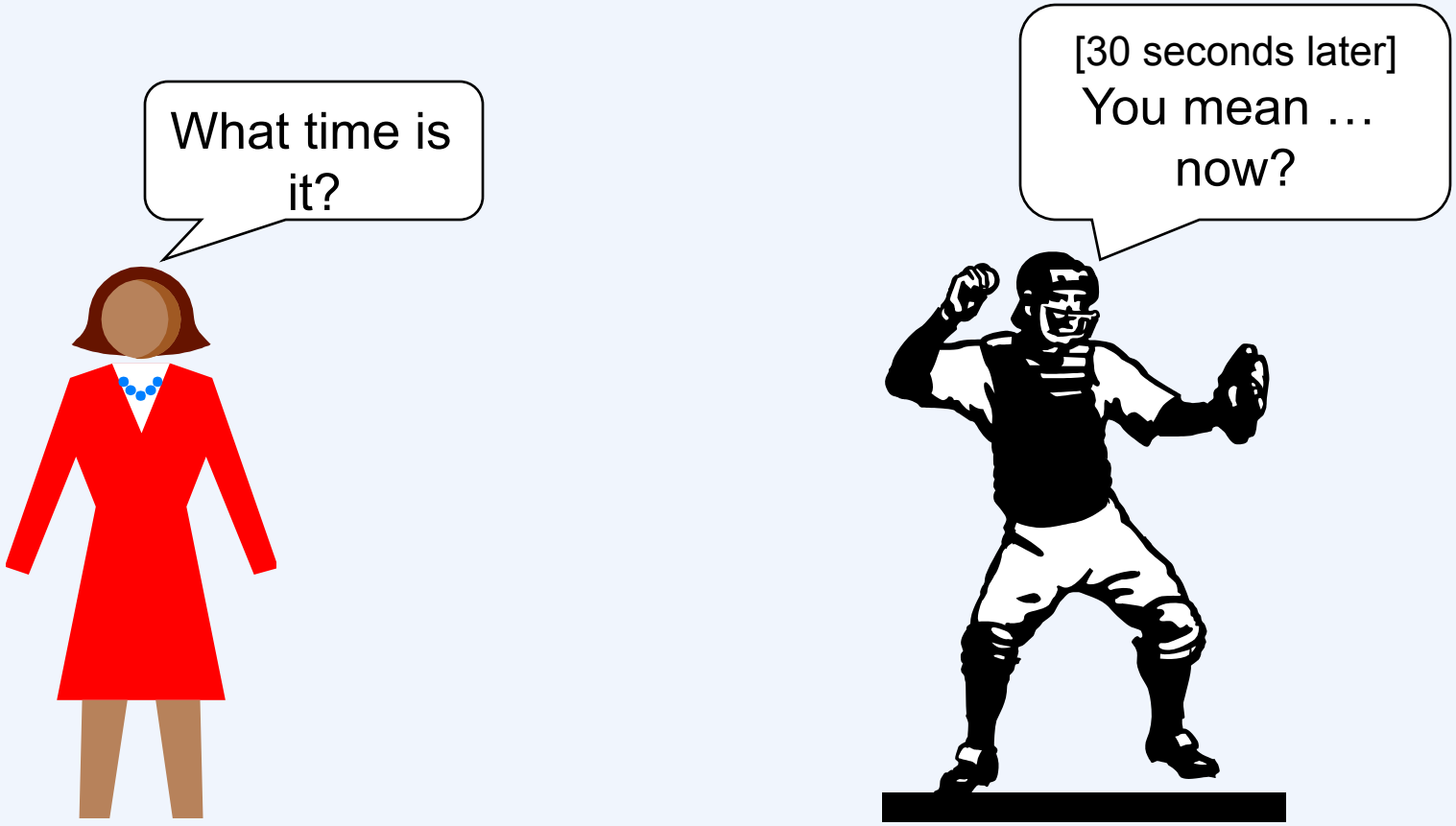
CS 138: Time

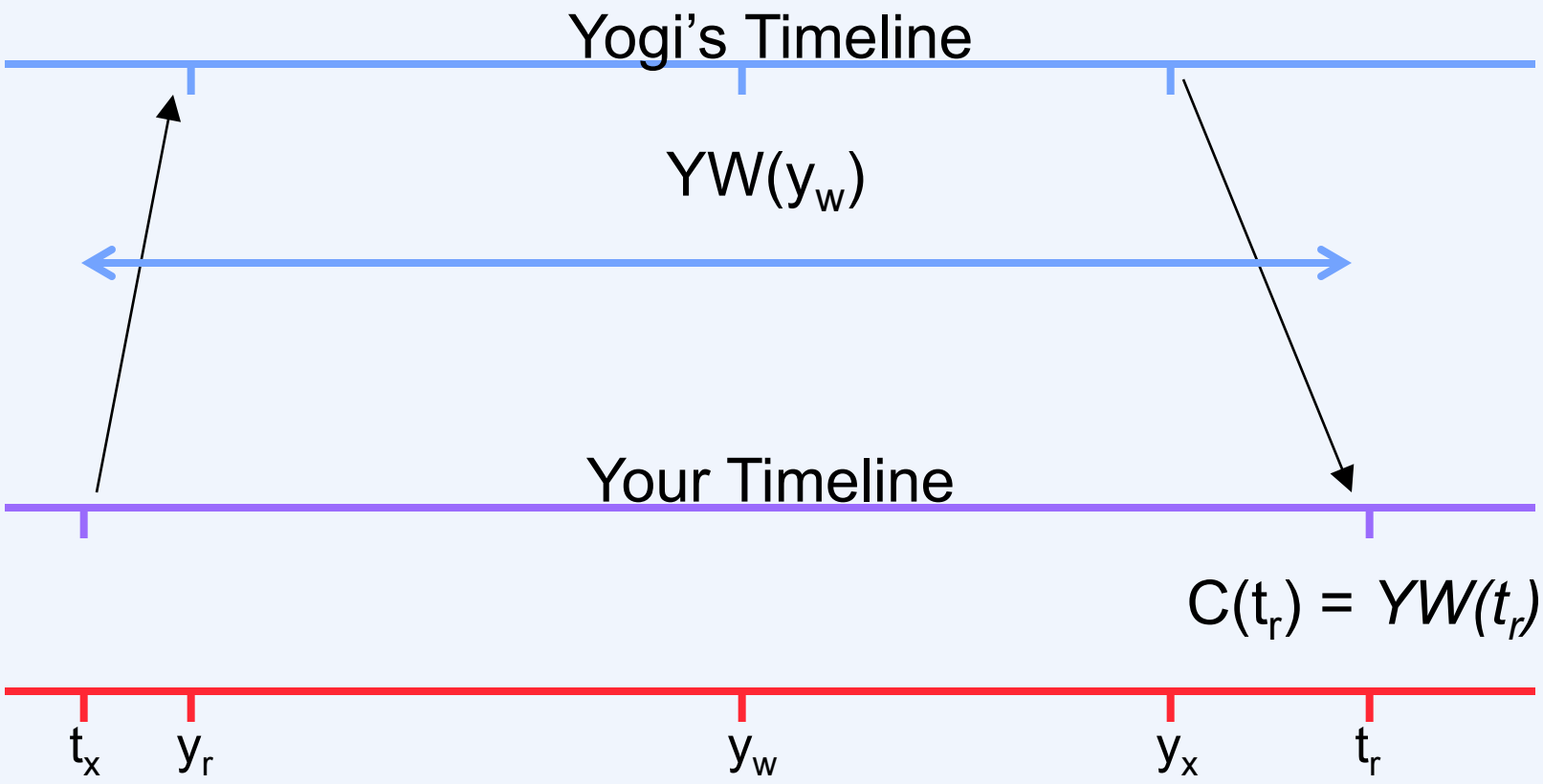
Topics

- Clock Synchronization
- Logical Clocks
- Causality
- Vector Clocks

Getting the Time



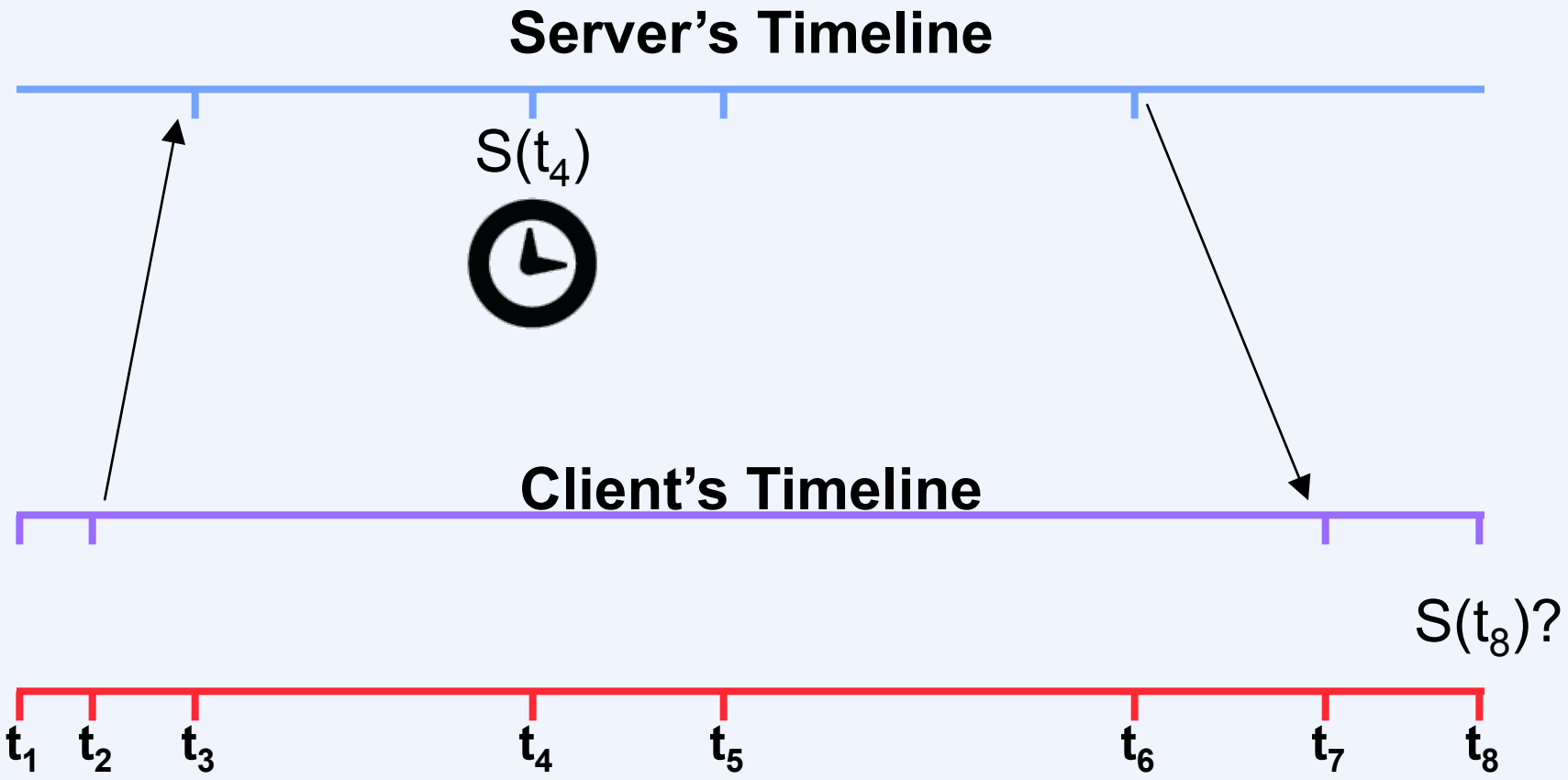
Analysis



$t_x <= YW(y_w) <= t_r$

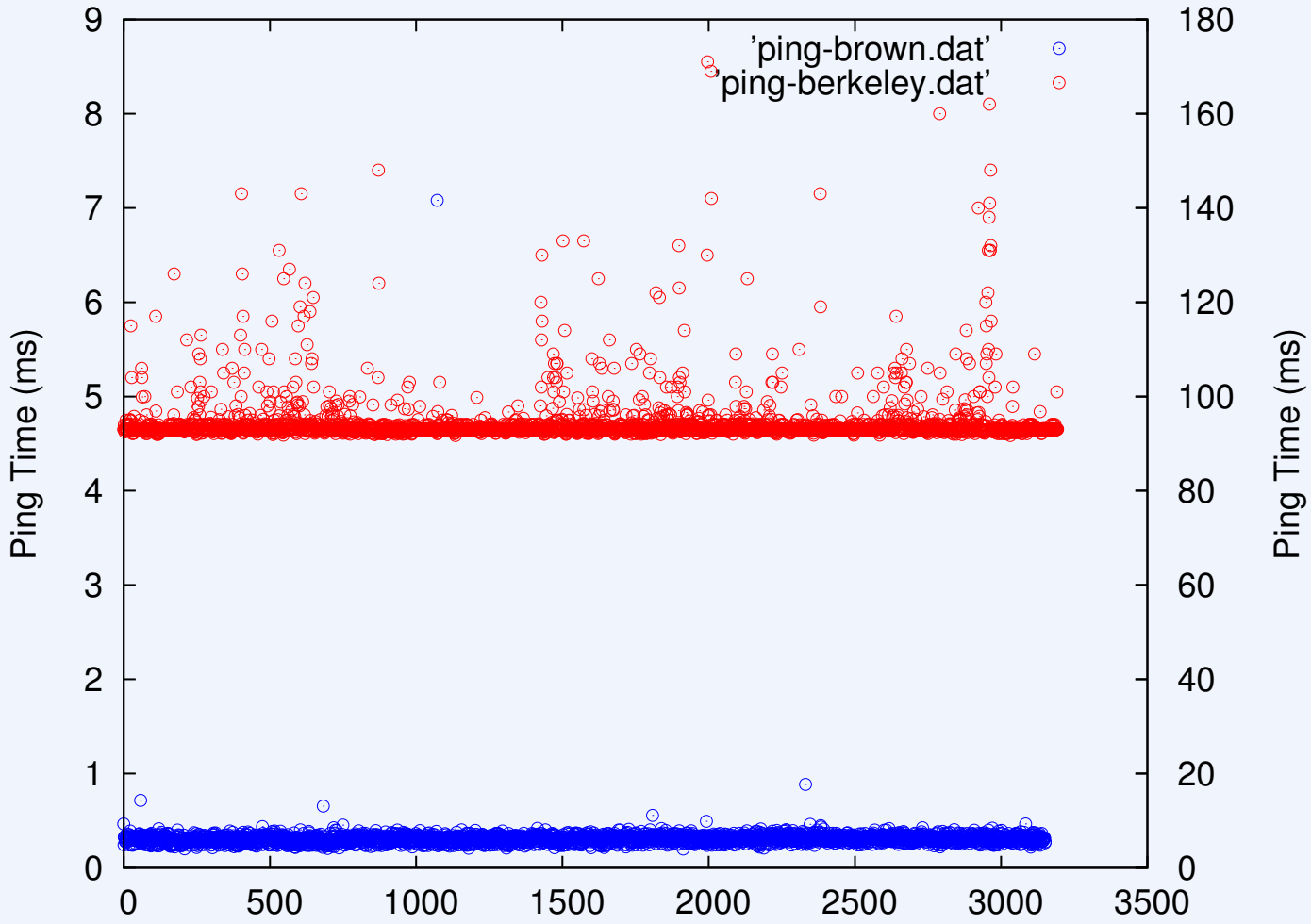
$YW(t_r) = [YW(y_w), YW(y_w) + (t_r - t_x)]$

Using a Time Server

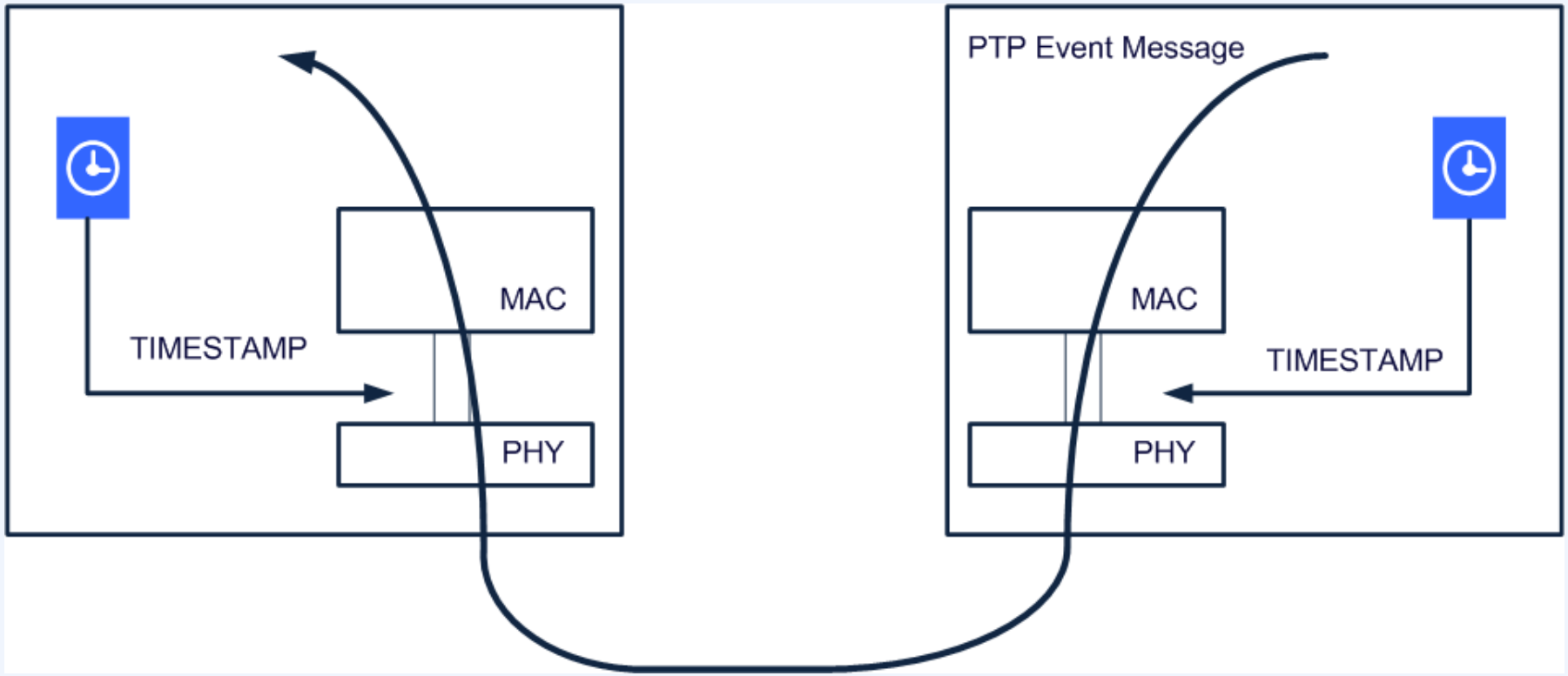


- $S(t_8)$ in $[S(t_4), S(t_4) + (C(t_8) - C(t_1))(1 + r)(1+d)]$

