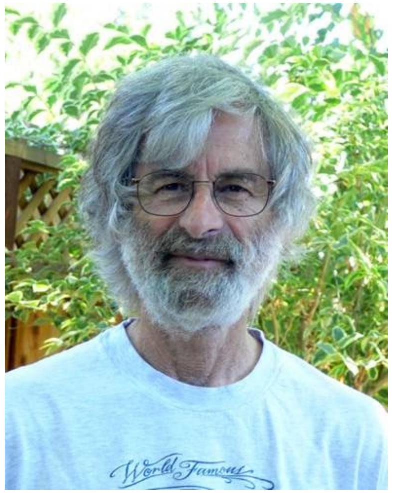
- How to enforce a consistent order?
  - Based on wall-clock time?

RFC 677 “The Maintenance of Duplicate Databases” (1975)

“To the extent that the communication paths can be made reliable, and the clocks used by the processes kept close to synchrony, the probability of seemingly strange behavior can be made very small. However, *the distributed nature of the system dictates that this probability can never be zero*.”



Idea: Logical clocks



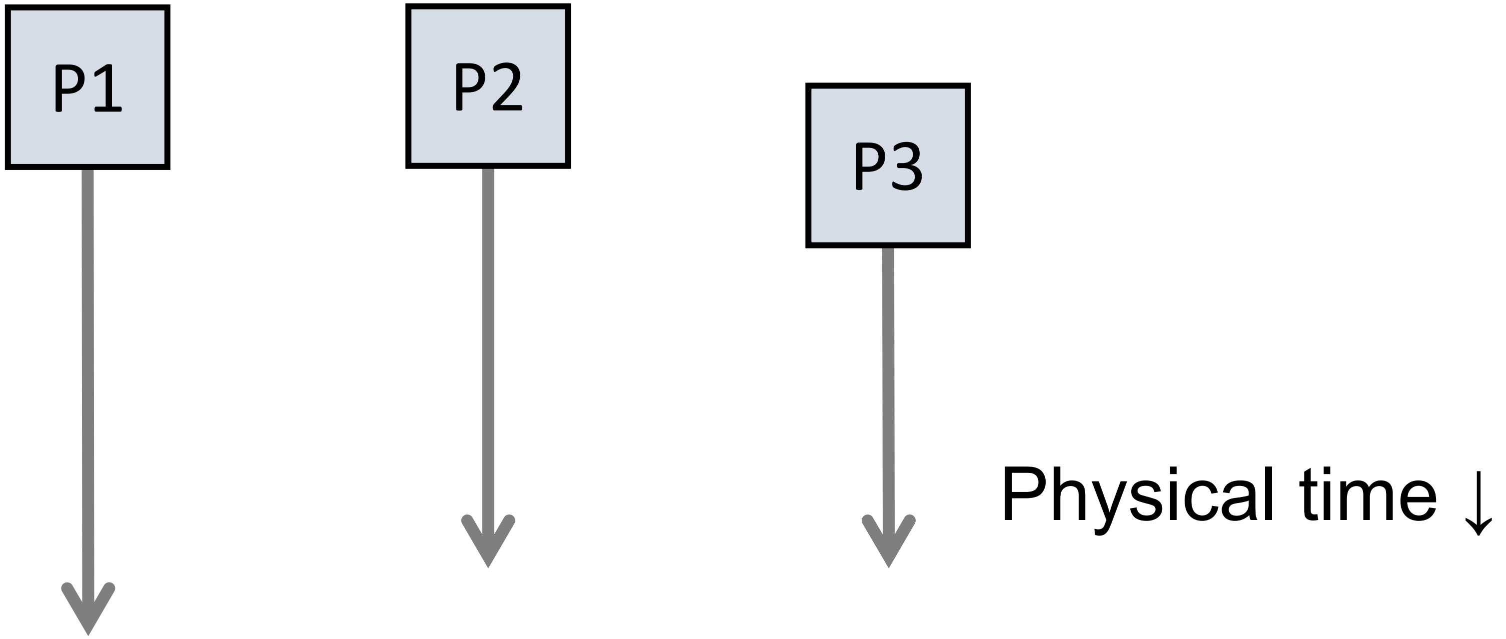
- Landmark 1978 paper by Leslie Lamport
- Insight: only the events themselves matter

Idea: Disregard the precise clock time

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

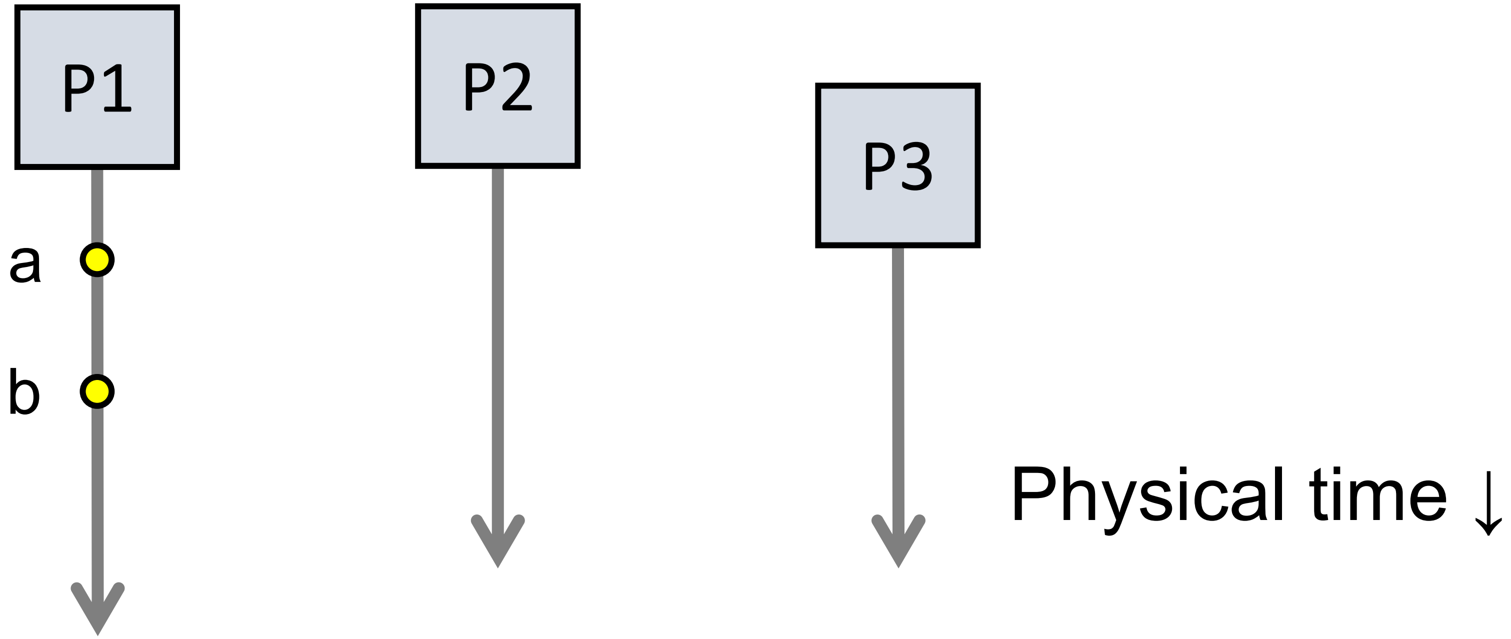
Defining "happens-before" (→)

