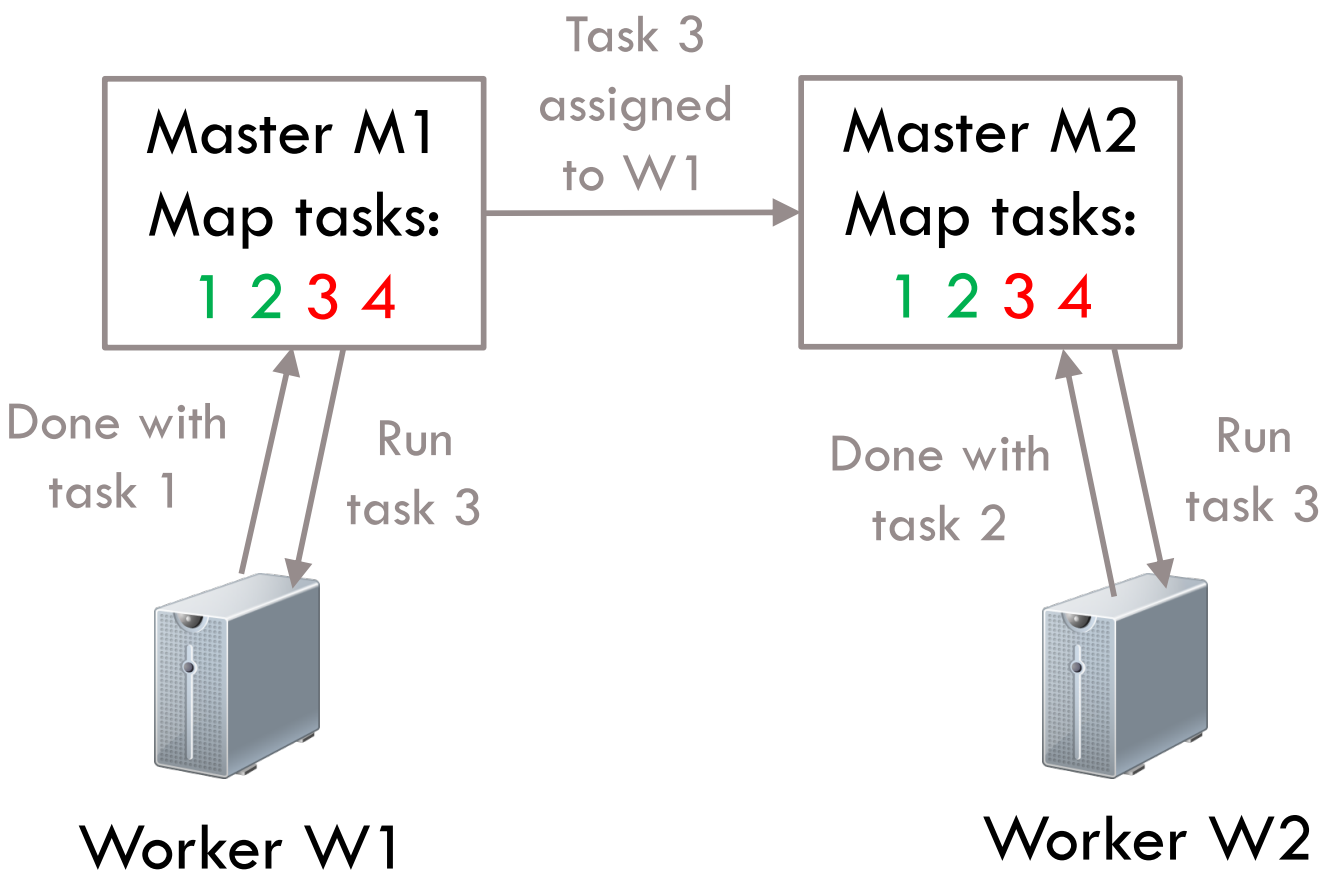
LECTURE 4

Synchronization and Logical Clocks

Handling Failures in MapReduce

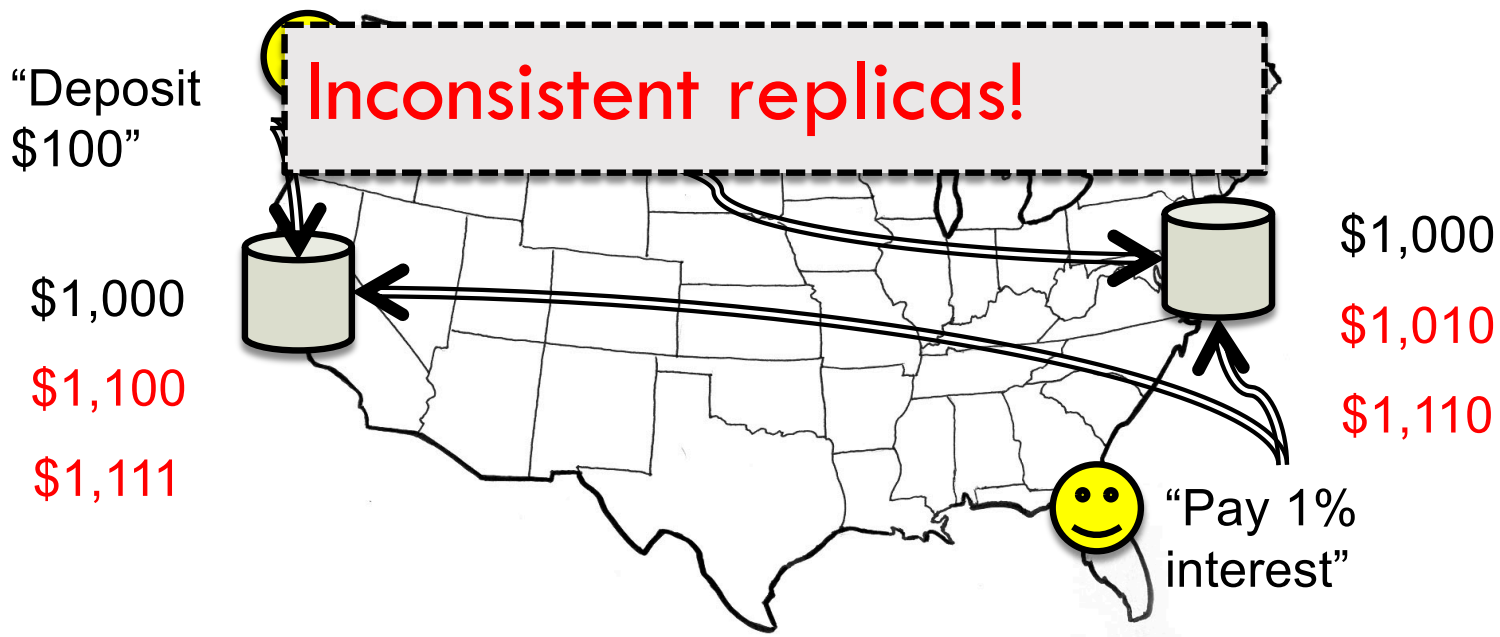
- If Master fails, can we start up a new Master and resume execution of job?
- No!
  - Lost all state about execution status of tasks
- Must replicate Master state to tolerate failures

Replicating MapReduce Master



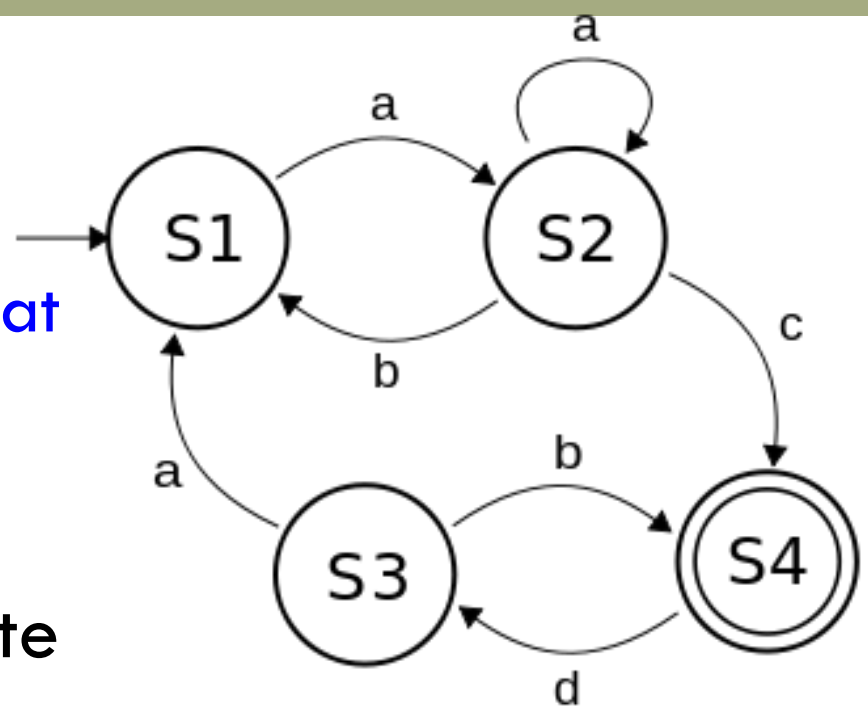
Replicating Bank Database

- One copy of account in SF, another copy in NY
- Clients send **updates to both** copies