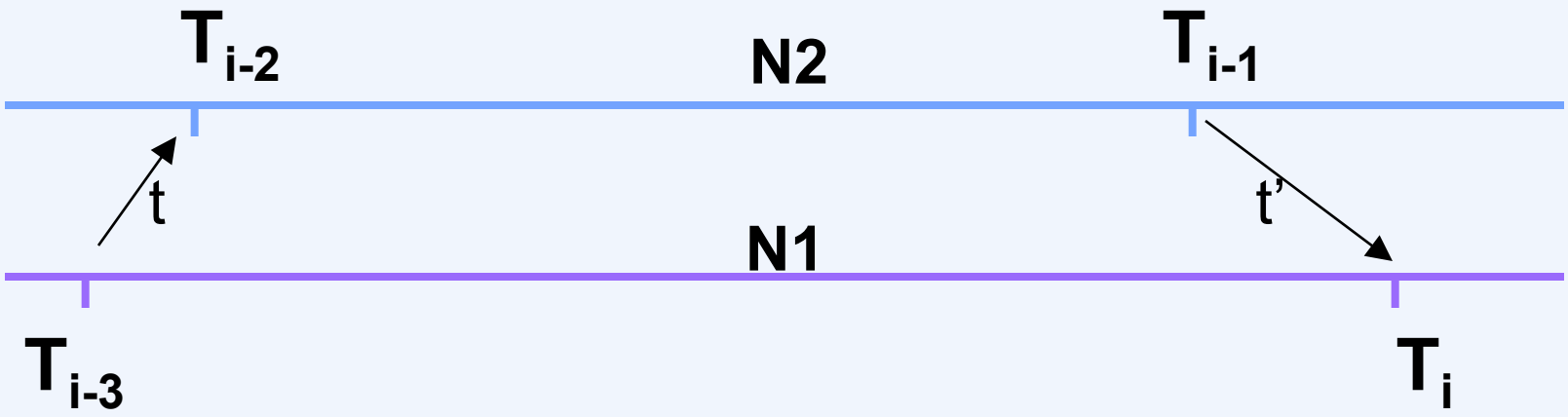
Variability in Ping Times



Precision using HW time-stamping (PTP Protocol)



Example: NTP



Goal: calculate offset O

$a = T_{i-2}-T_{i-3}$, $b=T_{i-1}-T_i$

$d_i = (T_i-T_{i-3}) - (T_{i-1} - T_{i-2}) = a-b;$

$o_i = (a+b)/2 = (T_{i-2}+T_{i-i})/2 – (T_i+T_{i-3})/2$

$t = T_{i-2}-O -T_{i-3} = a – O >= 0 \rightarrow a >= O$

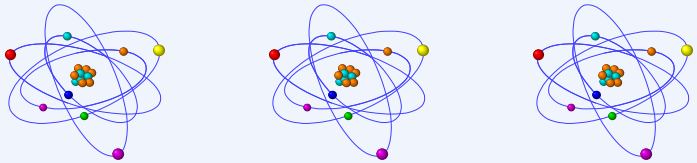
$t' = T_i-(T_{i-1}-O) = O – b >= 0 \rightarrow b <= O$