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



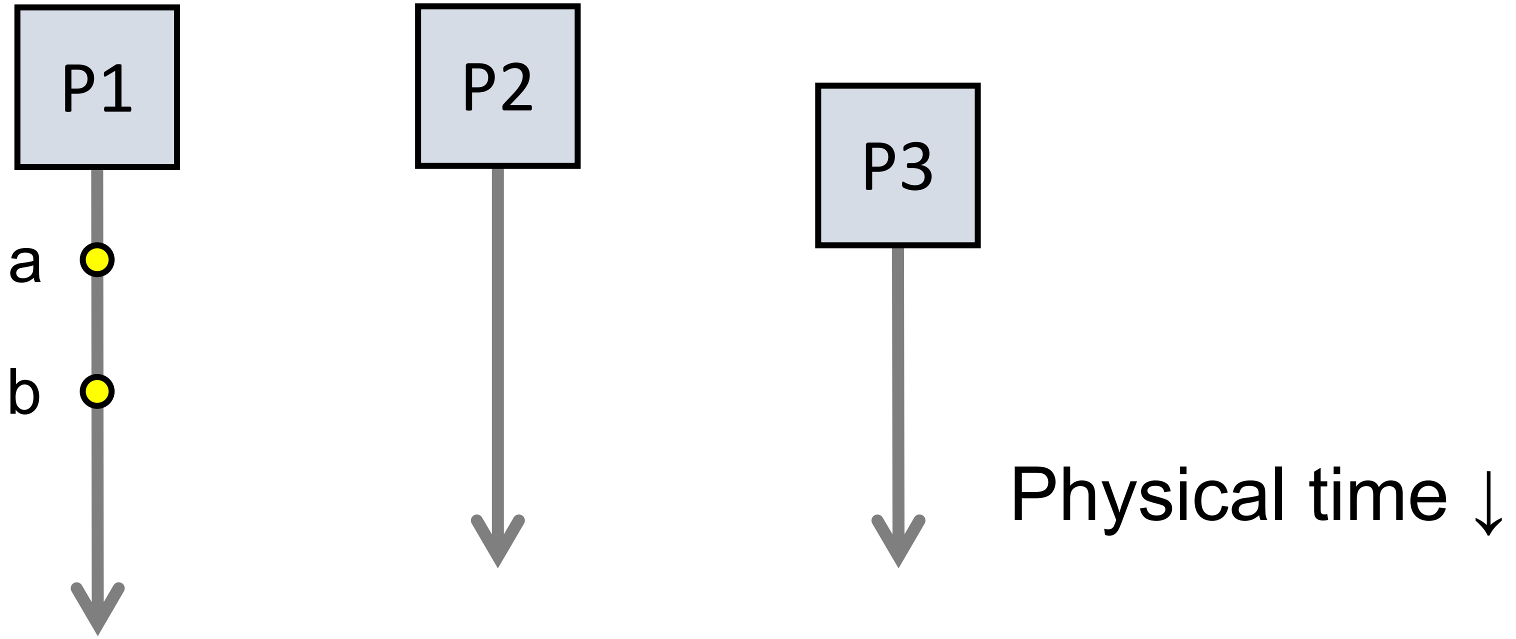
Defining "happens-before" (→)

- Can observe event order at a single process



Defining "happens-before" (→)

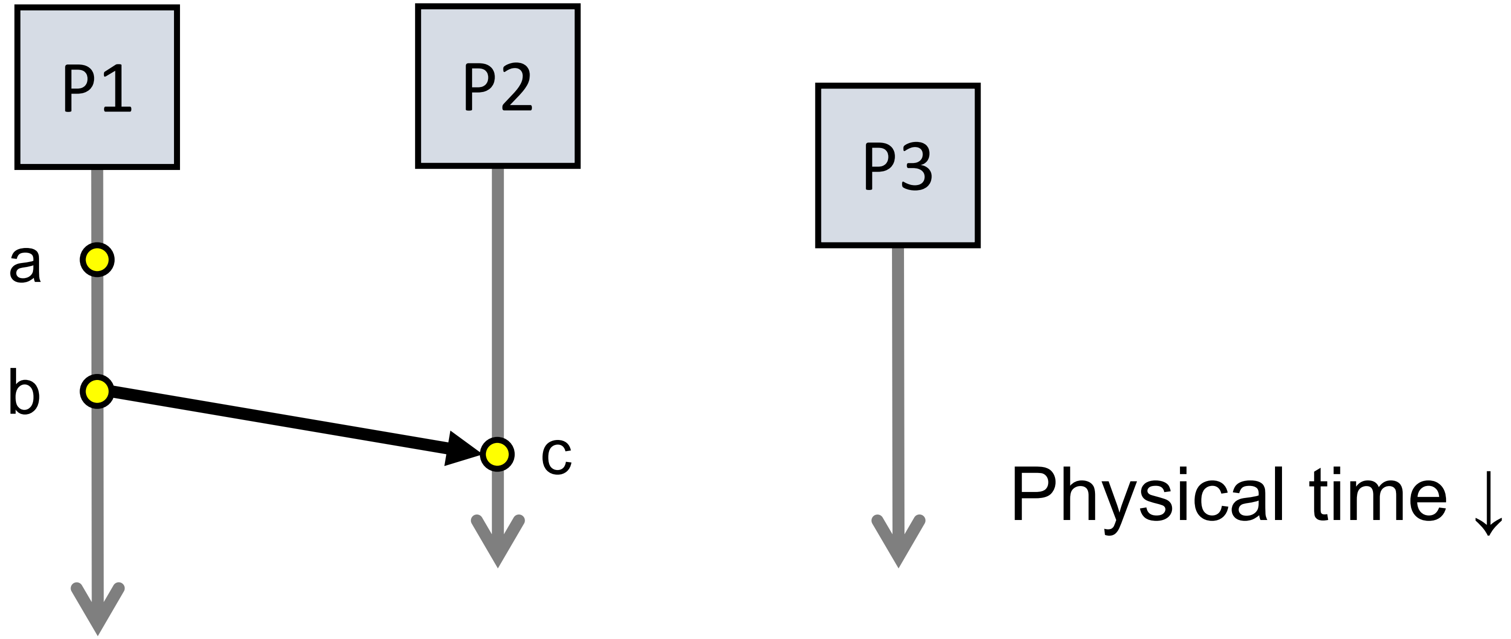
1. If **same process** and **a** occurs before **b**, then **a → b**



Defining "happens-before" (→)

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

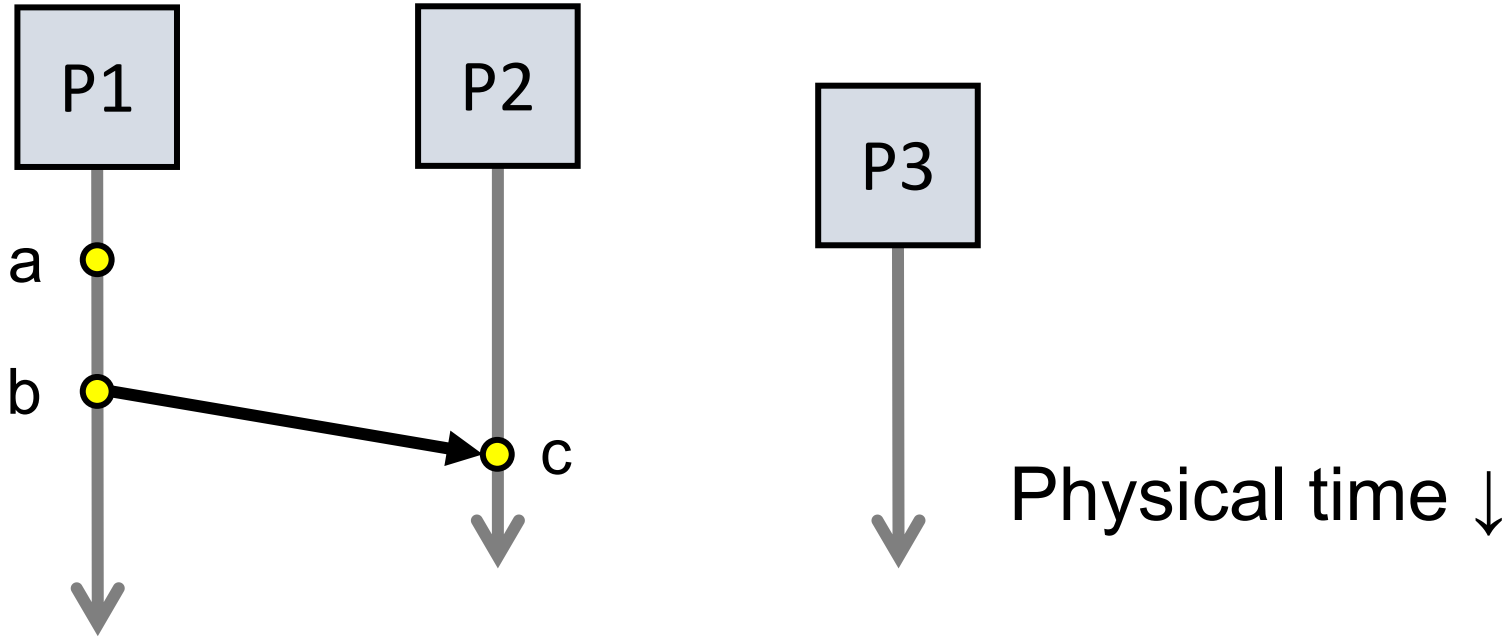
2. Can observe ordering when processes communicate