Synchronizing Replicas

- How to ensure replicas are in sync?
  - Apply updates in same order at all replicas



- Model every replica as a state machine
  - Given identical initial states, applying updates in same sequence results in same final state

→ Replicated state machine (RSM)

Replicated State Machine

- Replicate over time
  - Start a new replica after one goes down
  - Crash failures
- Replicate over space
  - Run multiple replicas simultaneously
  - Fail-stop failures (unable to handle exceptions)

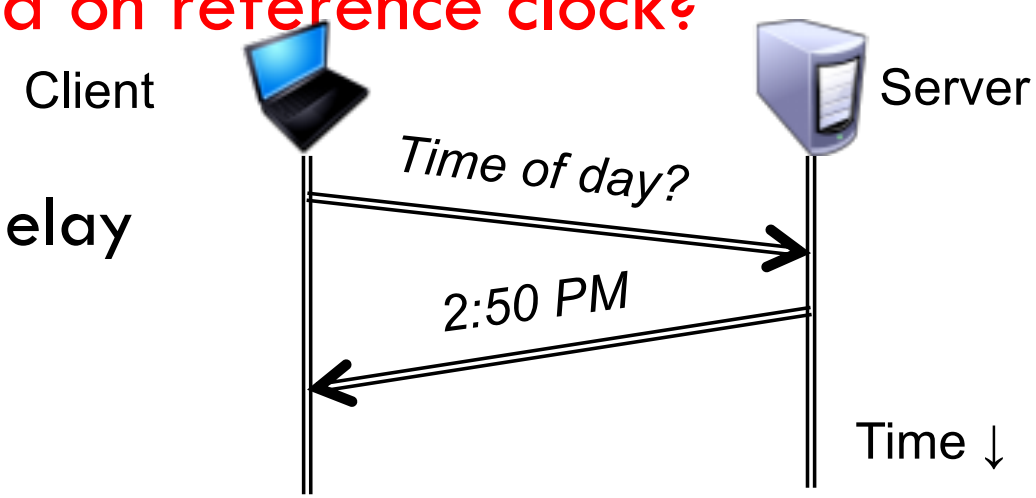
Implementing RSM

- Order updates based on time of receipt
- Challenge: Clocks not in sync across replicas

Clock Syncing: Challenges

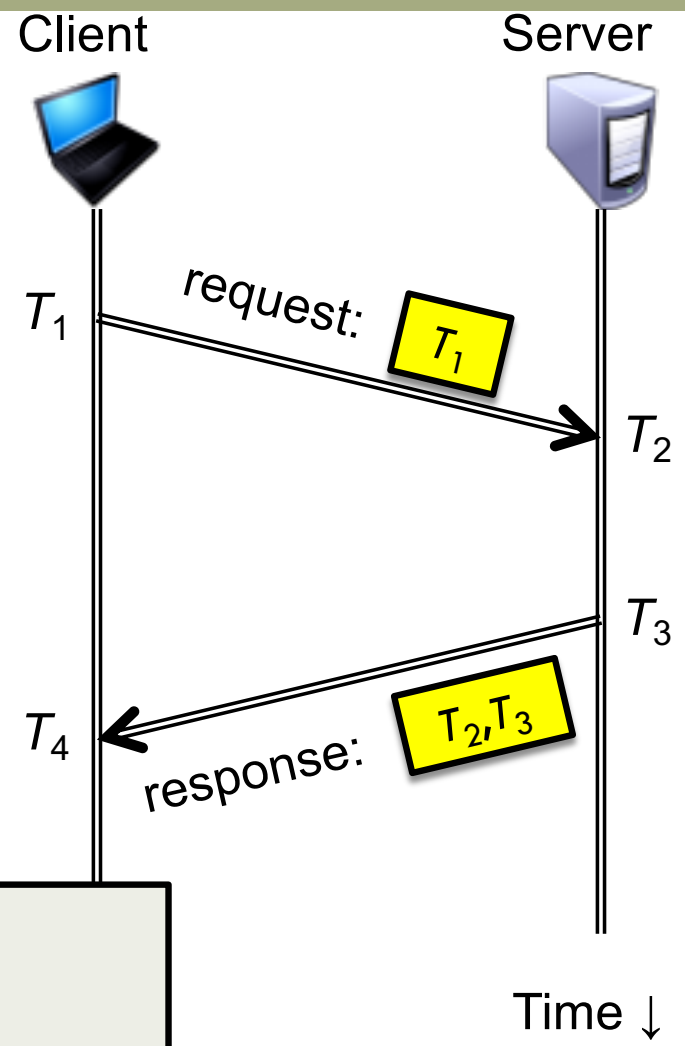
- Leverage GPS broadcasts?
  - Time from GPS accurate to about 1 microsecond
  - Power hungry and does not work indoors

- Correct for skew based on reference clock?
  - Clock drift
  - Unbounded network delay



Cristian's algorithm

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 receives the server's response T_4



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

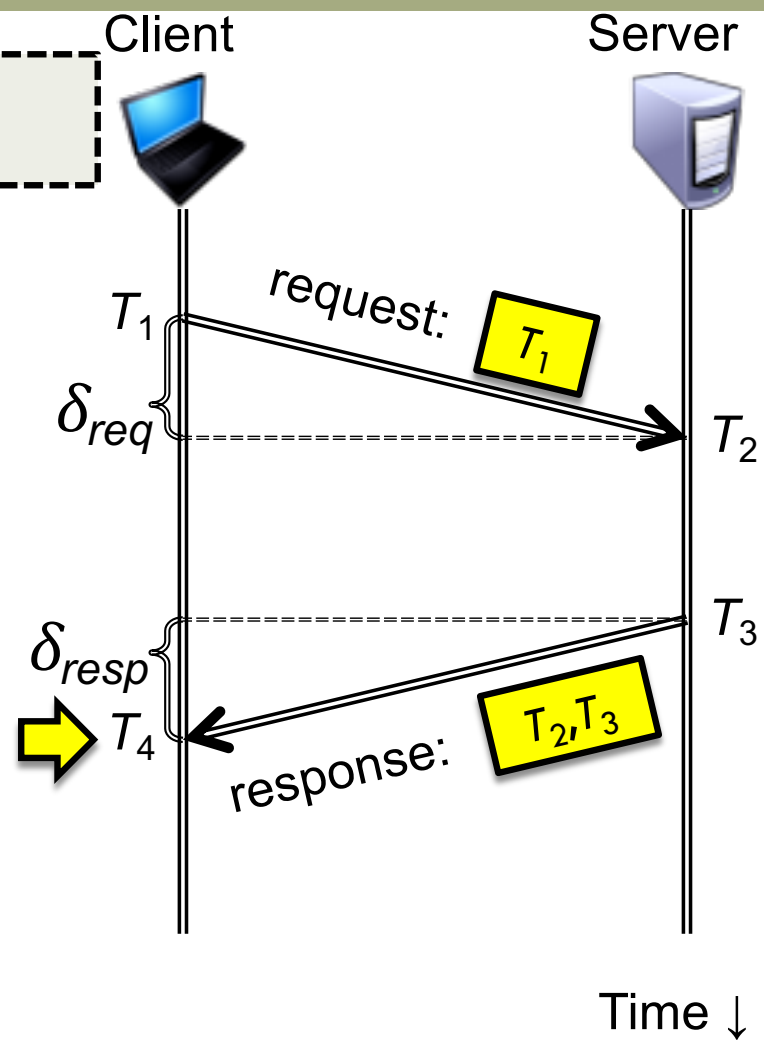
Cristian's algorithm

Goal: At T_4, client sets clock ← $T_3 + \delta_{resp}$

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

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

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



Further refinements

- Limitation of Cristian algorithm
  - Relies on timeserver availability
- NTP
  - Distributed ecosystem for time synchronization
  - Tolerates failures of network and time servers
  - Leverages heterogeneous clocks