$b = (a+b)/2 – (a-b)/2 <= O <= (a+b)/2+(a-b)/2 = a$

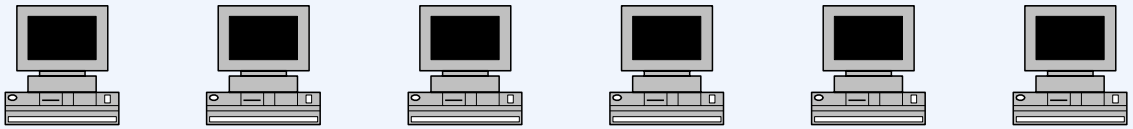
$o_i- d_i/2 <= O <= o_i + d_i/2$

Network Time Protocol

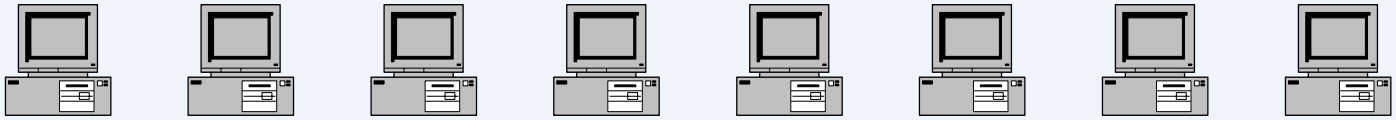
Stratum 0



Stratum 1



Stratum 2



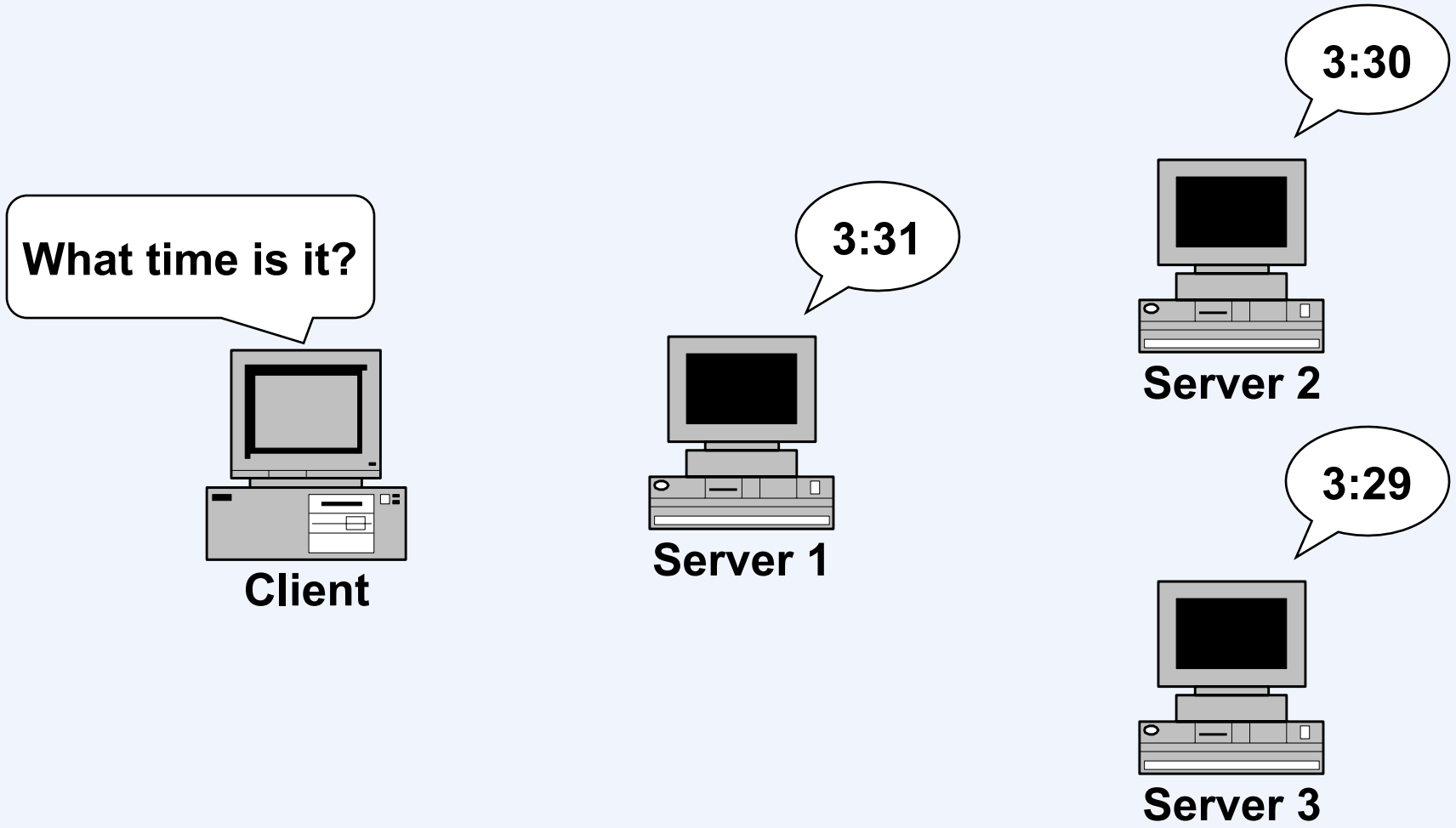
▪
▪
▪

Nobody's Perfect

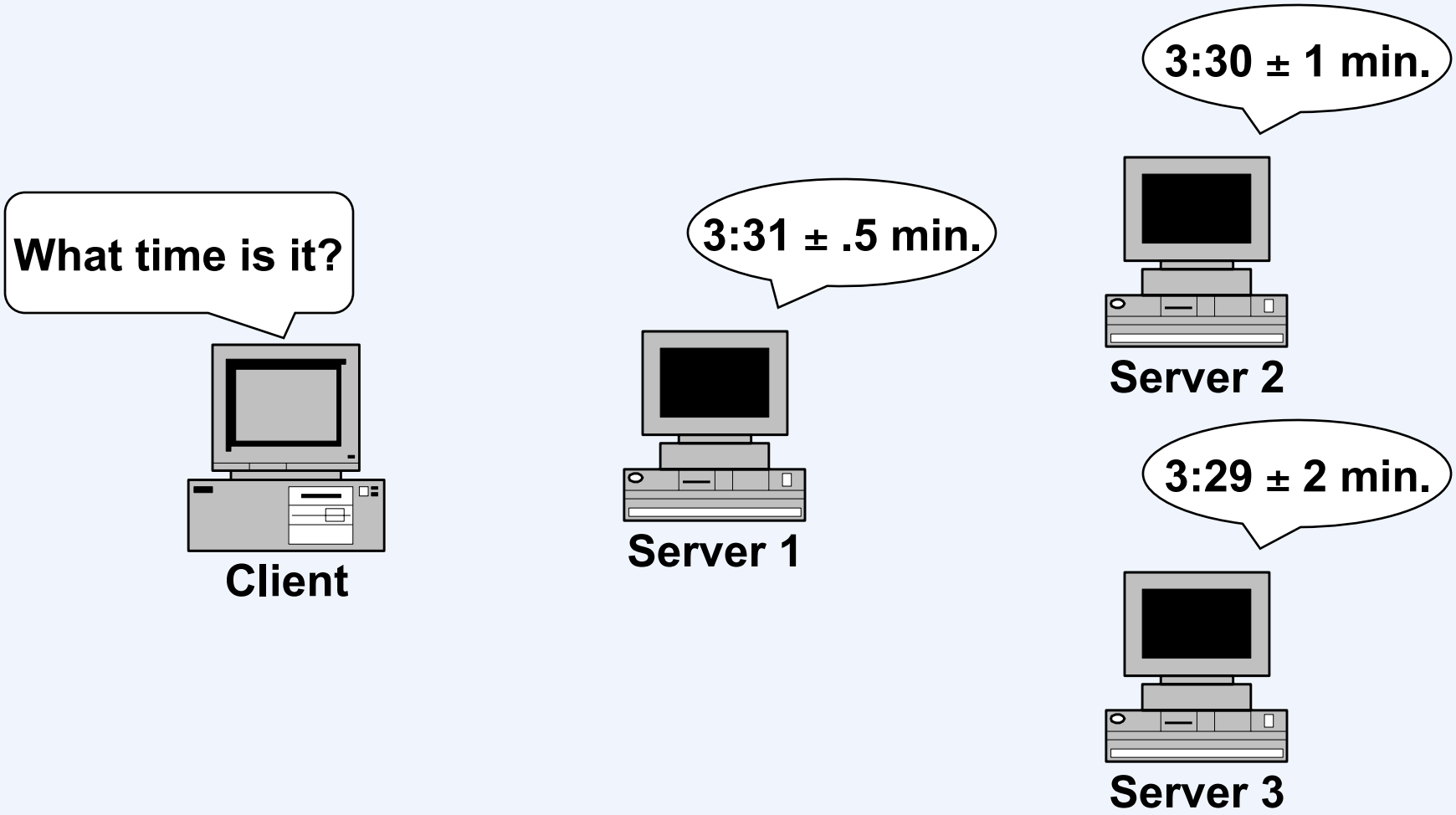
- **Inaccuracy:** $T(t) - I(t) <= t <= T(t) + I(t)$
- **Drift:** $I(t_2) <= d \cdot (t_2 - t_1) + I(t_1)$ (a bound on I)
- **Resolution: each tick represents *r* seconds**
- **Clocks are assumed always to make forward progress**