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**

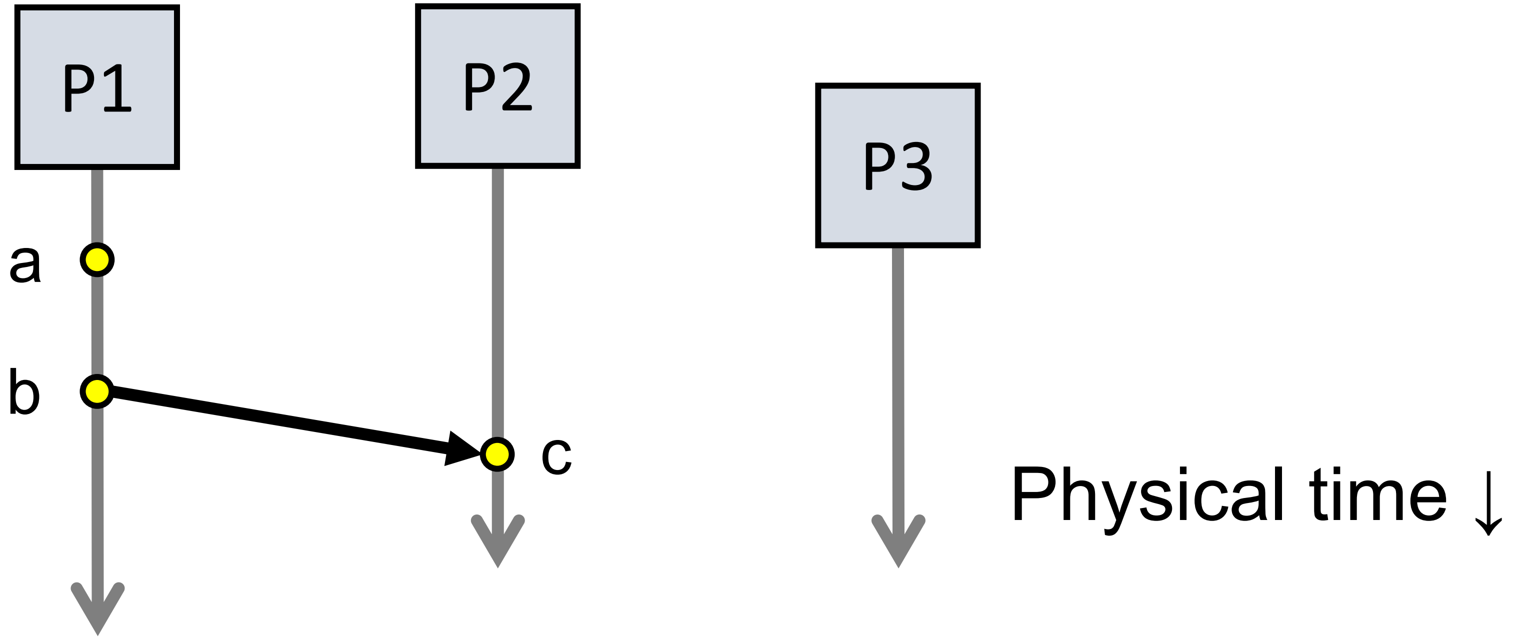


Defining "happens-before" (→)

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

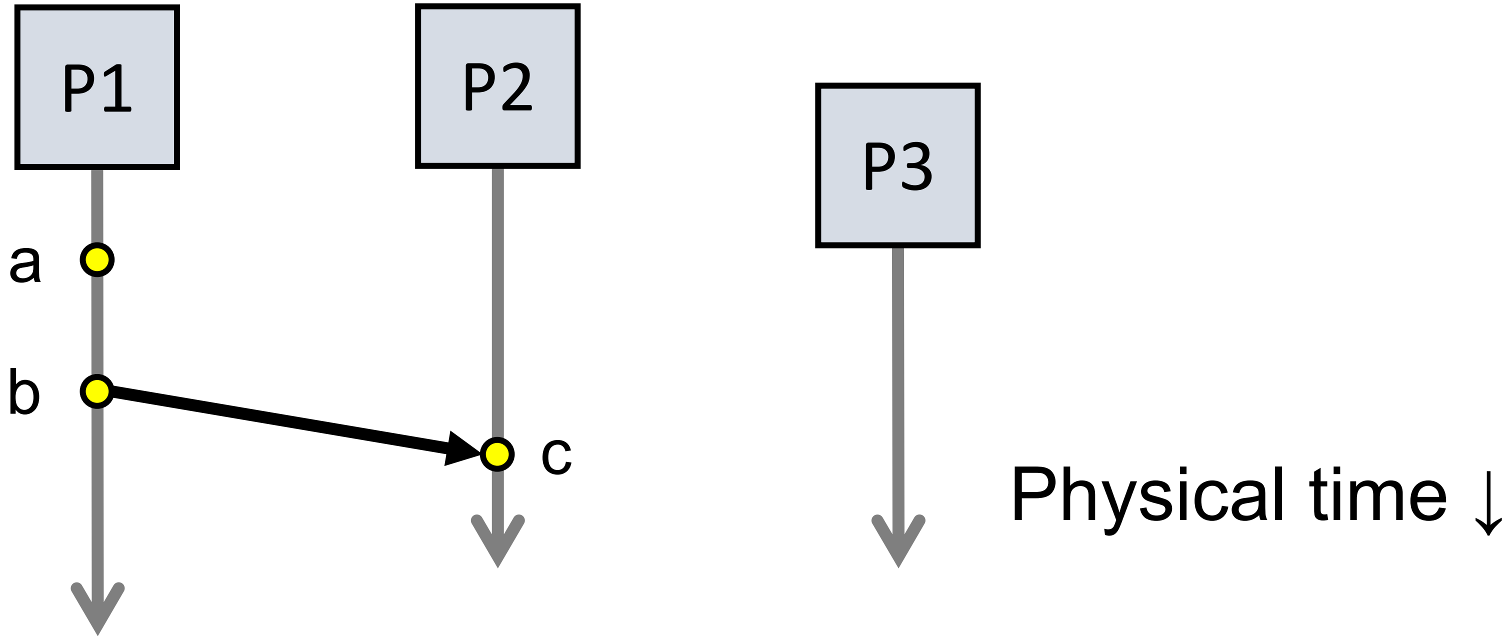


Defining "happens-before" (→)

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

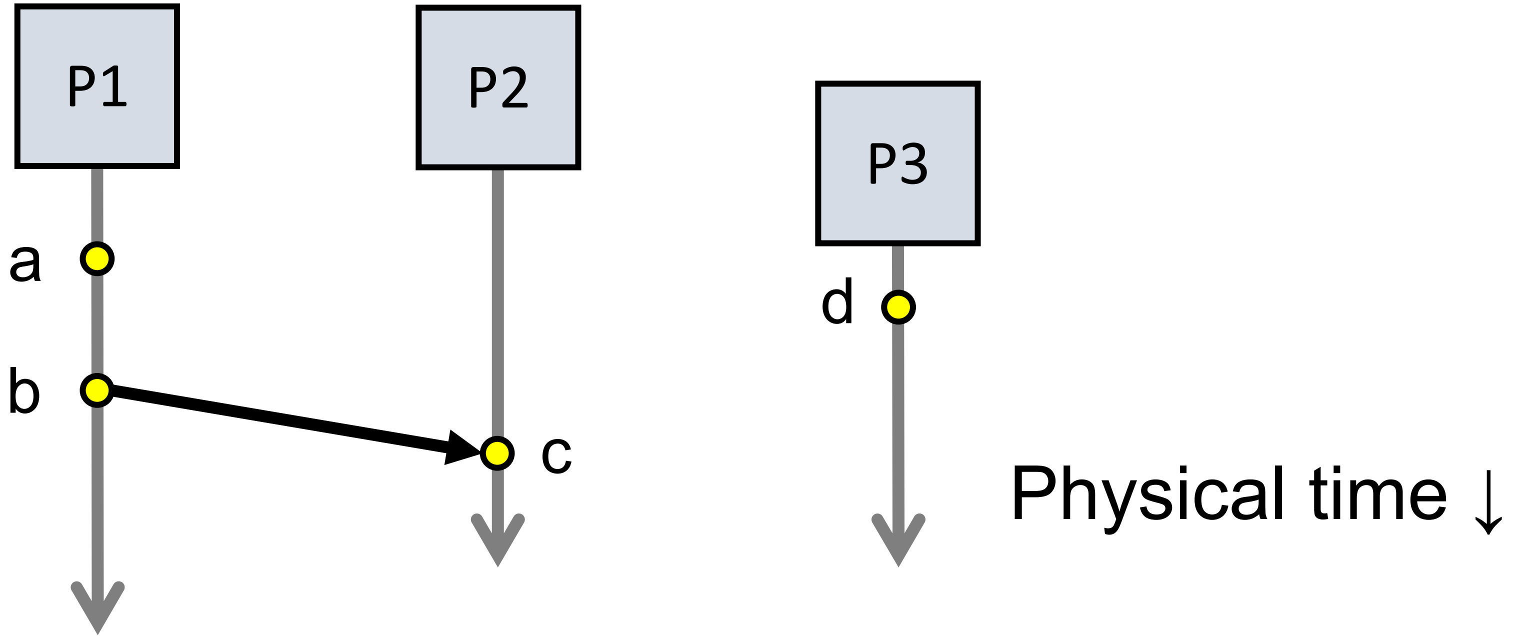
2. If **c** is a message receipt of **b**, then **b** → **c**