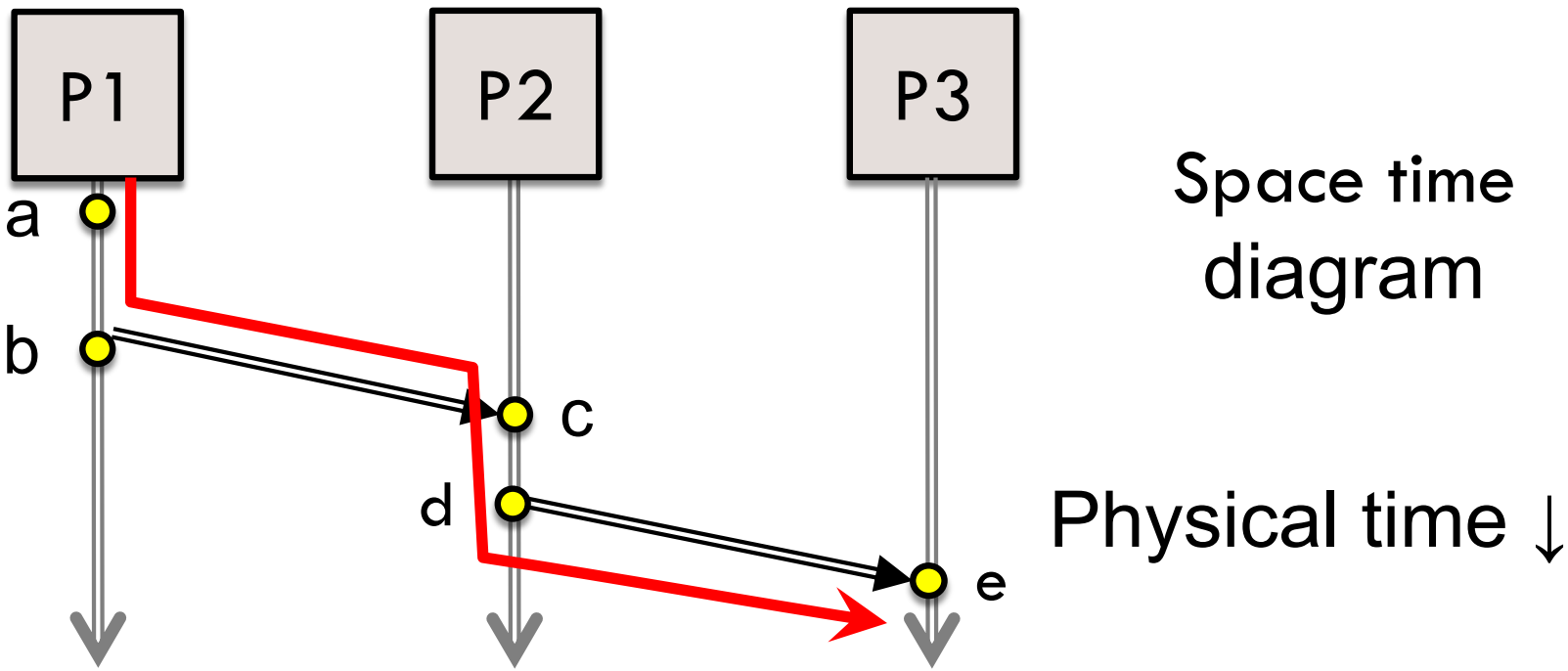
- All replicas must apply updates in same order
- Ordering updates based on time of receipt is hard because of clock skew and drift
- Challenge in synchronizing clocks:
  - Unbounded network delay

Idea: *Logical* clocks

- Landmark 1978 paper by Leslie Lamport
- Insight: Disregard precise clock time
  - Only relationships between events matter
- Associate every event with "logical time"
  - Preserve "happens before" relationships
  - If *a* *happens before* *b*, then *clock(a)* < *clock(b)*

Defining "happens-before"

1. In **same process**, if **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**



Back to RSMs …

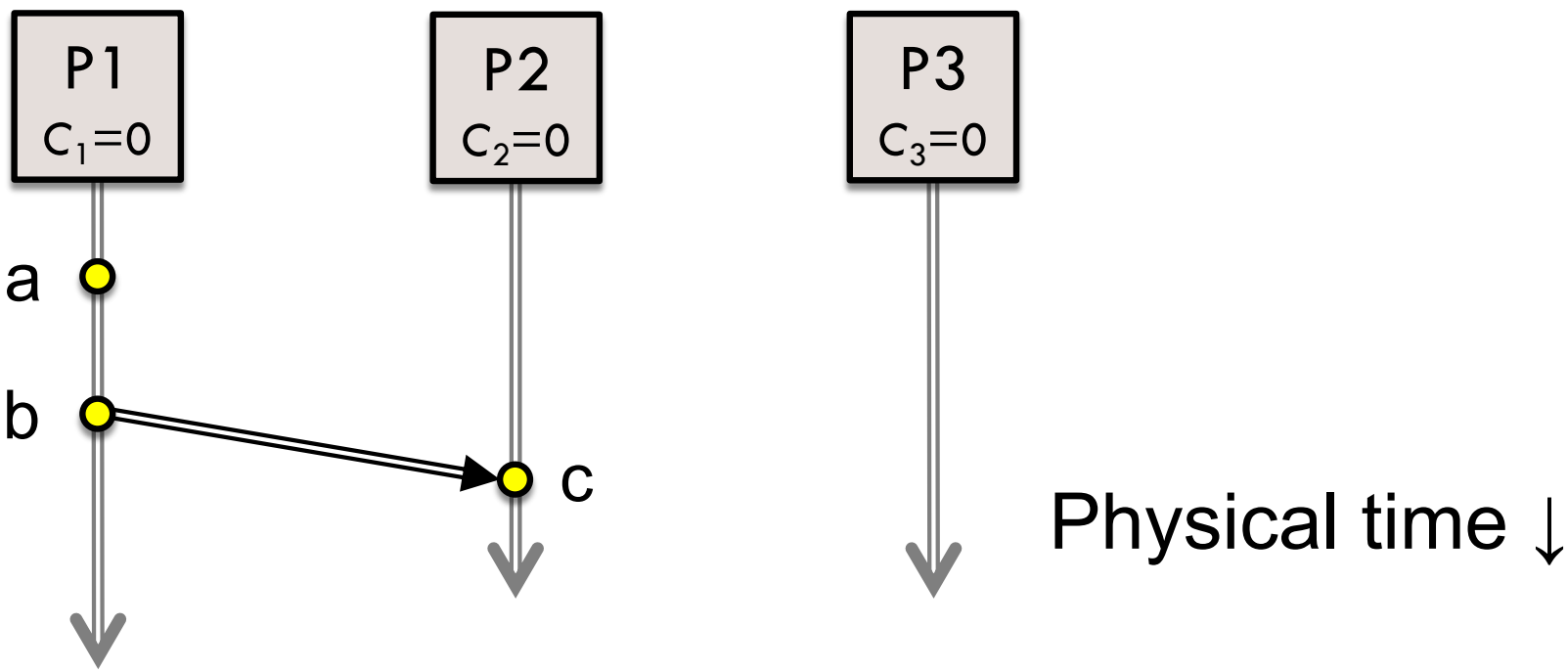
- How do we use "happens before" to ensure consistent ordering of updates in an RSM?

Lamport clock: Objective

- Associate any event **a** with a *clock time* C(**a**)
- Clock condition: If **a** → **b**, then C(**a**) < C(**b**)
- Order events at all nodes based on clock time