$t_2 >= t_1 + r/(1-d)$ ➔ $T(t_2) >= T(t_1) + r$

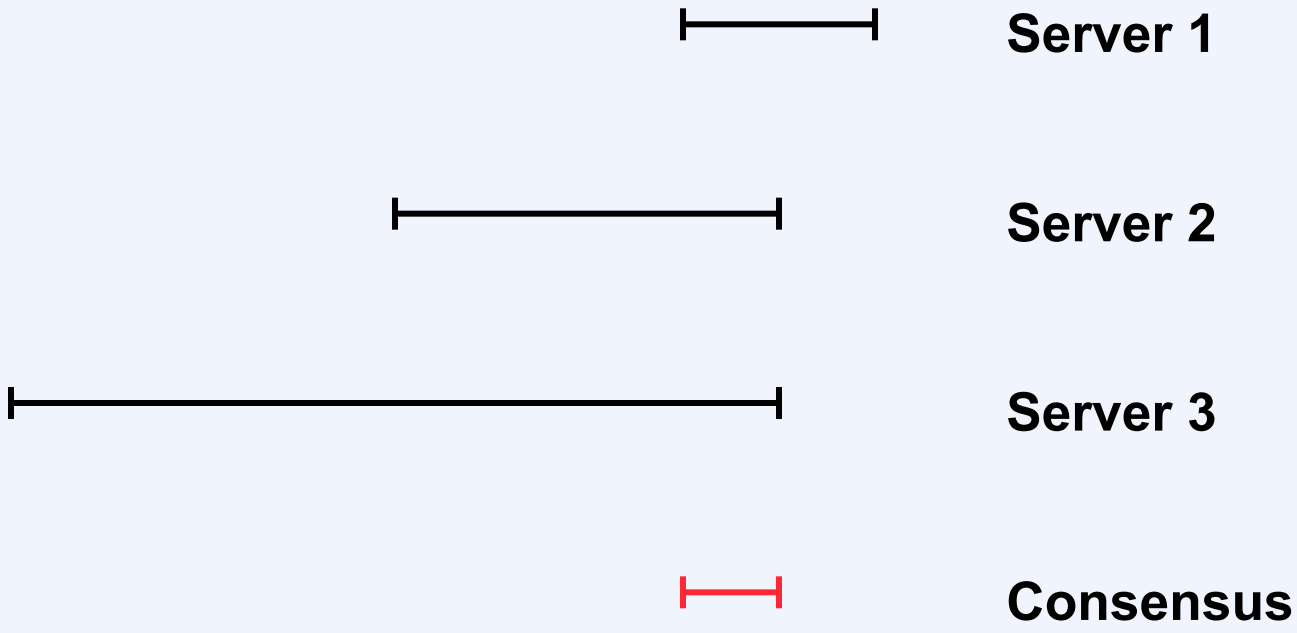
Getting Other Opinions



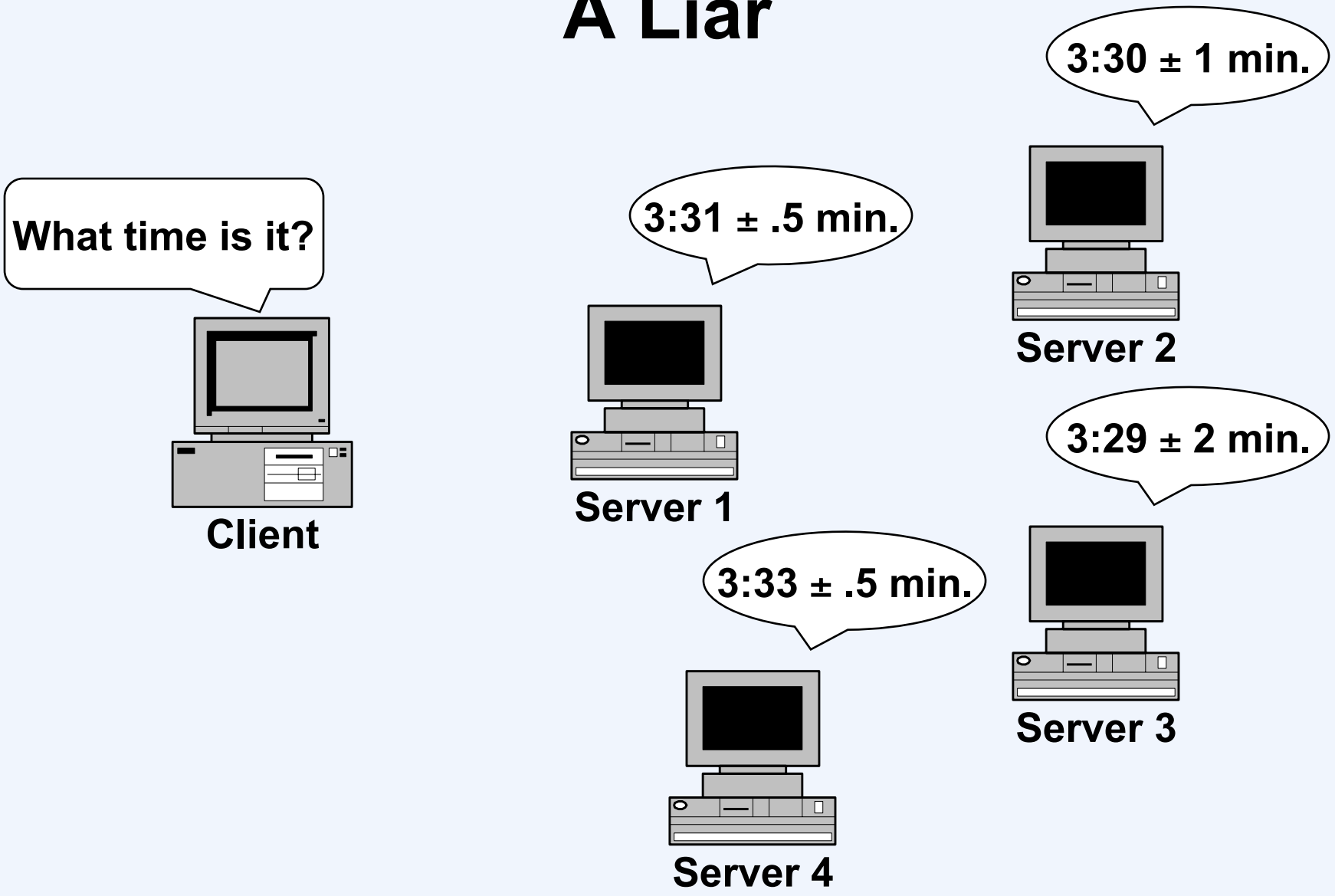
Truth in Advertising



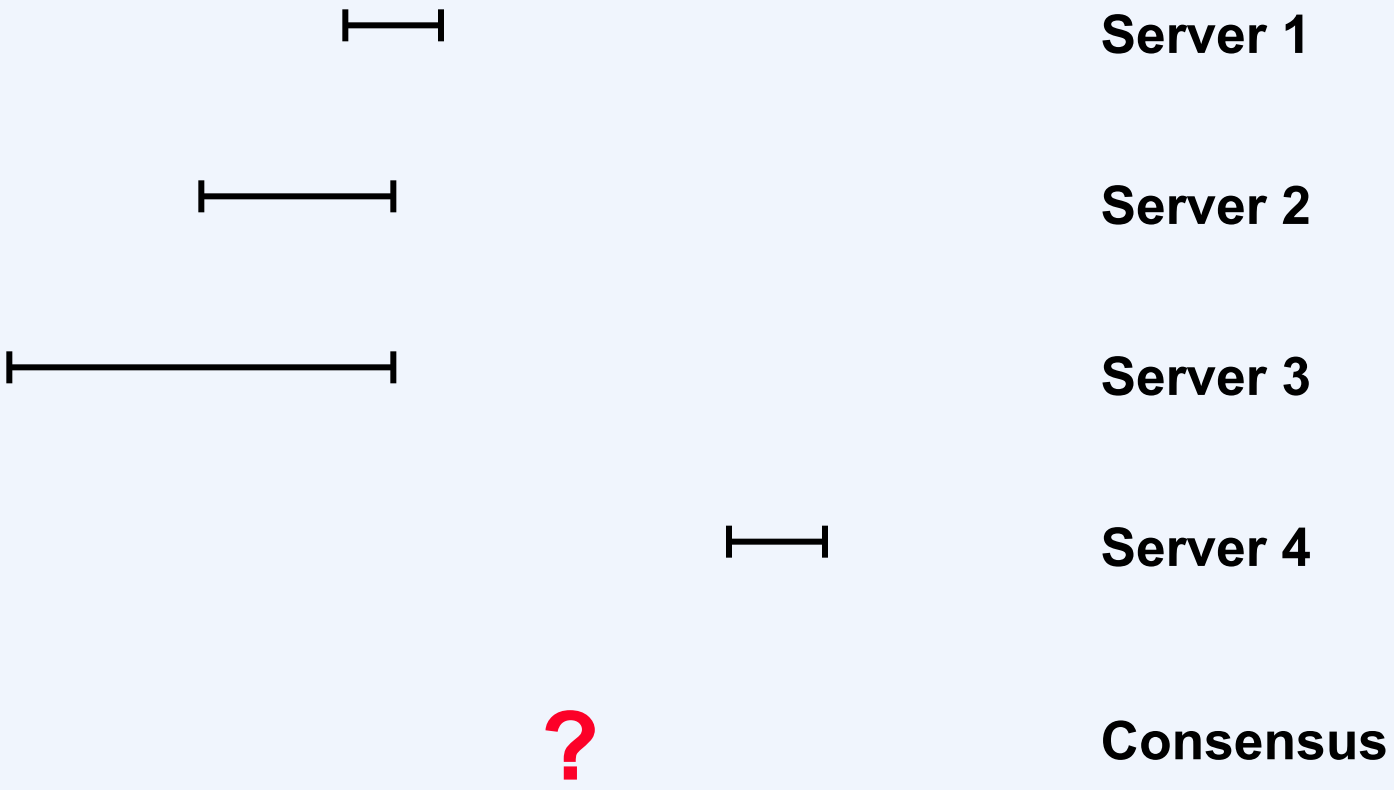
Arriving At a Consensus



A Liar

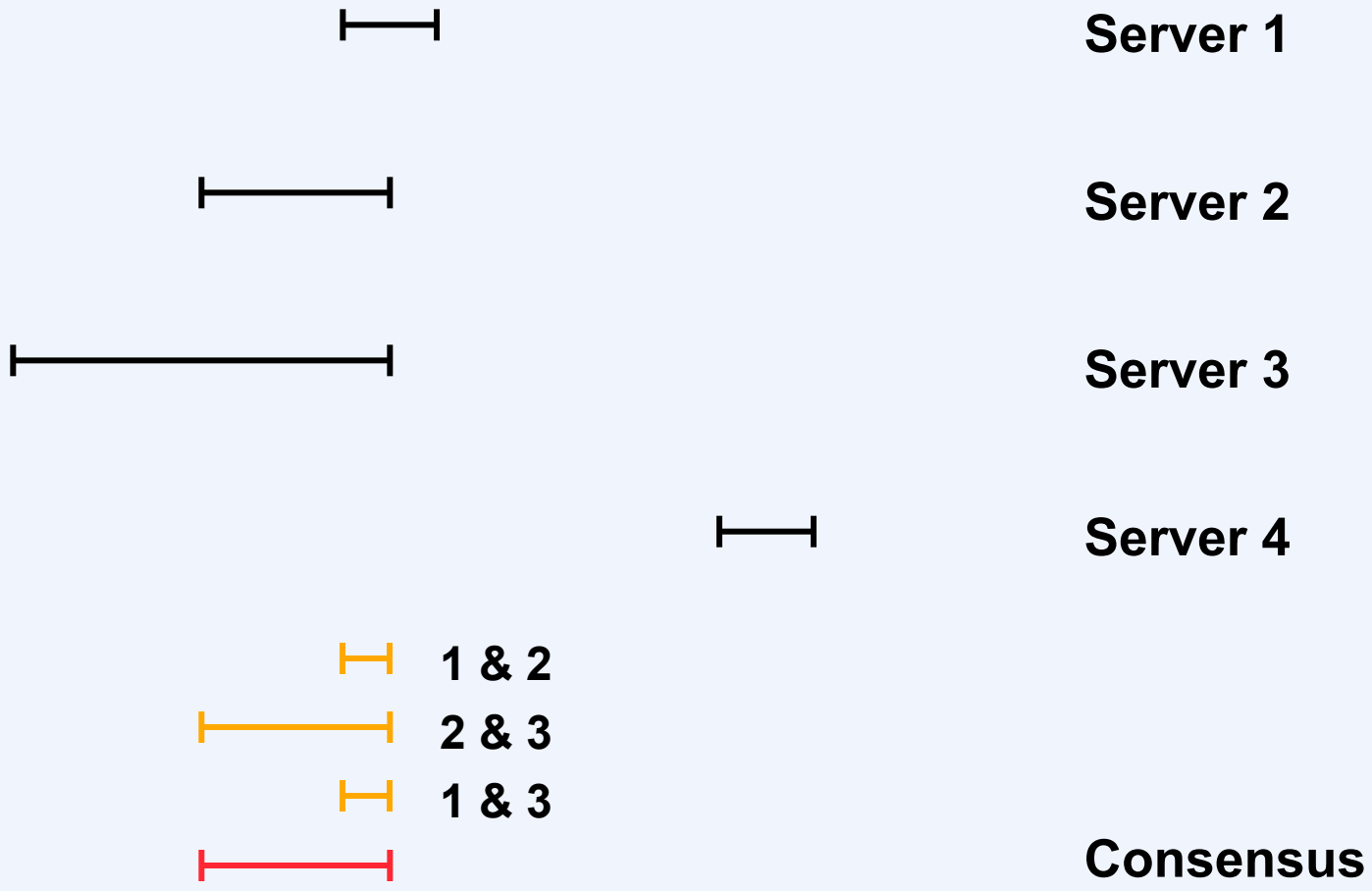


What To Do?