3. If **a** → **b** and **b** → **c**, then **a** → **c**



Concurrent events

- Not all events are related by →
- a, d not related by → so concurrent, written as a || d



Lamport clocks: Objective

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

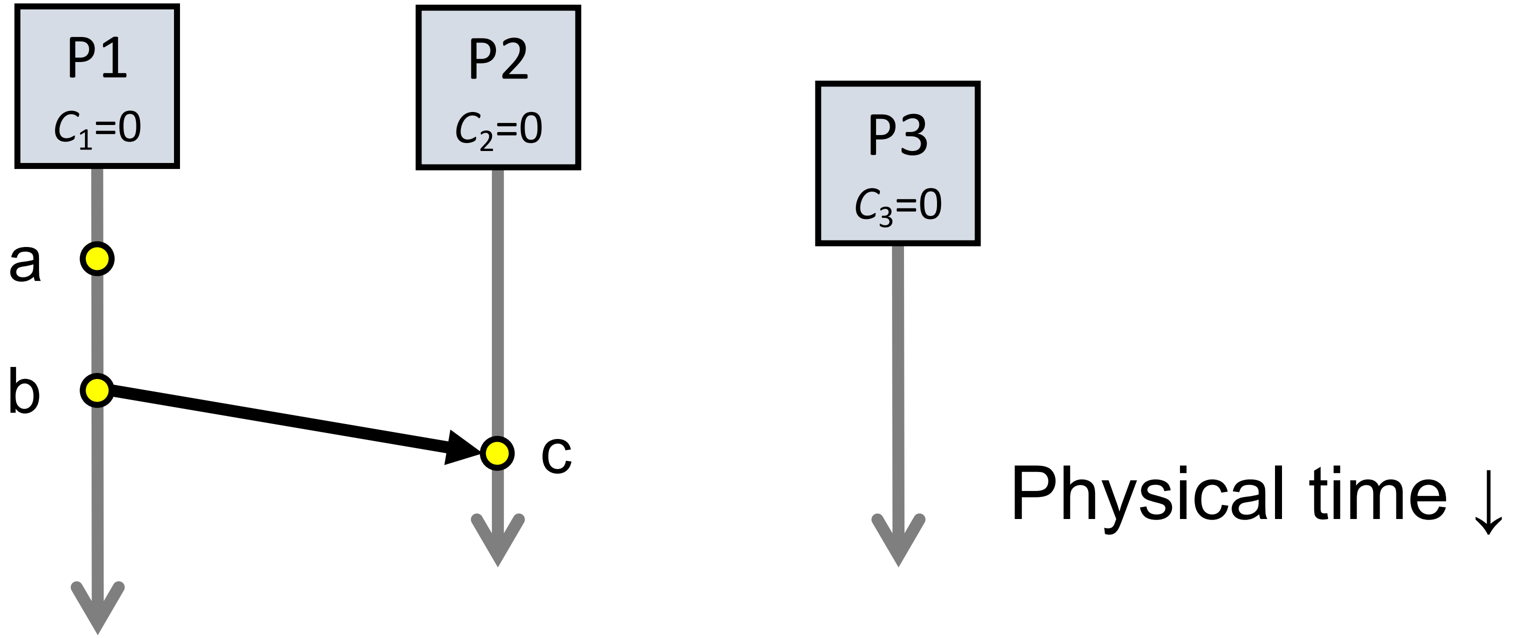
Plan: Tag events with clock times; use clock times to make distributed system correct

- Clock condition: If a → b, then C(a) < C(b)

The Lamport Clock algorithm

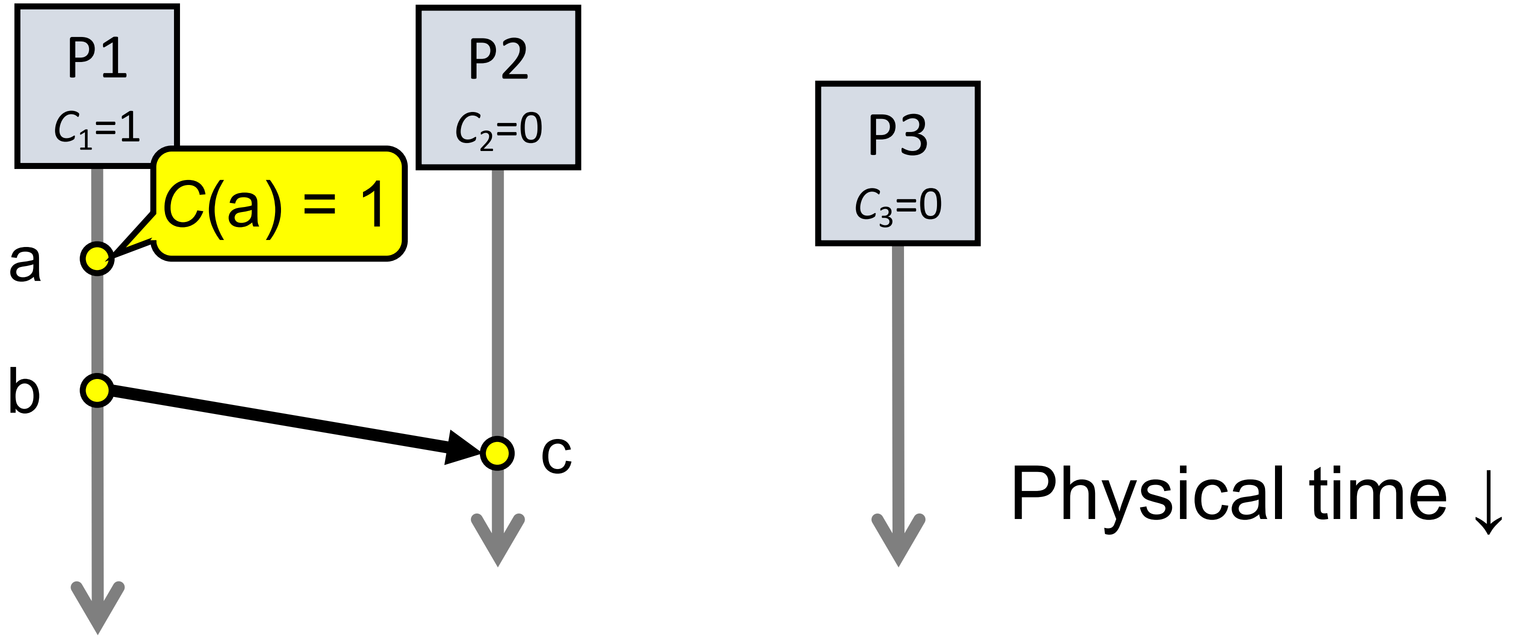
- Each process P_i maintains a local clock C_i