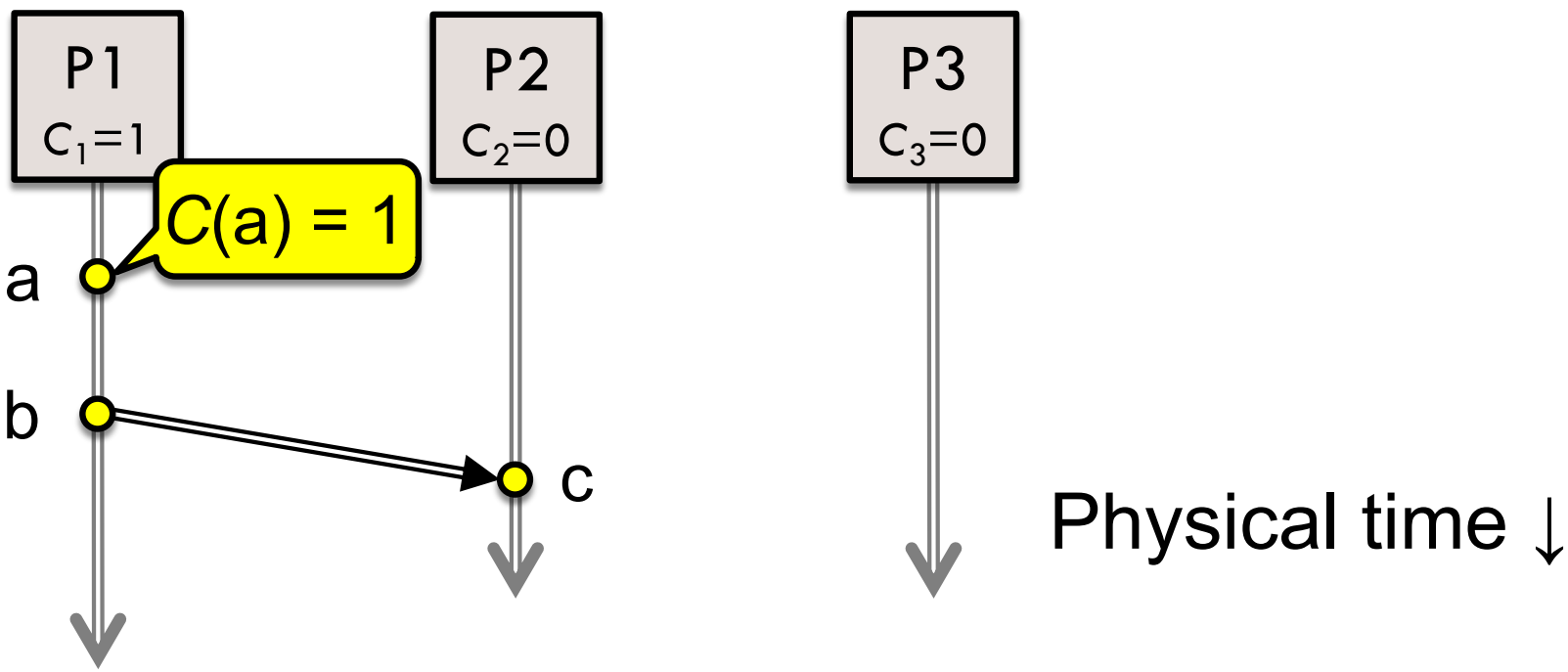
The Lamport Clock algorithm

- Each process $\mathbf{P}_i$ maintains a local clock $\mathbf{C}_i$



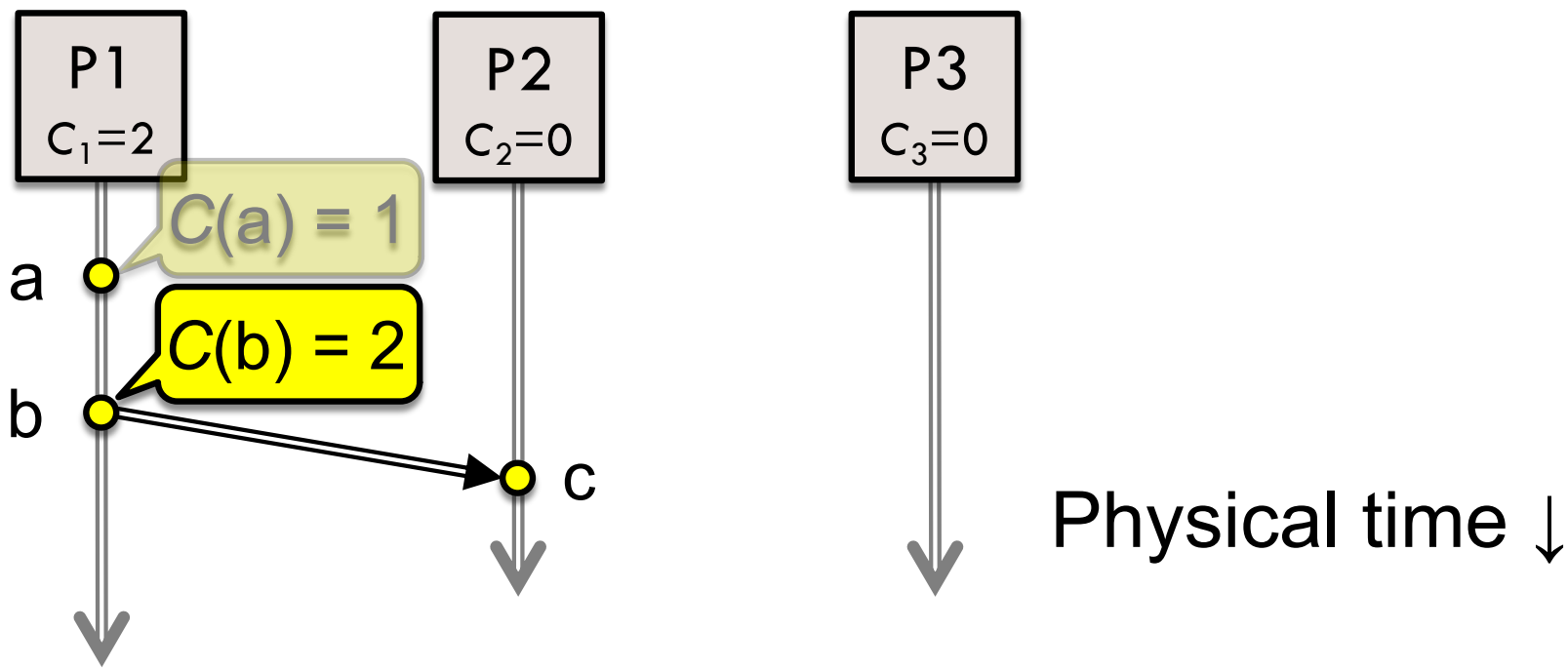
The Lamport Clock algorithm

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



The Lamport Clock algorithm

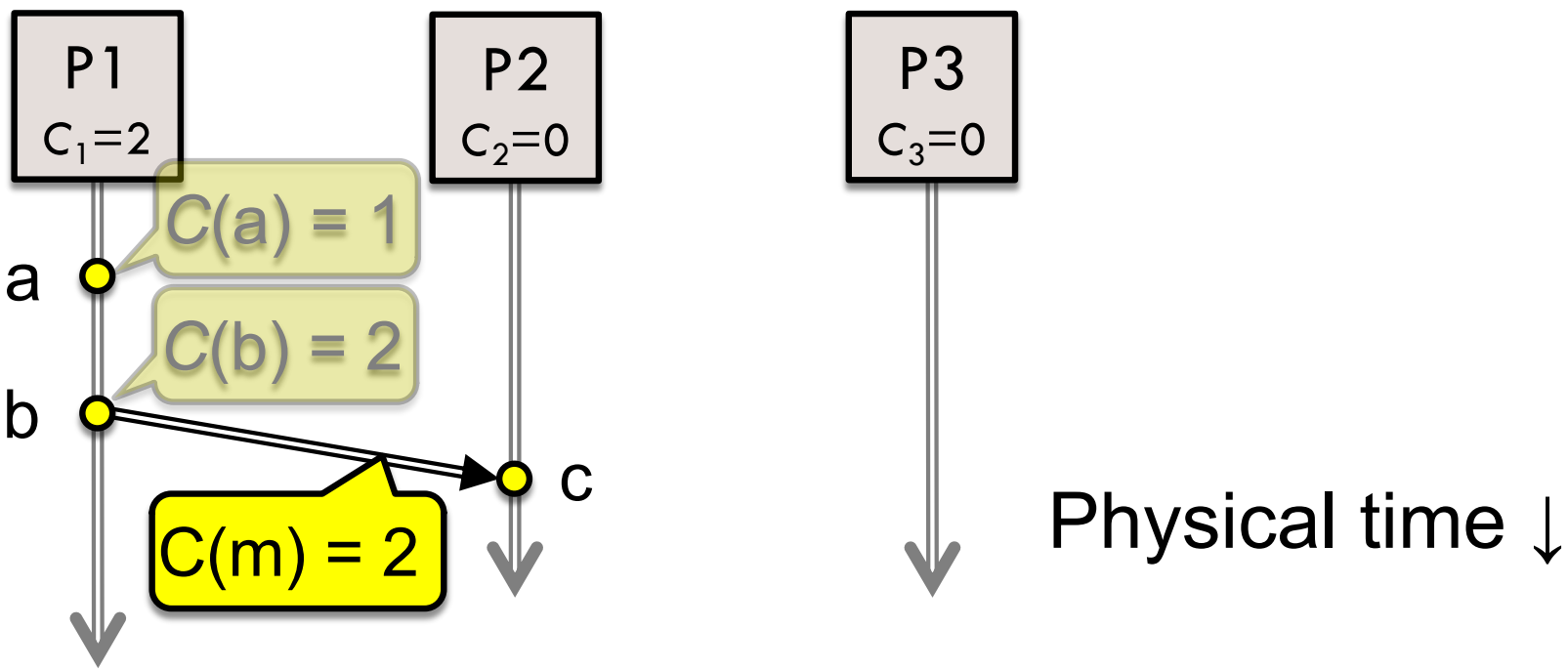
1. Before executing an event, $C_i \leftarrow C_i + 1$



The Lamport Clock algorithm

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

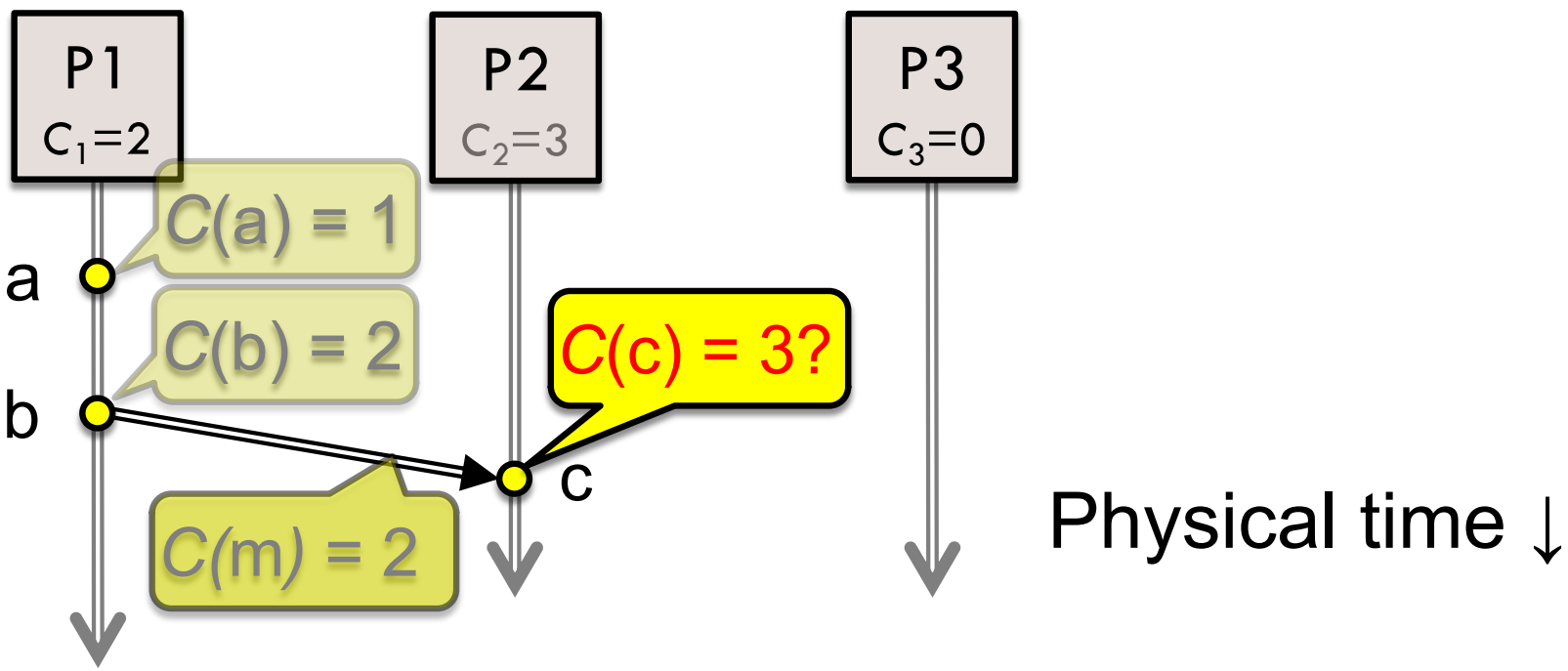
2. Send the local clock in the message **m**