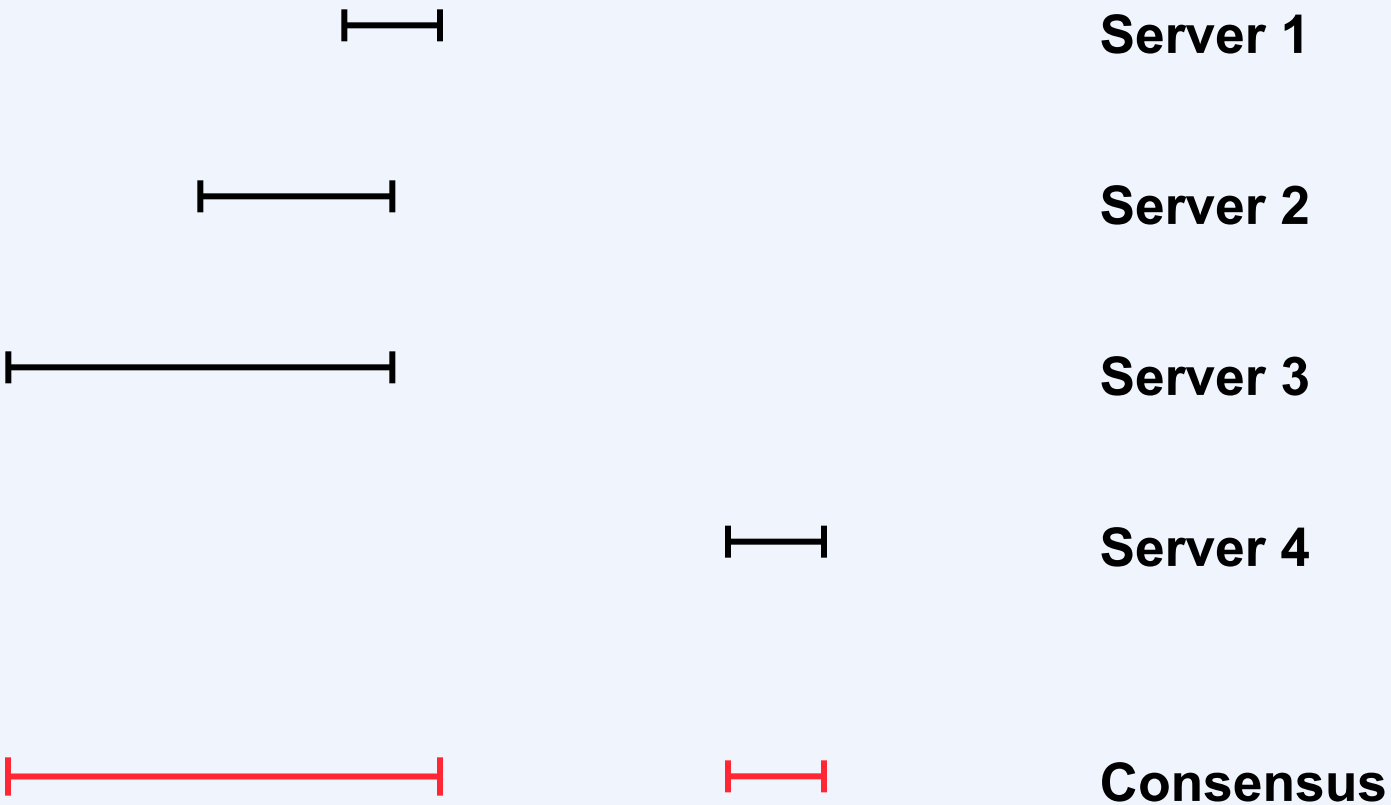


At Most One Liar

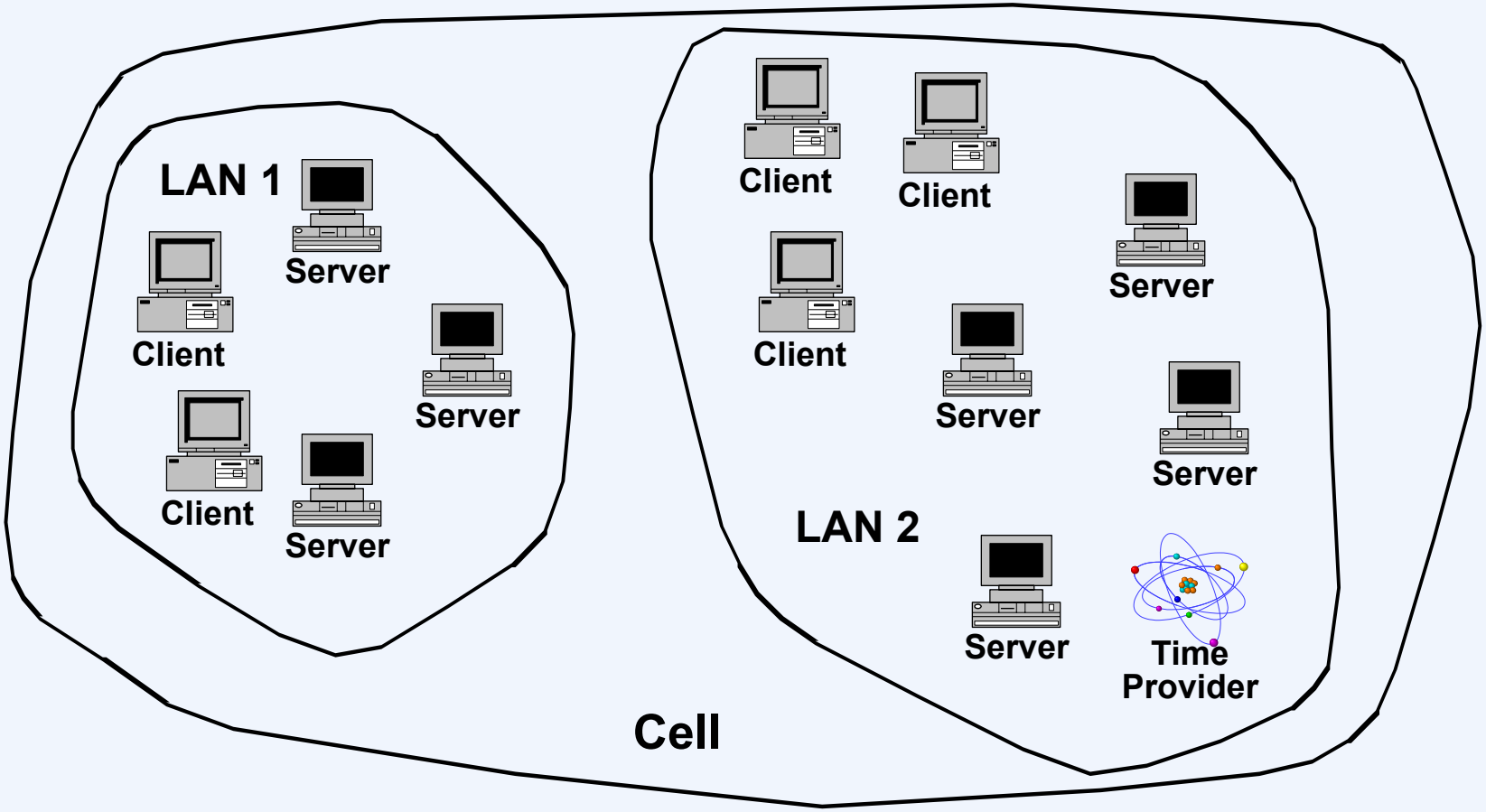
At Most Two Liars



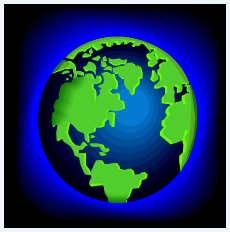
At Most Three Liars



DCE Time Service



What's Time?



- GMT
  - determined by astronomical observations at Greenwich
  - improved versions, accounting for irregularities of earth's rotation and orbit, are UT0, UT1, UT2
- International Atomic time (TAI)
  - based on transitions of energy levels of cesium atom
  - synchronized with earth time in 1958
  - earth has been running slow since then
    - UTC: leap seconds added as necessary to adjust

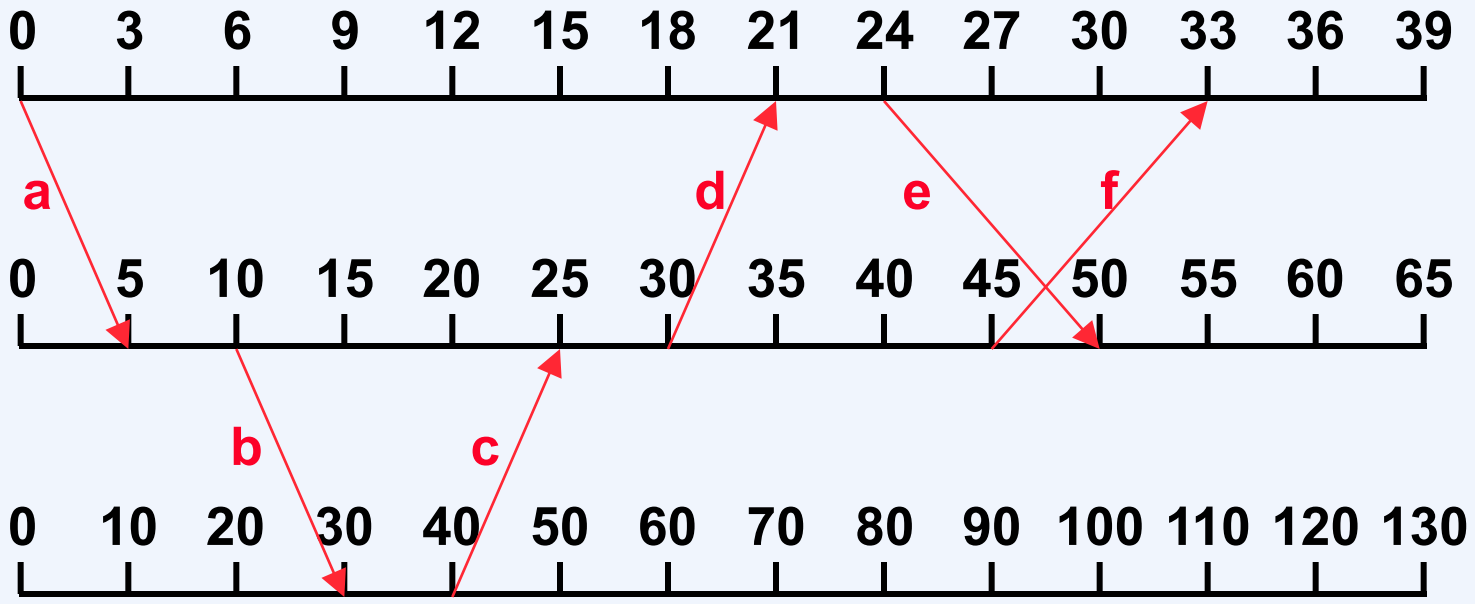
What's Time? (Really …)

- "Time is God's way of keeping everything from happening at once."