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



The Lamport Clock algorithm

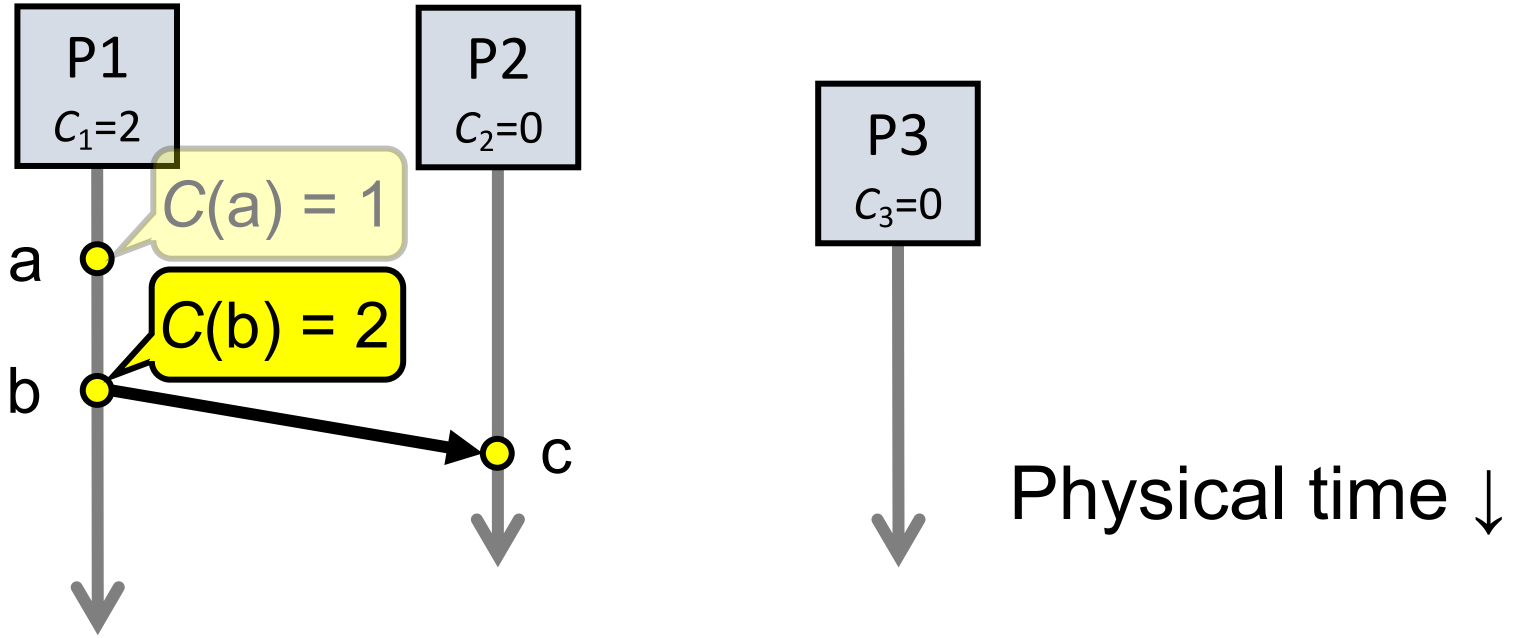
1. Before executing an event a, $C_i \leftarrow C_i + 1$:
    - Set event time $C(a) \leftarrow C_i$



The Lamport Clock algorithm

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

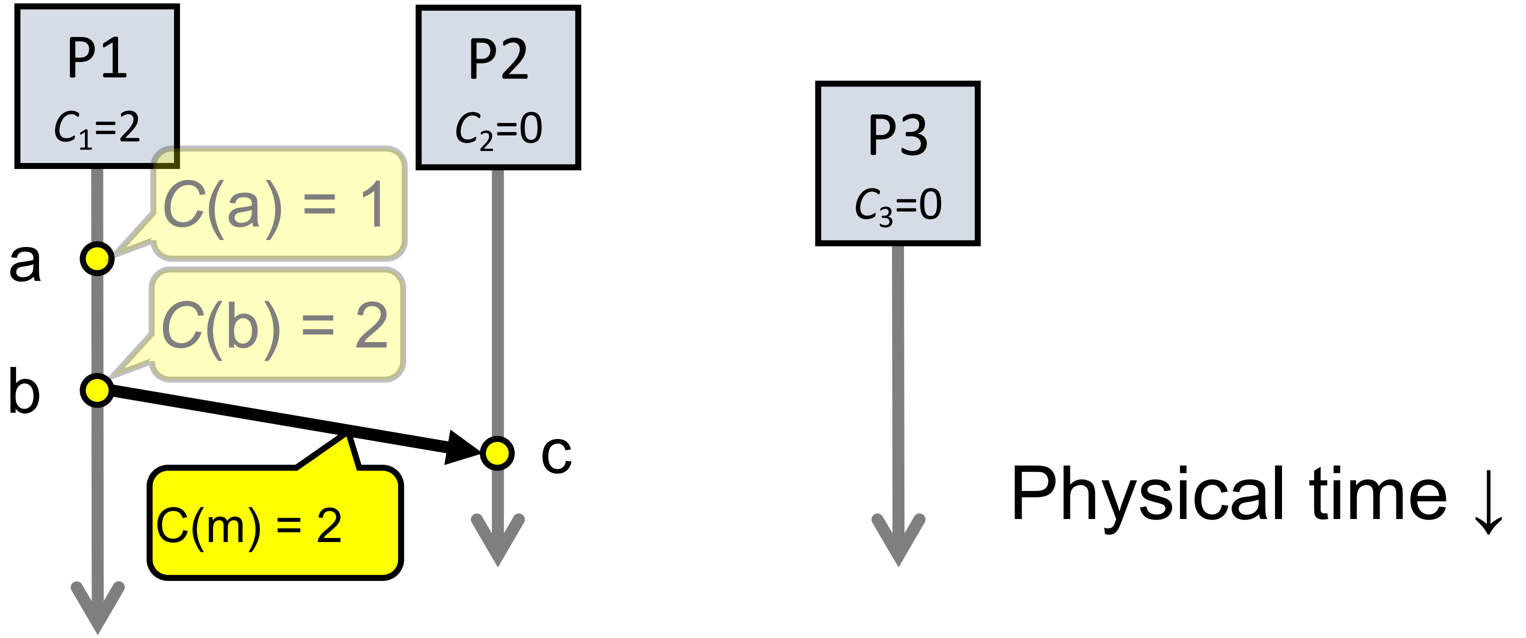
   - Set event time $C(\mathrm{b}) \leftarrow C_i$



The Lamport Clock algorithm

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

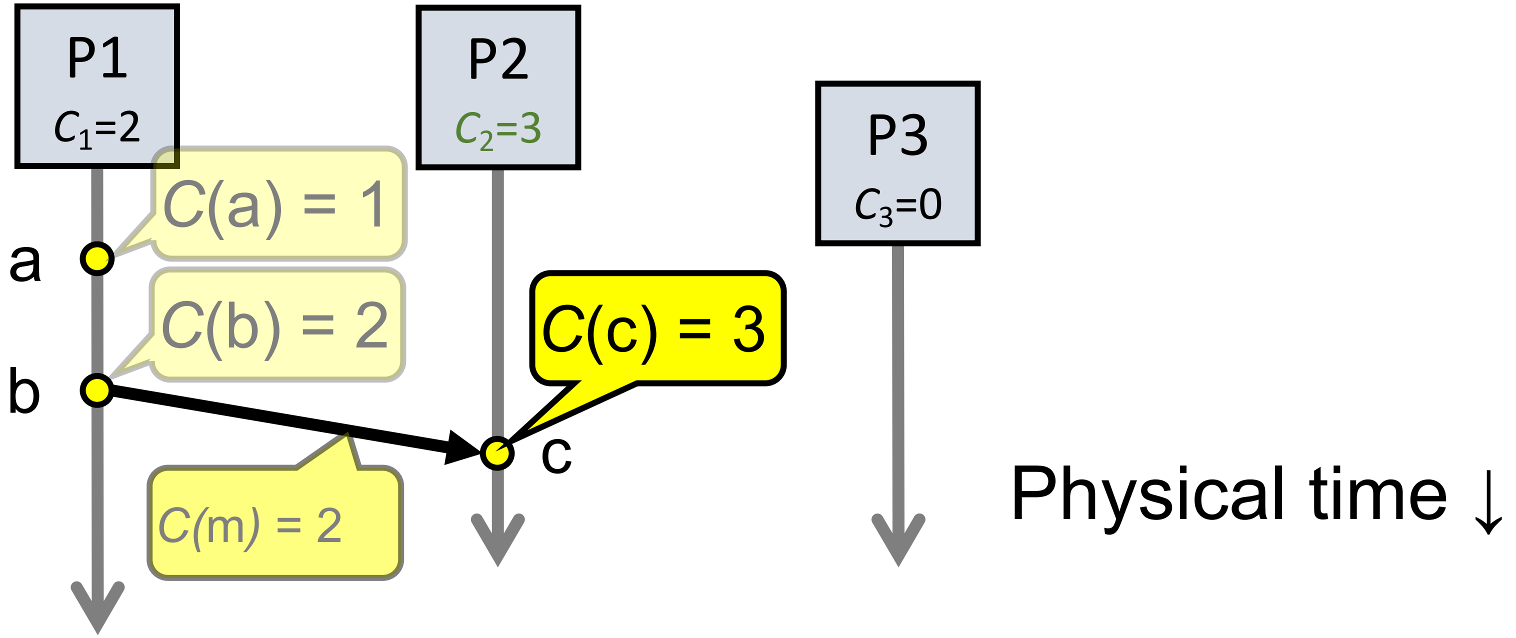
2. Send the local clock in the message m