The Lamport Clock algorithm

3. On process $\mathbf{P}_i$ receiving a message **m**:

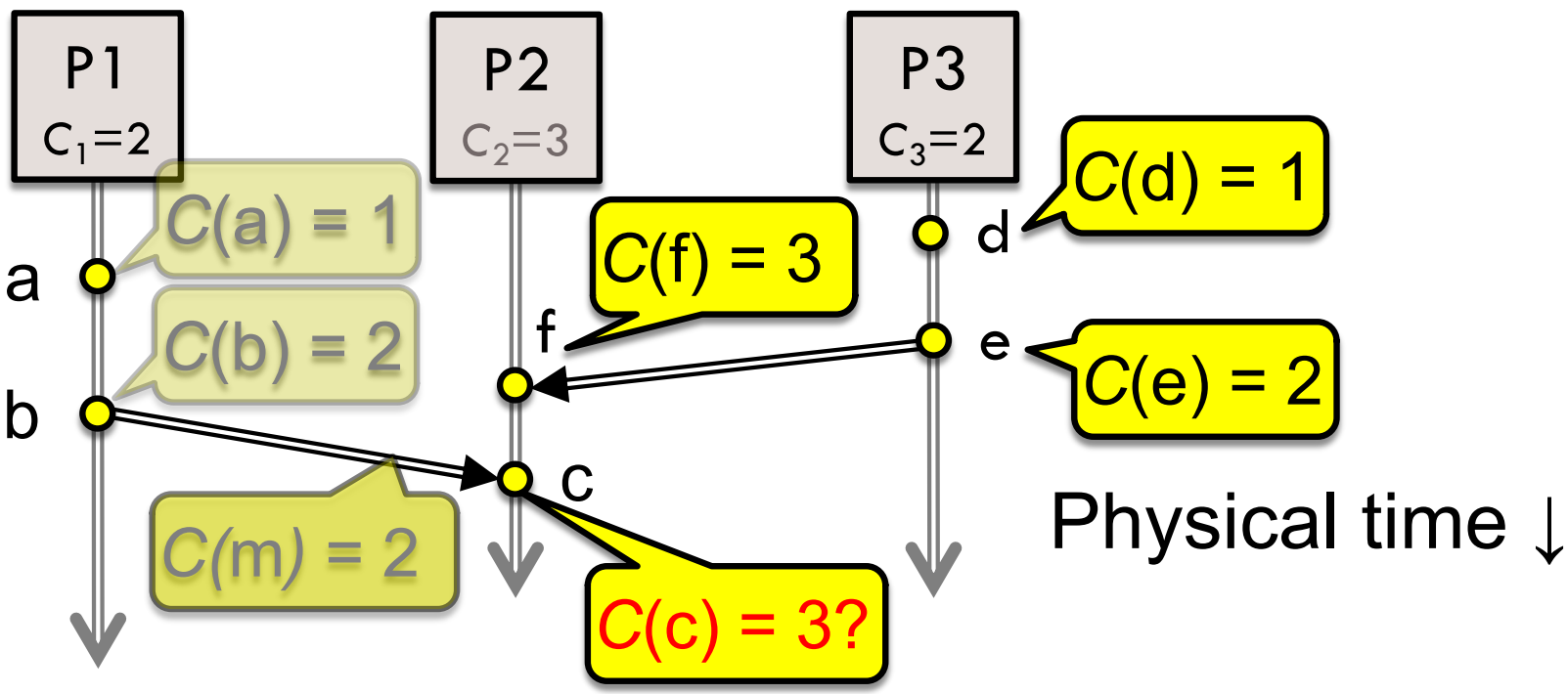
- What time to set for receive event? C(**m**) + 1?



The Lamport Clock algorithm

3. On process $\mathbf{P}_i$ receiving a message **m**:

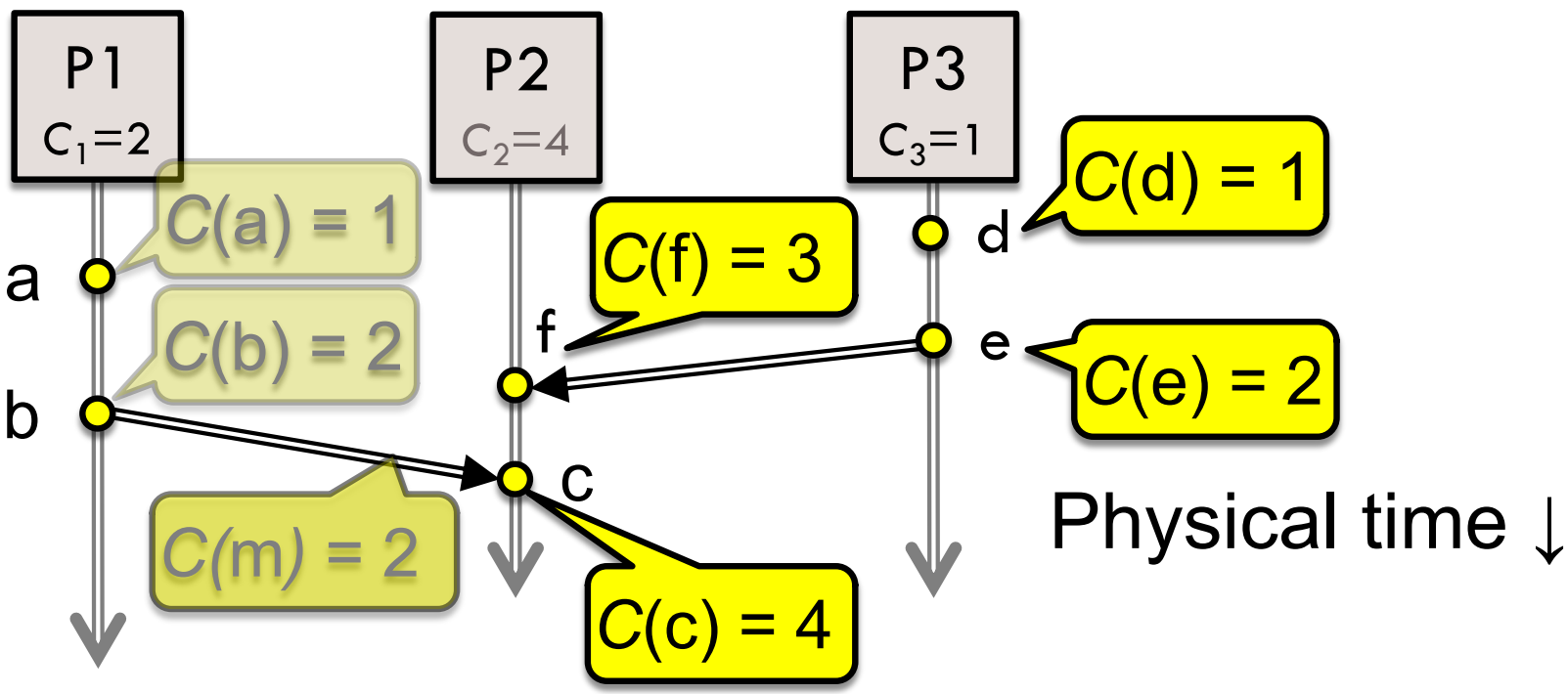
- What time to set for receive event? C(**m**) + 1?