  –various sources

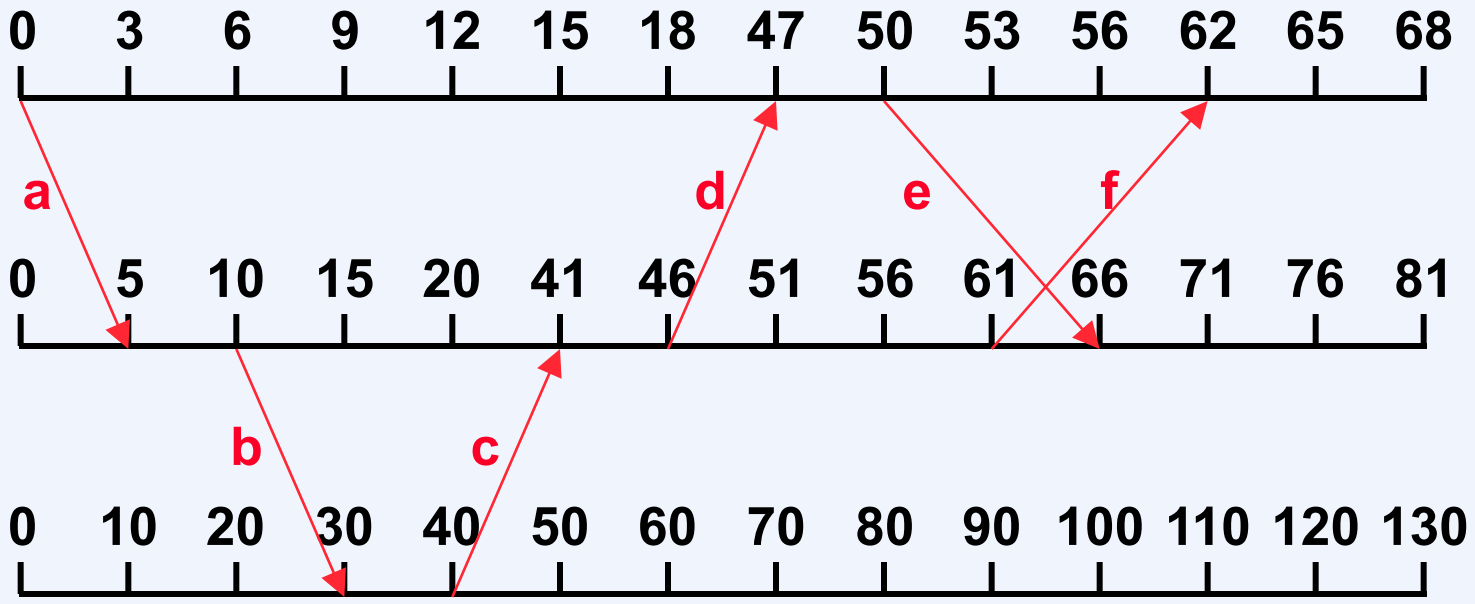
- Timestamp events to determine order of their occurrence
- Problems
  - doesn't work if clocks either aren't perfectly in sync or don't have sufficiently fine resolution
  - an order might not be meaningful if two events have no causal relationship
- A → B iff A could possibly have had an effect on B (pronounced A *happened before* B)

Logical Clocks (1)



Invariant: if x h.b. y, T(x)<T(y)

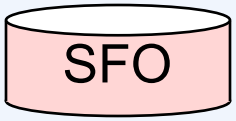
Logical Clocks (2)



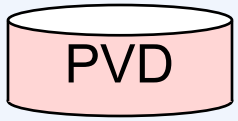
Using Logical Timestamps

- We can use Lamport Clocks to create a total order of events agreed to by all processes

Distributed Banking



add interest based
on current balance



deposit $1000

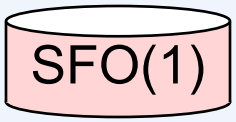
Total Order

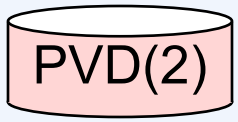
- Tie-breaking rule
  - what if $T_i(a) = T_h(b)$?
  - *a* comes before *b* iff $i<h$
- Total order for all events in a distributed system

Totally Ordered Multicast

- To send multicast:
  - tag message with sender's timestamp (<time, sender ID>)
    - sender receives own multicast
- On receipt of message
  - queue message in timestamp order
  - multicast an acknowledgement
- On receipt of acknowledgement
  - link to acknowledged message
- Deliver message to application when
  - message is at front of receive queue
  - has been acknowledged by all

Totally Ordered Multicast





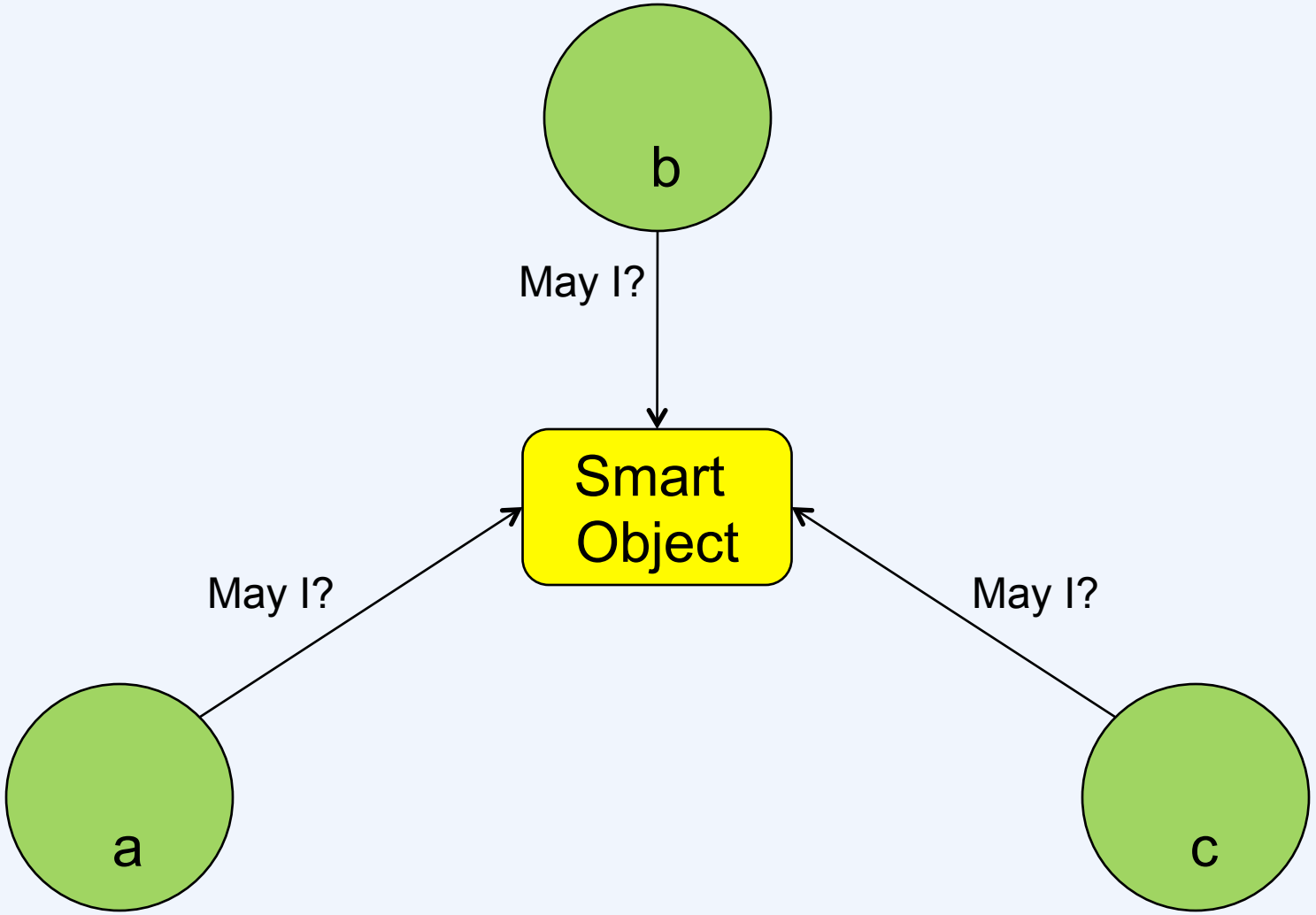
In	Out	Out	In
	1: compute interest (1,1)	1: deposit $1000 (1,2)	
2: compute interest (1,1)			2 : deposit $1000 (1,2)
	3: ack interest-1	3 : ack deposit-2	
4: ack interest-1			4: ack deposit-2
5: deposit $1000 (1,2)			5: compute interest (1,1)
	6: ack deposit-1	6: ack interest-2	
7: ack deposit-1			7: ack interest-2
8: ack deposit-2			8: ack interest-1
9: ack interest-2			9: ack deposit-1