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

Lamport Timestamps: Ordering all events

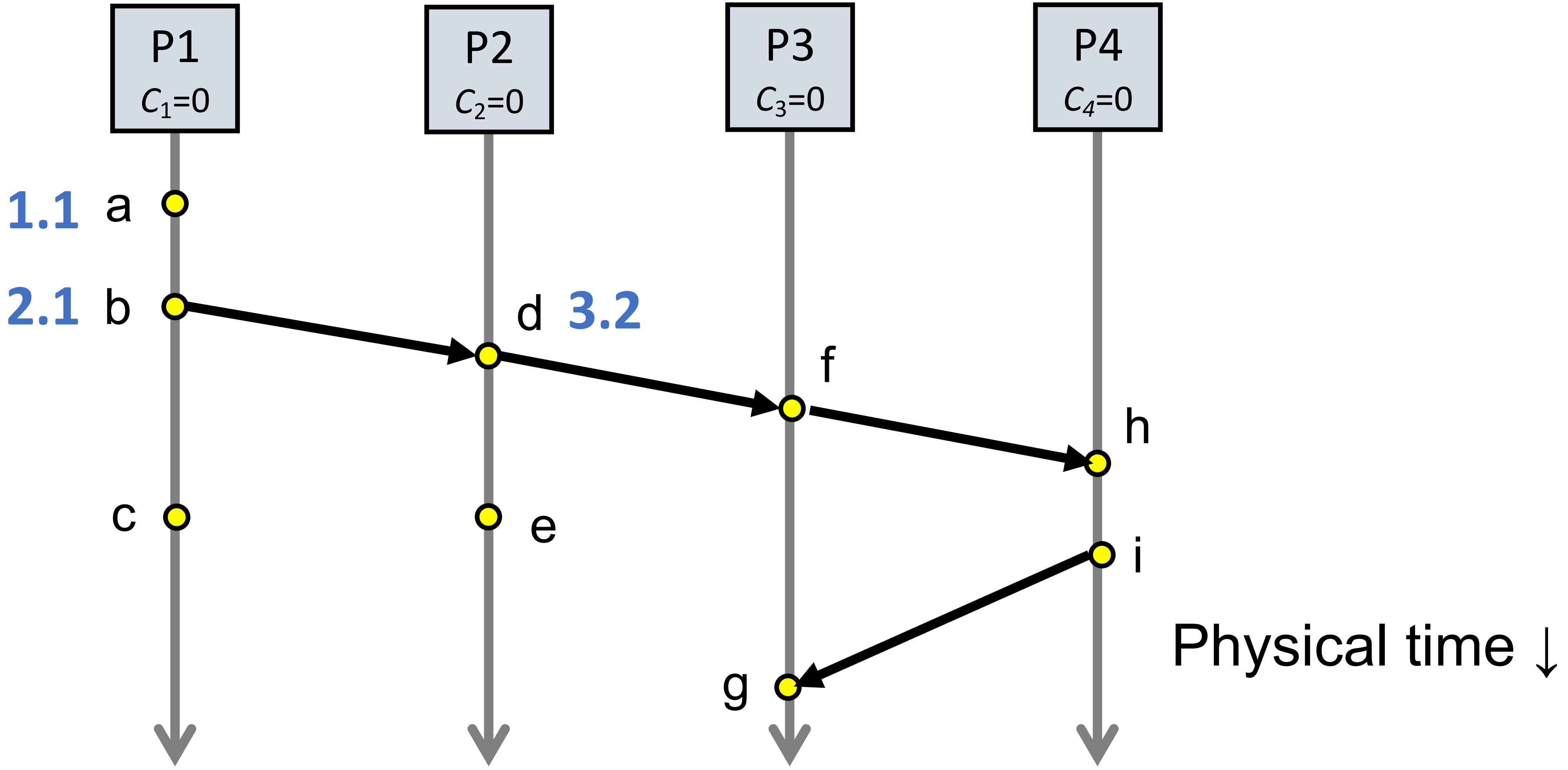
- Break ties by appending the process number to each event:

  1. Process P_i timestamps event e with $C_i(e).i$

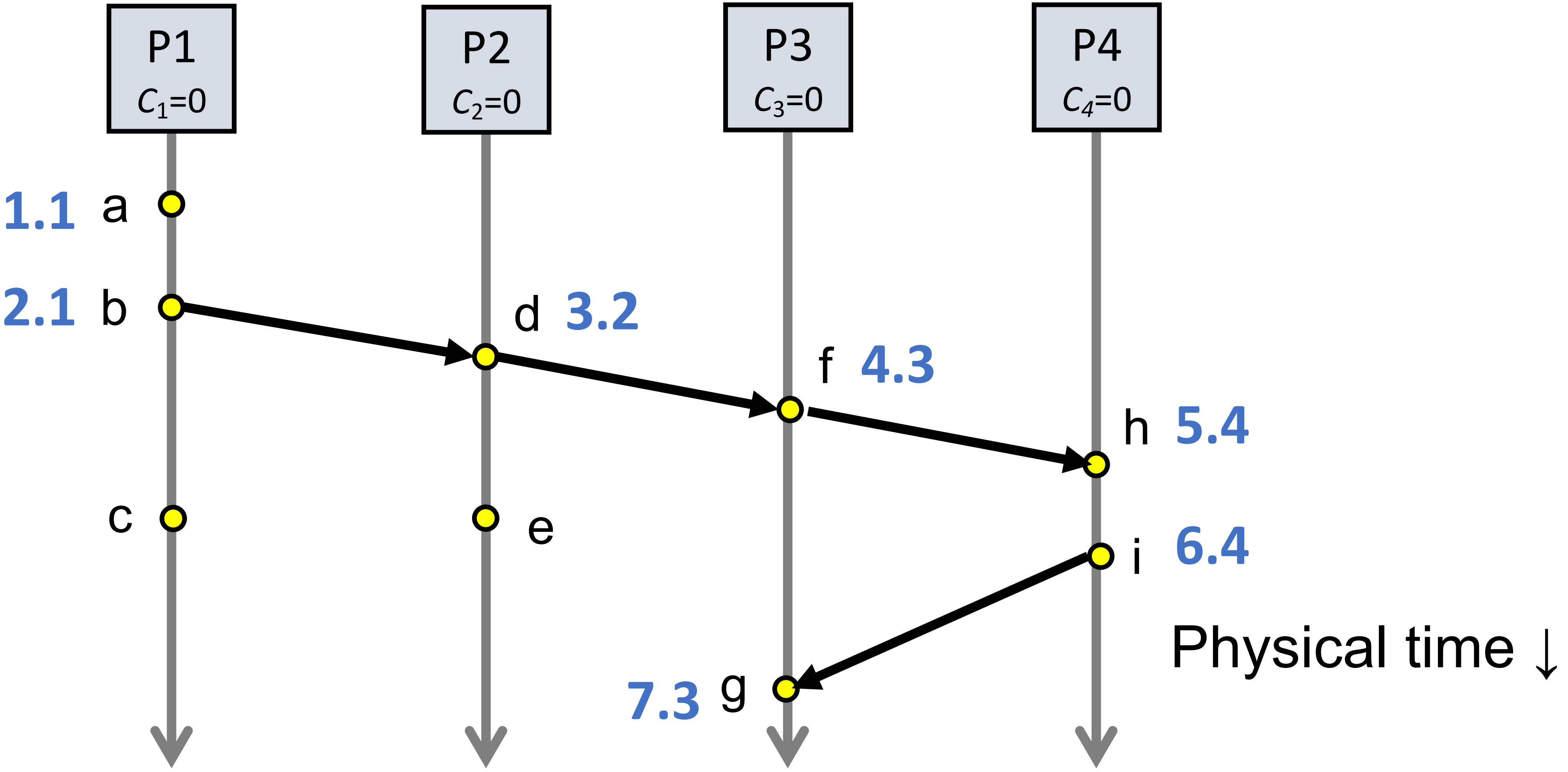
  2. $C(a).i < C(b).j$ when:
     - $C(a) < C(b)$, or $C(a) = C(b)$ and $i < j$