The Lamport Clock algorithm

3. On process $\mathbf{P}_i$ receiving a message **m**:

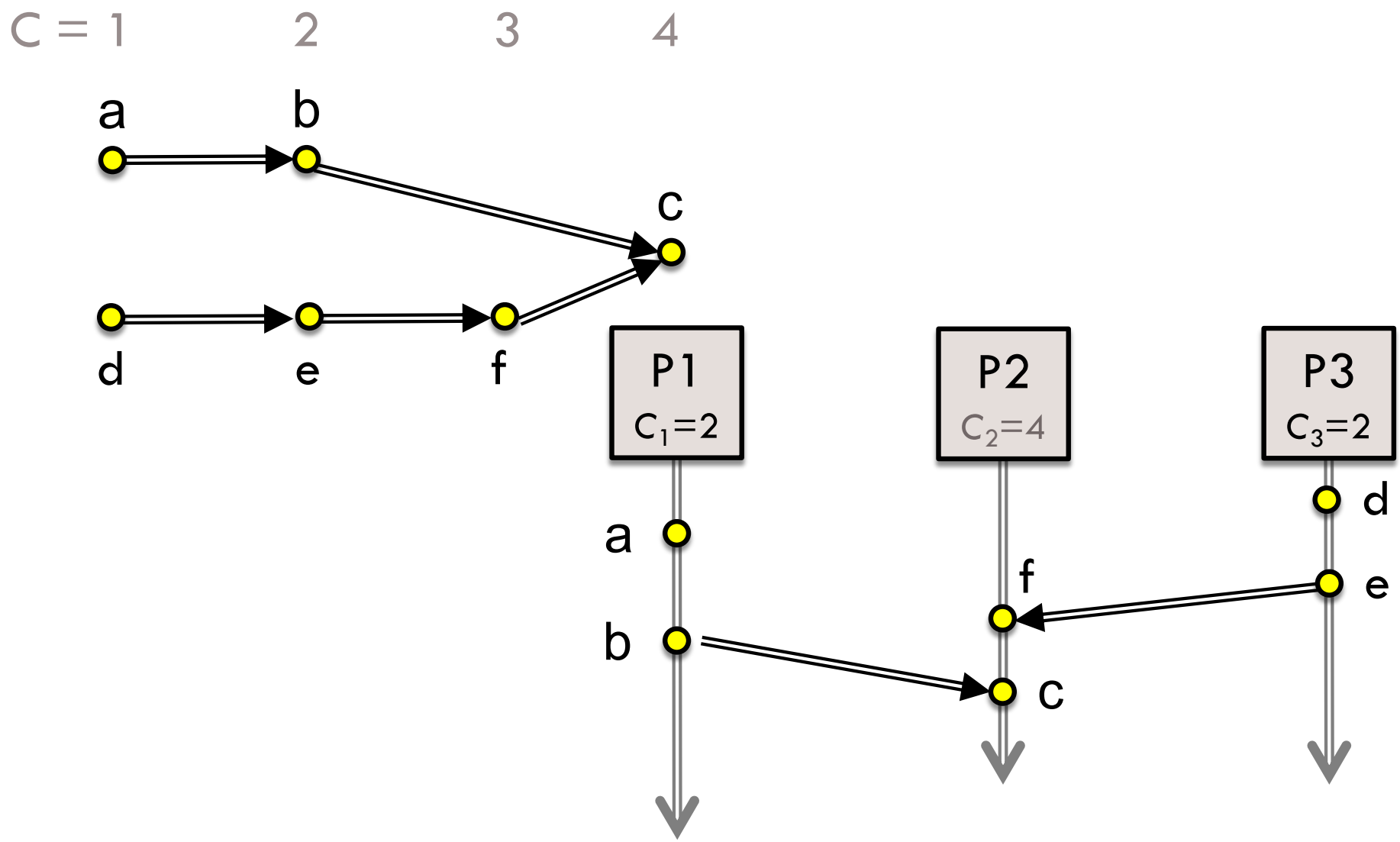
- Set C_i **and** time of receive event to 1 + **max**{ C_i, C(**m**) }



Lamport clock

- Associate any event **a** with a *clock time* C(**a**)
- Clock condition: If **a** → **b**, then C(**a**) < C(**b**)
- If a causally leads to b, then C(a) < C(b)

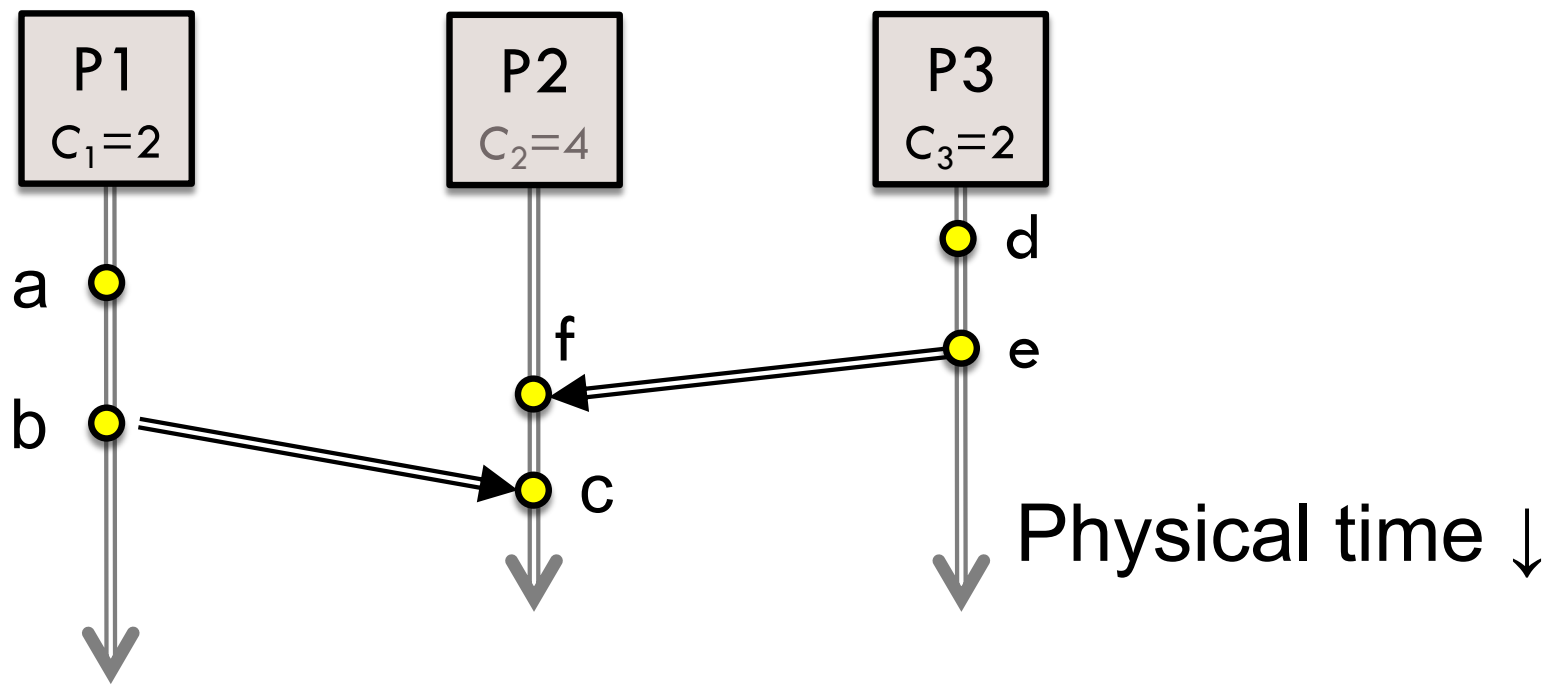
Ordering all events



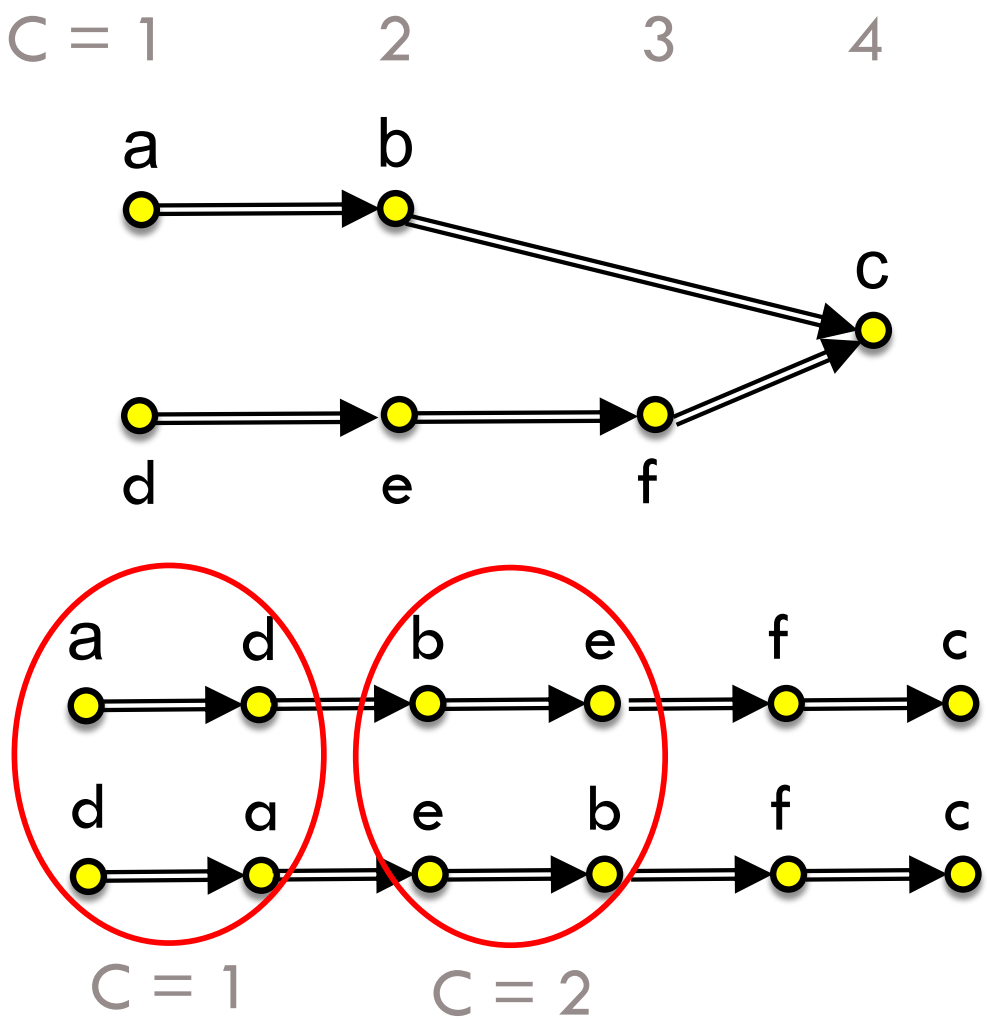
Concurrent events

- Not all events are related by ***"happens before"***

- **a, d** not related by → so concurrent, written as **a || d**



Ordering all events



- How to ensure *total* ordering at all replicas?
- Can two events at same process have same clock value?