PVD must reorder queue once all acks are in

Mutual Exclusion

- Central server
- Logical clocks

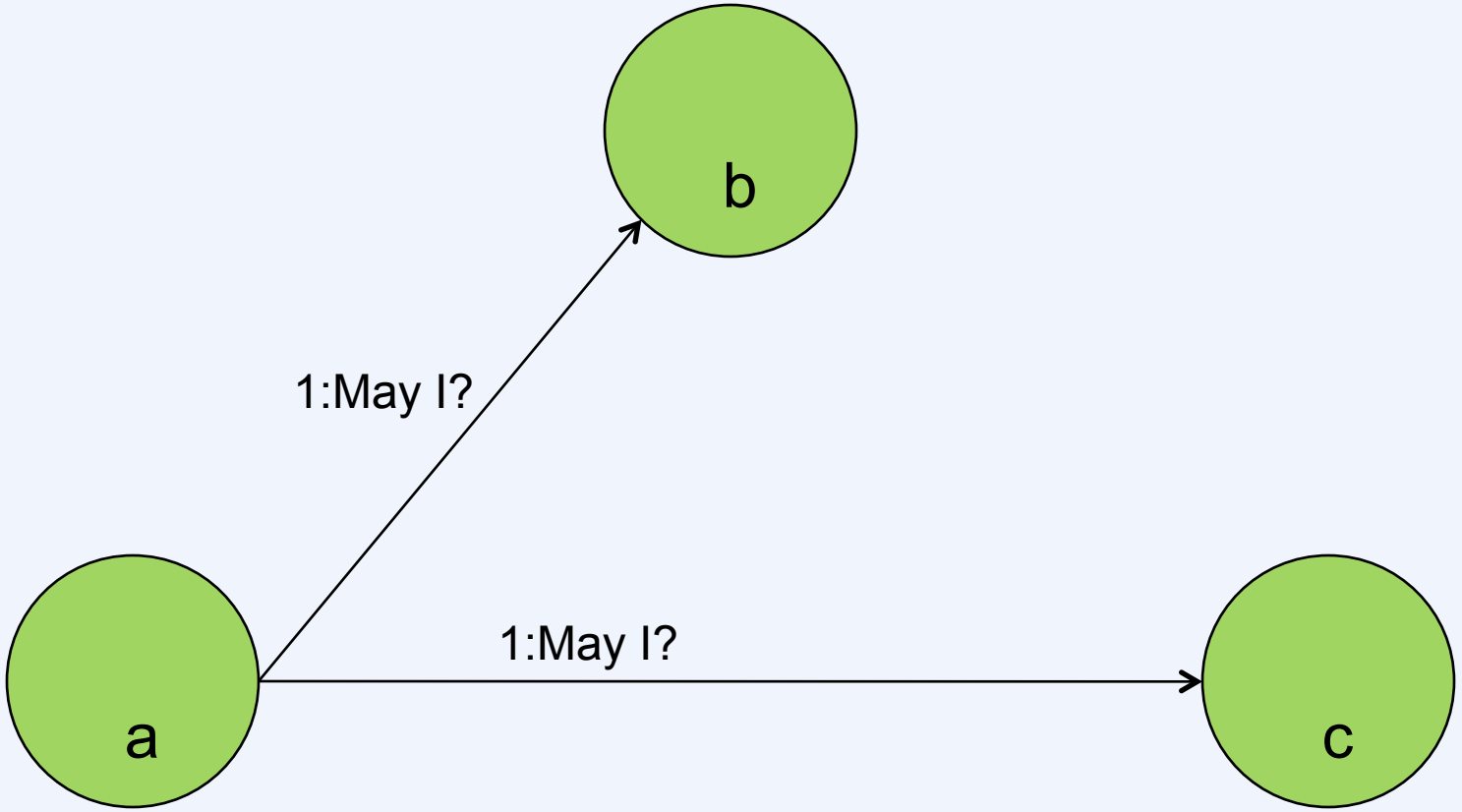
Central-Server Mutual Exclusion



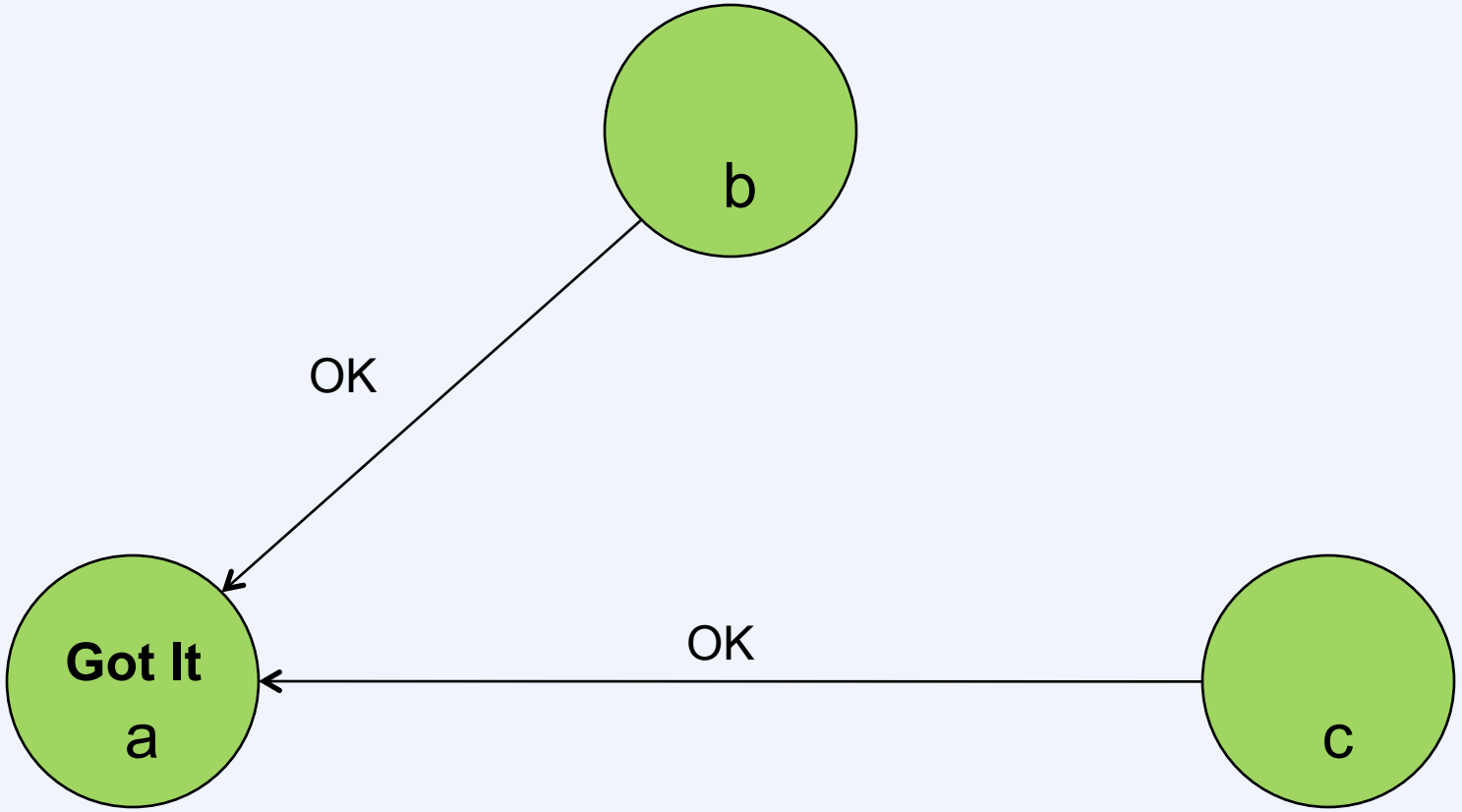
Mutual Exclusion with Logical Clocks

- Requester
  - multicast request with timestamp
  - proceed when all other parties respond OK
- Receiver of request
  - if neither using nor waiting for resource, respond OK
  - if waiting for resource, respond OK if request's *timestamp is lower than own*, otherwise queue request
  - if using resource, queue request
- When finished
  - respond OK to queued requests

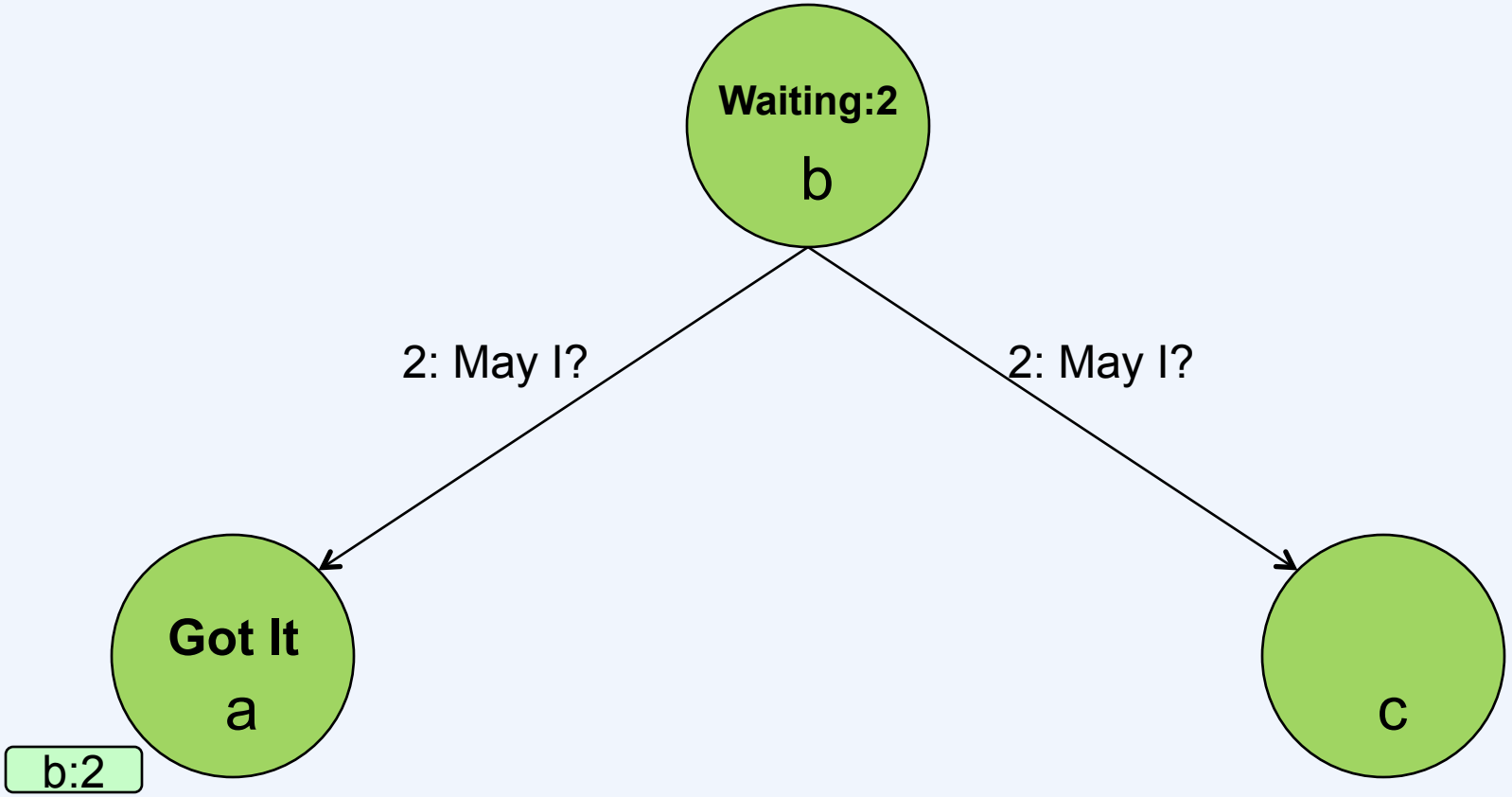
Mutex Exclusion (1)



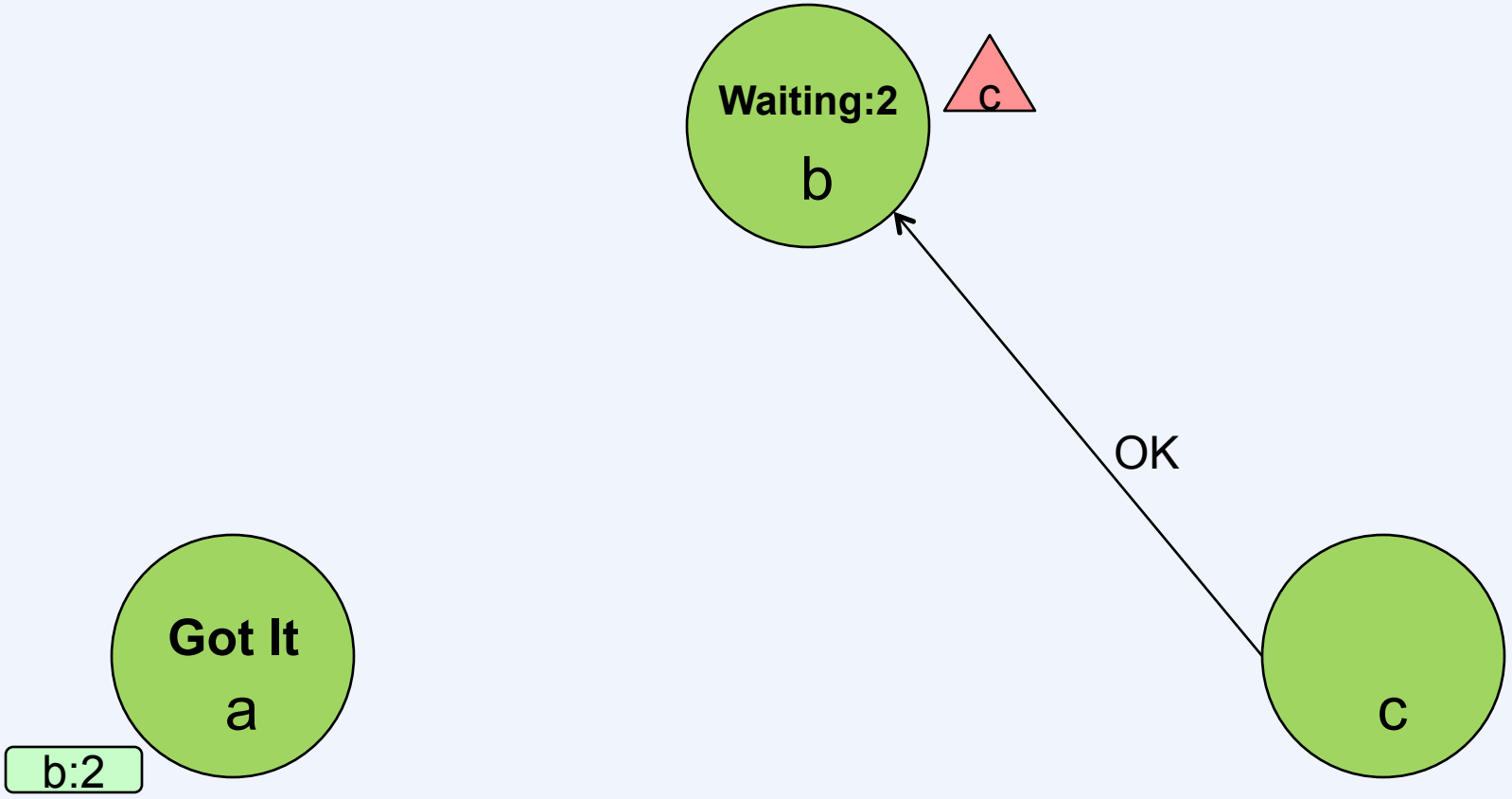
Mutex Exclusion (2)