- 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

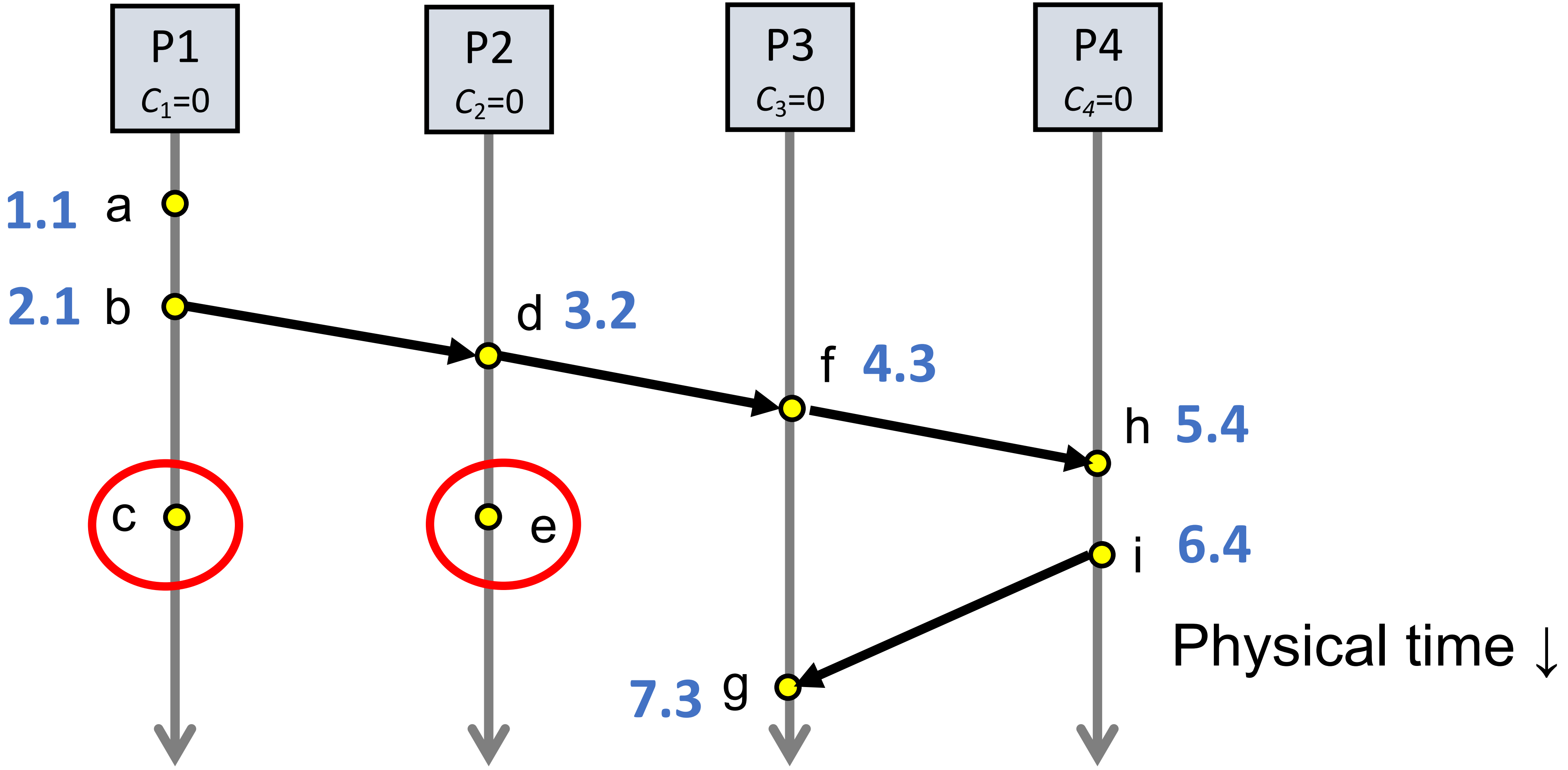
Order all these events (with total ordering)



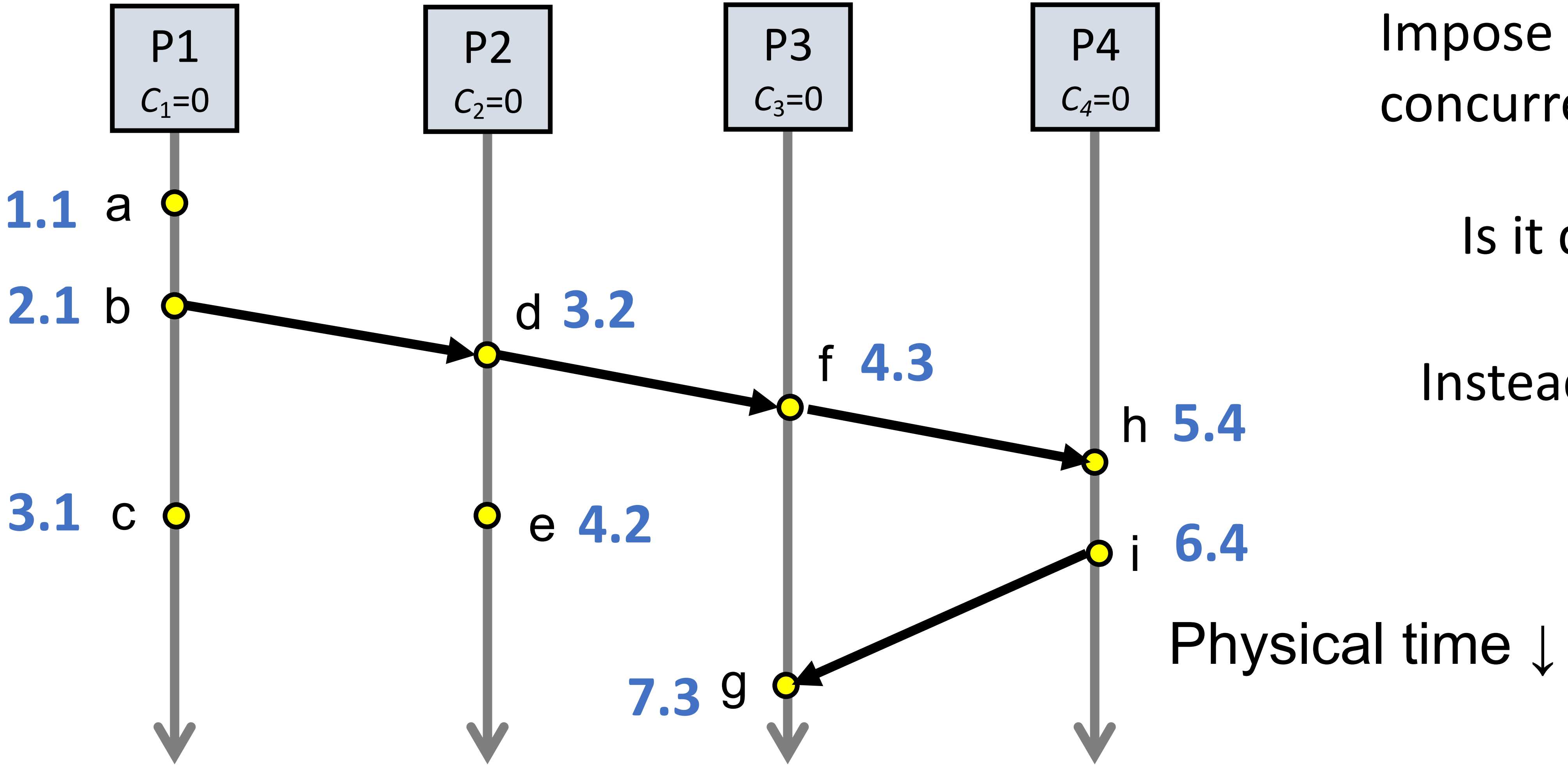
Order all these events (with total ordering)



Which events are concurrent with h?



How are c, d, e, f ordered (in the total order)?



Impose an order on concurrent events!

Is it d, f, c, e?