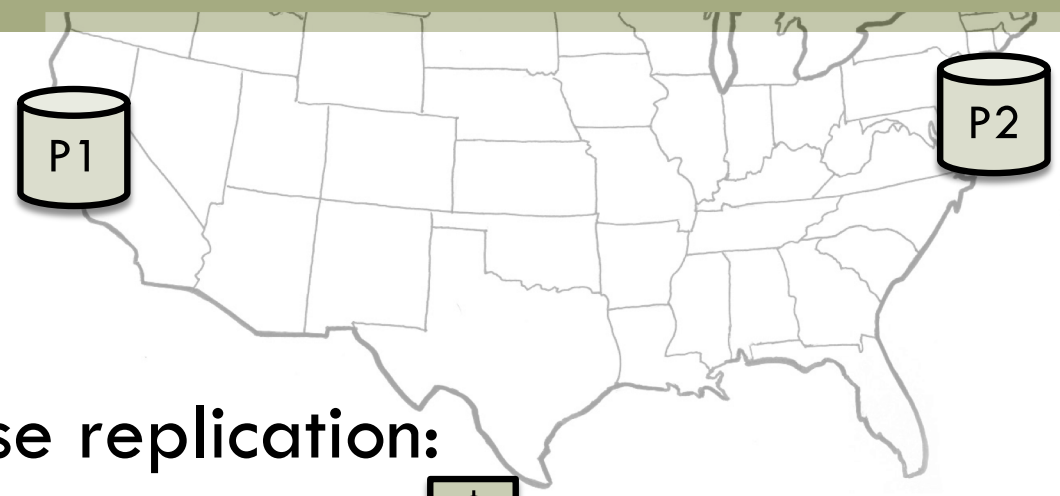
Ordering all events

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

  1. Process $\mathbf{P}_i$ timestamps event **e** with $C_i(\mathbf{e}).i$
  2. $C(\mathbf{a}).i < C(\mathbf{b}).j$ when:
     - $C(\mathbf{a}) < C(\mathbf{b})$, or $C(\mathbf{a}) = C(\mathbf{b})$ and $i < j$

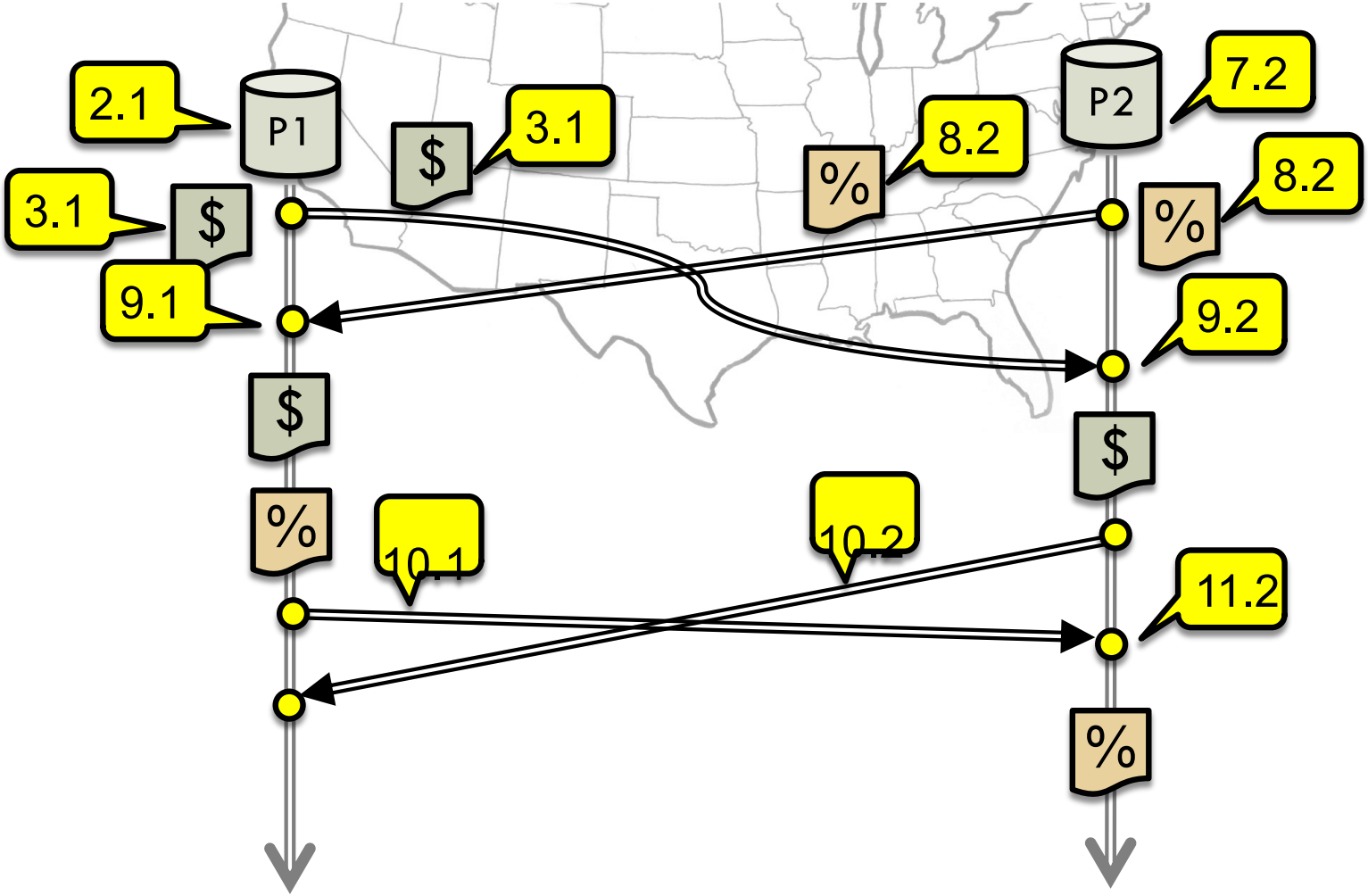
- Now, for any two events **a** and **b**, $C(\mathbf{a}) < C(\mathbf{b})$ or $C(\mathbf{b}) < C(\mathbf{a})$
  - Total ordering of events

Making concurrent updates consistent



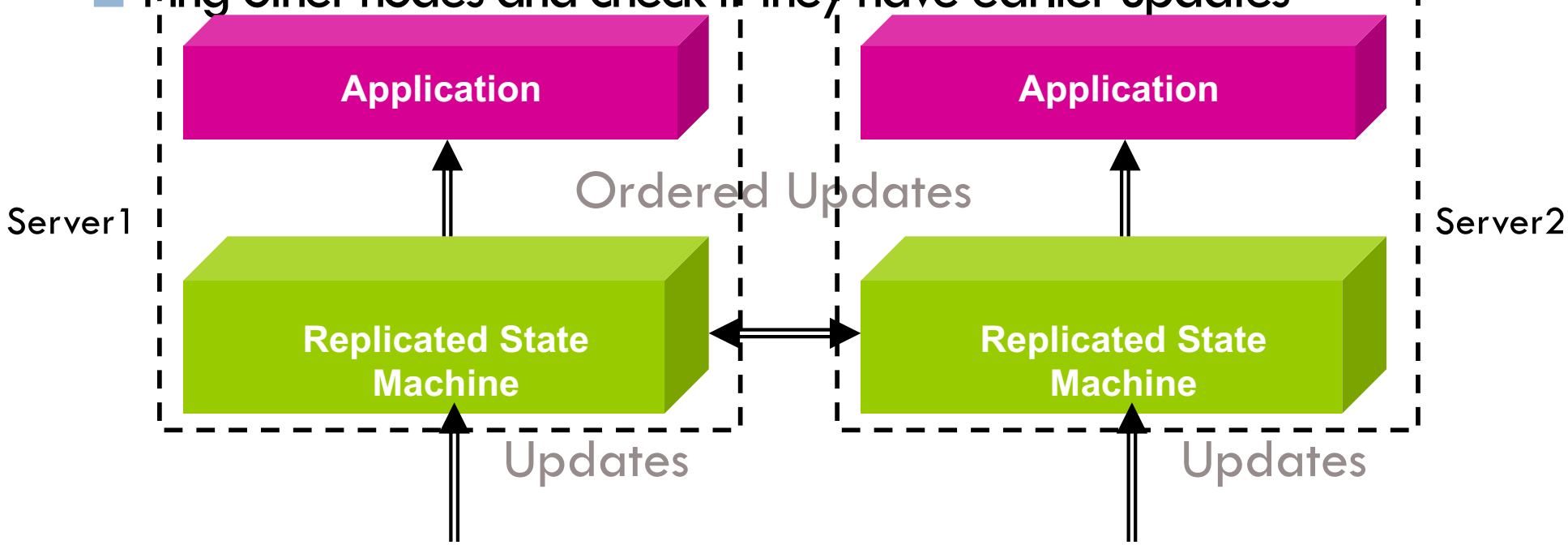
- Recall multi-site database replication:
  - San Francisco (**P1**) deposited $100: $
  - New York (**P2**) paid 1% interest: %
- Applying updates in different order made replicas inconsistent
- All replicas apply updates in Lamport clock order