Instead, c, d, e, f

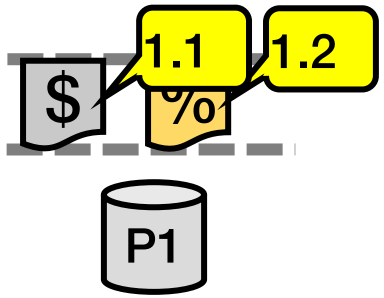
Totally-Ordered Multicast

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:



← Timestamps

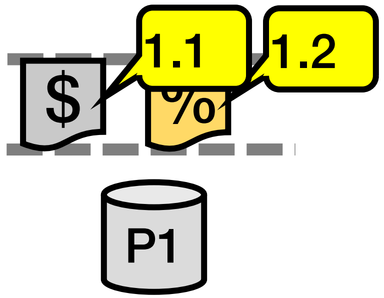
Totally-Ordered Multicast

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:



← Timestamps

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 <u>everyone</u> has ack'ed from <u>head</u> of 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)

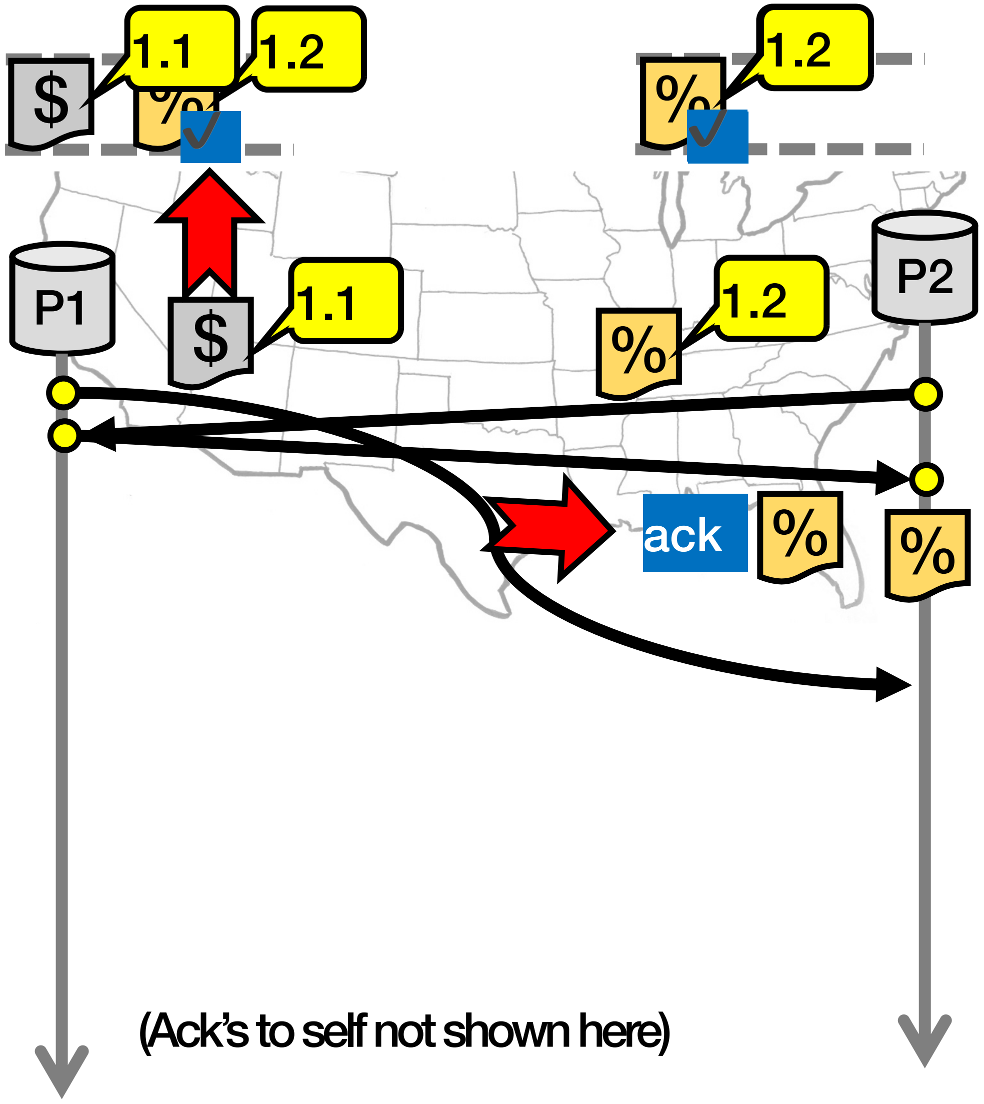
3. On receiving an acknowledgement:
   - Mark corresponding update acknowledged in your queue

4. Remove and process updates <u>everyone</u> has ack'ed from <u>head</u> 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

✗ P2 processes %



Totally-Ordered Multicast (Correct version)

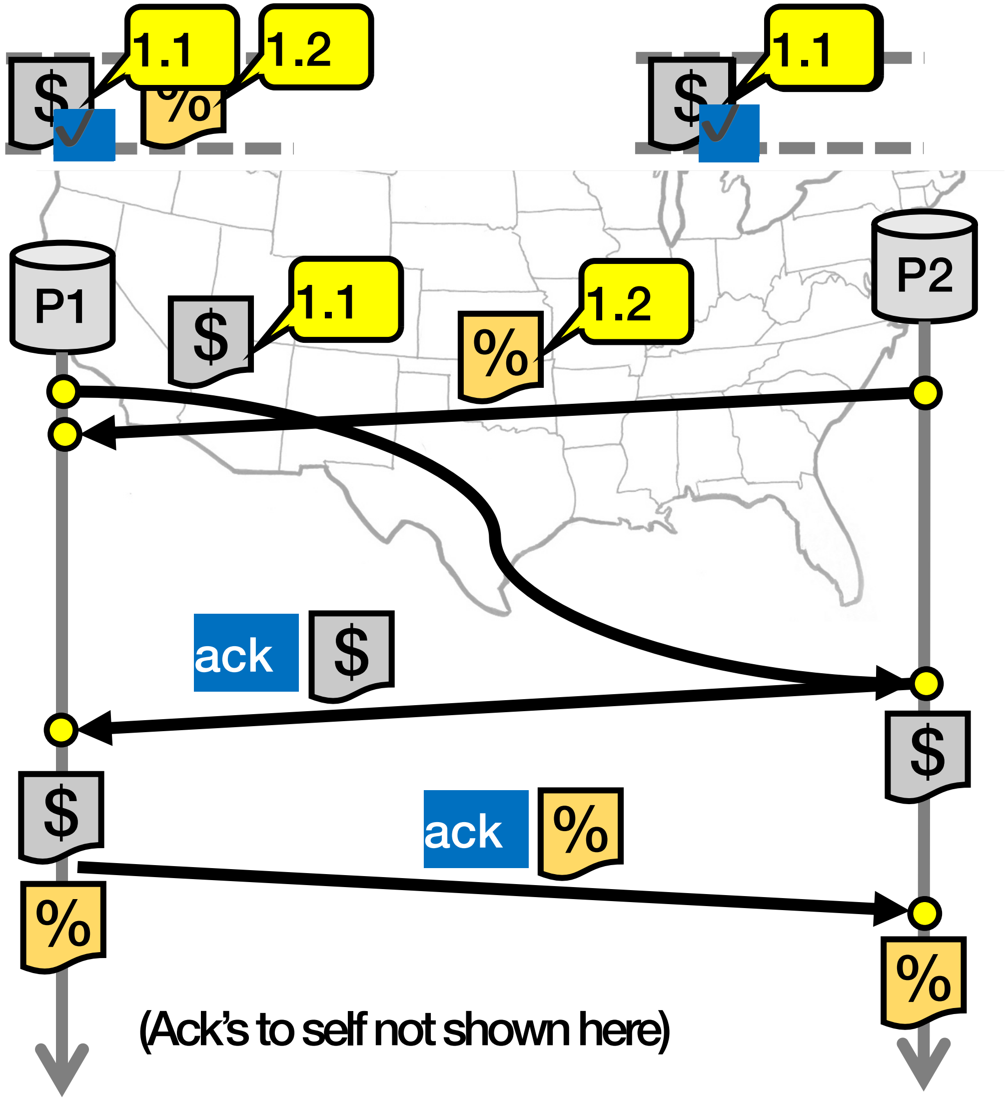
1. On receiving an update from client, broadcast to others (including yourself)

2. On receiving or processing an update:
   a) Add it to your local queue, if received update
   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?*

- 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?)

Intro to Concurrency Conclusion

- Concurrency is great for performance, hard to reason about, and often unavoidable in systems
- Replicated DB example
  - Concurrent updates can lead to inconsistency between replicas
  - Lamport clocks can order events in a distributed system
  - Lamport clocks + careful protocol = correct replication
- What is “correct”?