RSMs with Lamport Clocks

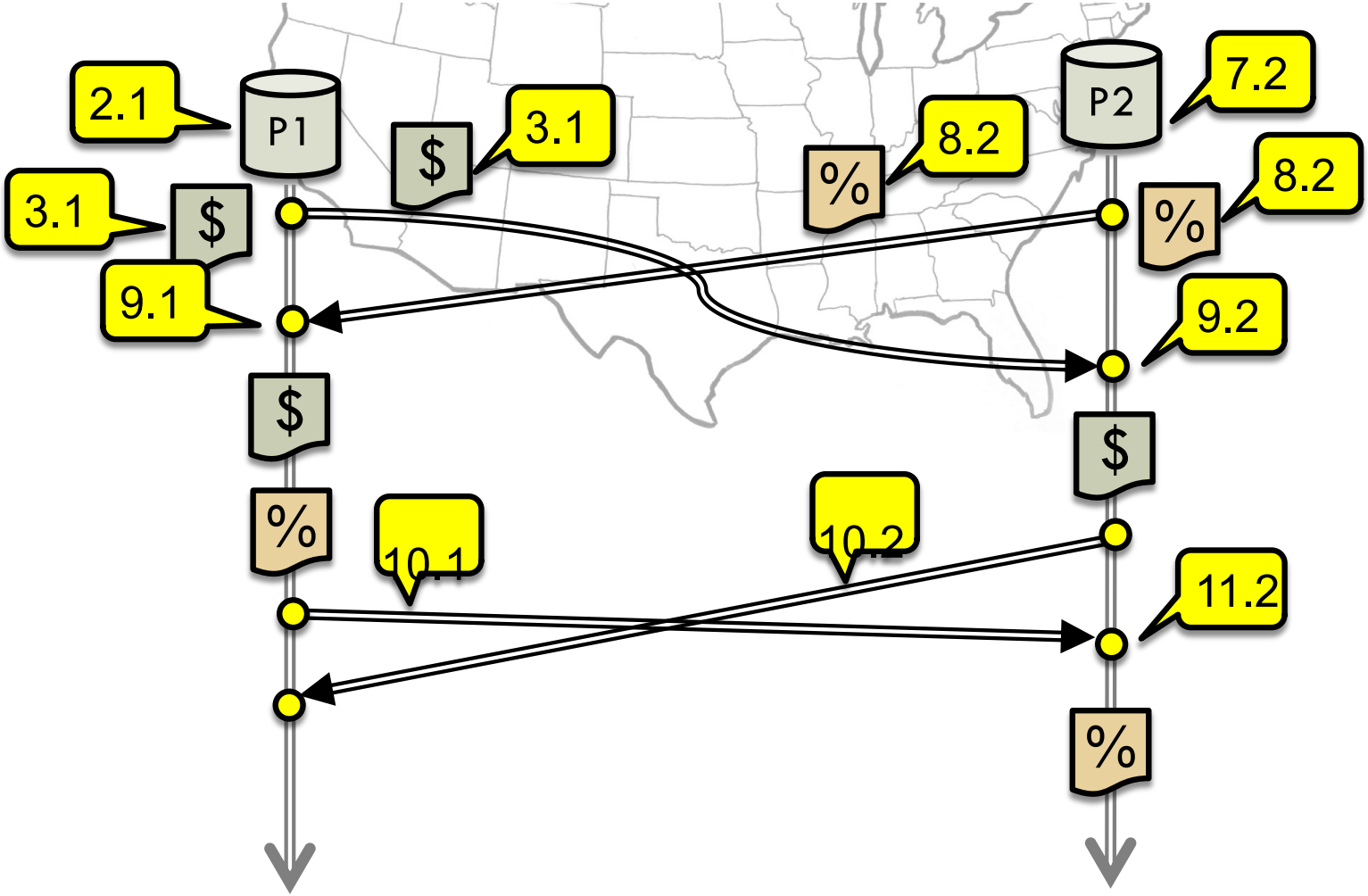


Consistent Ordering of Updates

- Key idea: Place events into a **local queue**
  - **Sorted** by increasing C(x)
- How to tell if update at head of queue can be applied?
  - Ping other nodes and check if they have earlier updates



RSMs with Lamport Clocks



Reducing Waiting

- How to reduce time between receiving an update and applying it?
- Periodically ping all other nodes to sync clock
- Tolerate temporary inconsistency
  - Apply update immediately upon receipt
  - But, maintain log of updates in order to rollback

Causal ordering

- Ordering **all** updates at all replicas may be unnecessary
  - Example: Replication of Facebook posts
- Causal ordering suffices
  - Show comments only if post visible to user
- With Lamport clock, **a** → **b** implies C(**a**) < C(**b**)
  - But, converse is not necessarily true
  - C(**a**) < C(**b**) does not imply **a** → **b** (possibly, **a** || **b**)

Vector clock (VC)

- Label each event **e** with a vector
  - $V(\mathbf{e}) = [c_1, c_2 \ldots, c_n]$
  - No. of components = No. of processes
- Semantic interpretation:
  - c_i is a count of events in process *i* that causally precede **e**

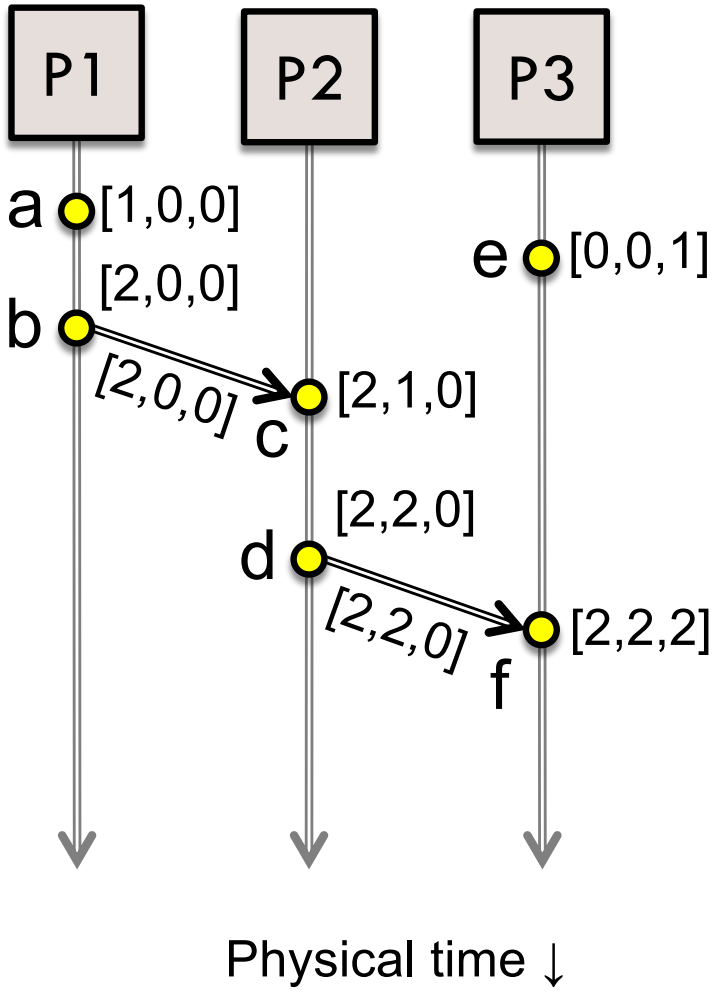
Updating vector clocks

- Initially, all vectors are [0, 0, …, 0]
- Two rules for updating

1. For each **local event** on process *i*, increment local entry c_i
2. If process *j* **receives** message with vector $[d_1, d_2, \ldots, d_n]$:
   - Set each local entry $c_k = \max\{c_k, d_k\}$
   - Increment local entry c_j

Vector clock: Example

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



Vector clocks can establish causality

- Rule for comparing vector clocks:
  - $V(\mathbf{a}) = V(\mathbf{b})$ when $\mathbf{a}_k = \mathbf{b}_k$ for all k
  - $V(\mathbf{a}) < V(\mathbf{b})$ when $\mathbf{a}_k \leq \mathbf{b}_k$ for all k and $V(\mathbf{a}) \neq V(\mathbf{b})$

- How to tell if two events are concurrent?
  - $\boldsymbol{a} \mid\mid \boldsymbol{b}$ if $\mathbf{a}_i < \mathbf{b}_i$ and $\mathbf{a}_j > \mathbf{b}_j$, some i, j

- $V(a) < V(z)$ **if and only if** there is a chain of events linked by $\rightarrow$ between a and z