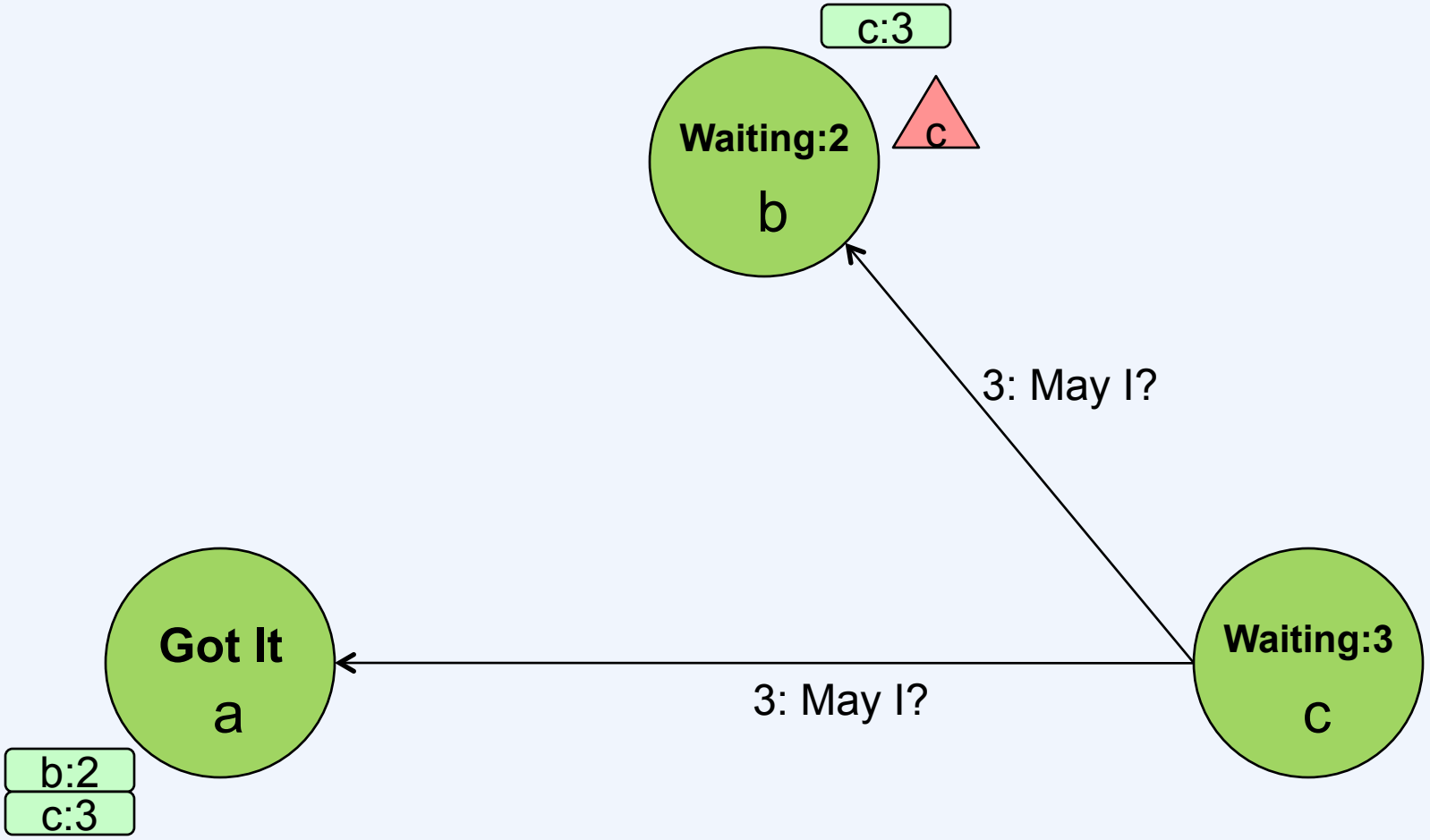
Mutex Exclusion (3)



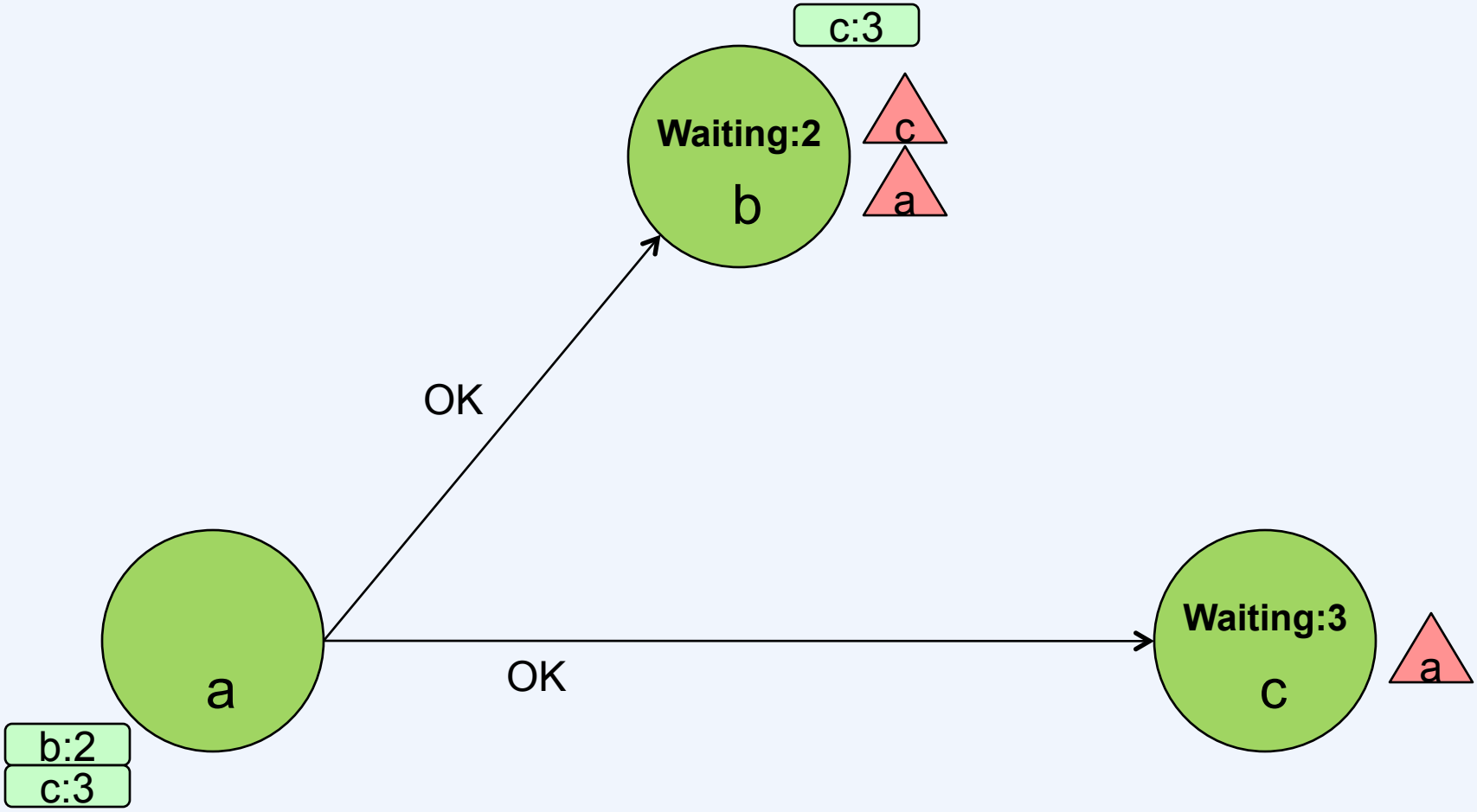
Mutex Exclusion (4)



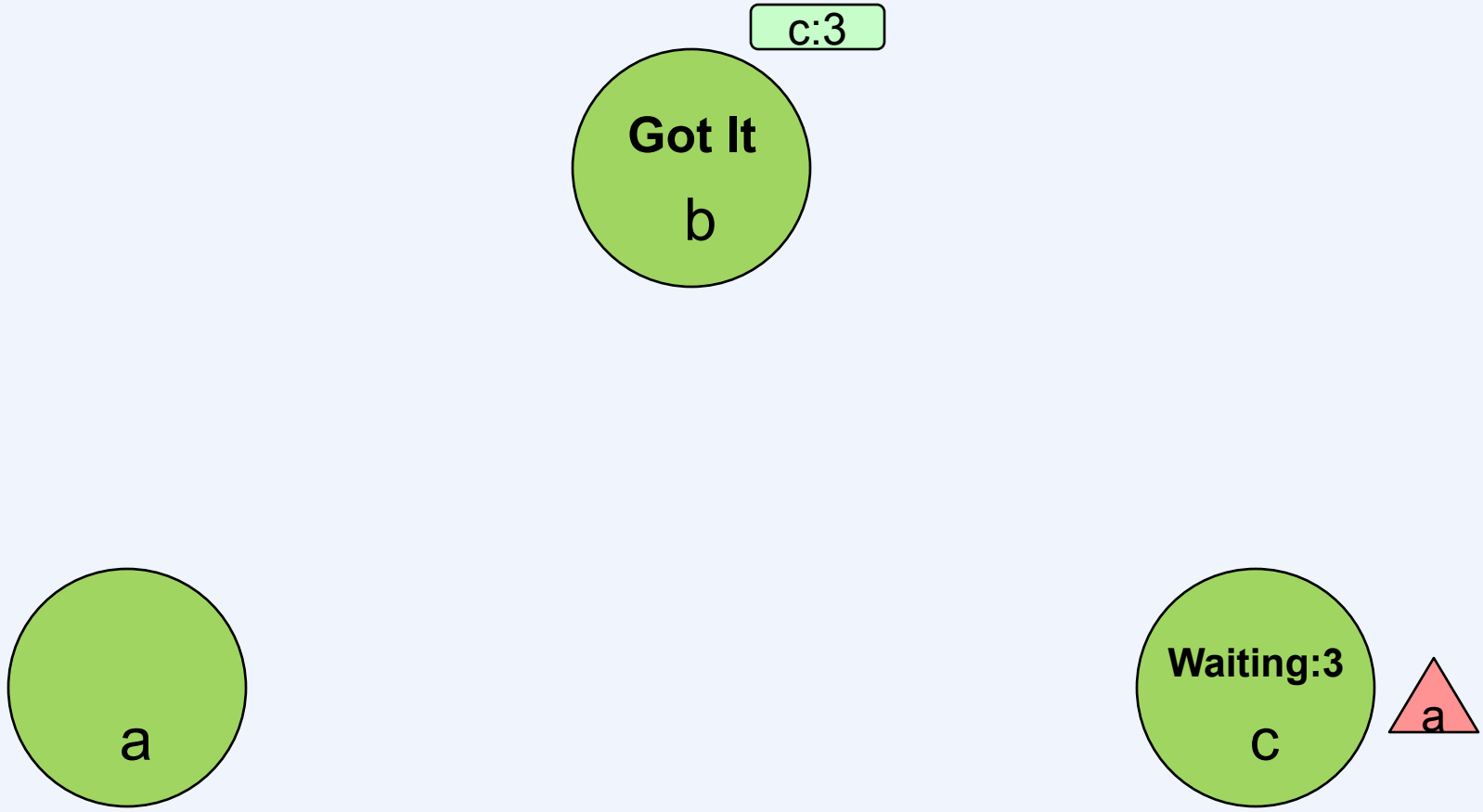
Mutex Exclusion (5)



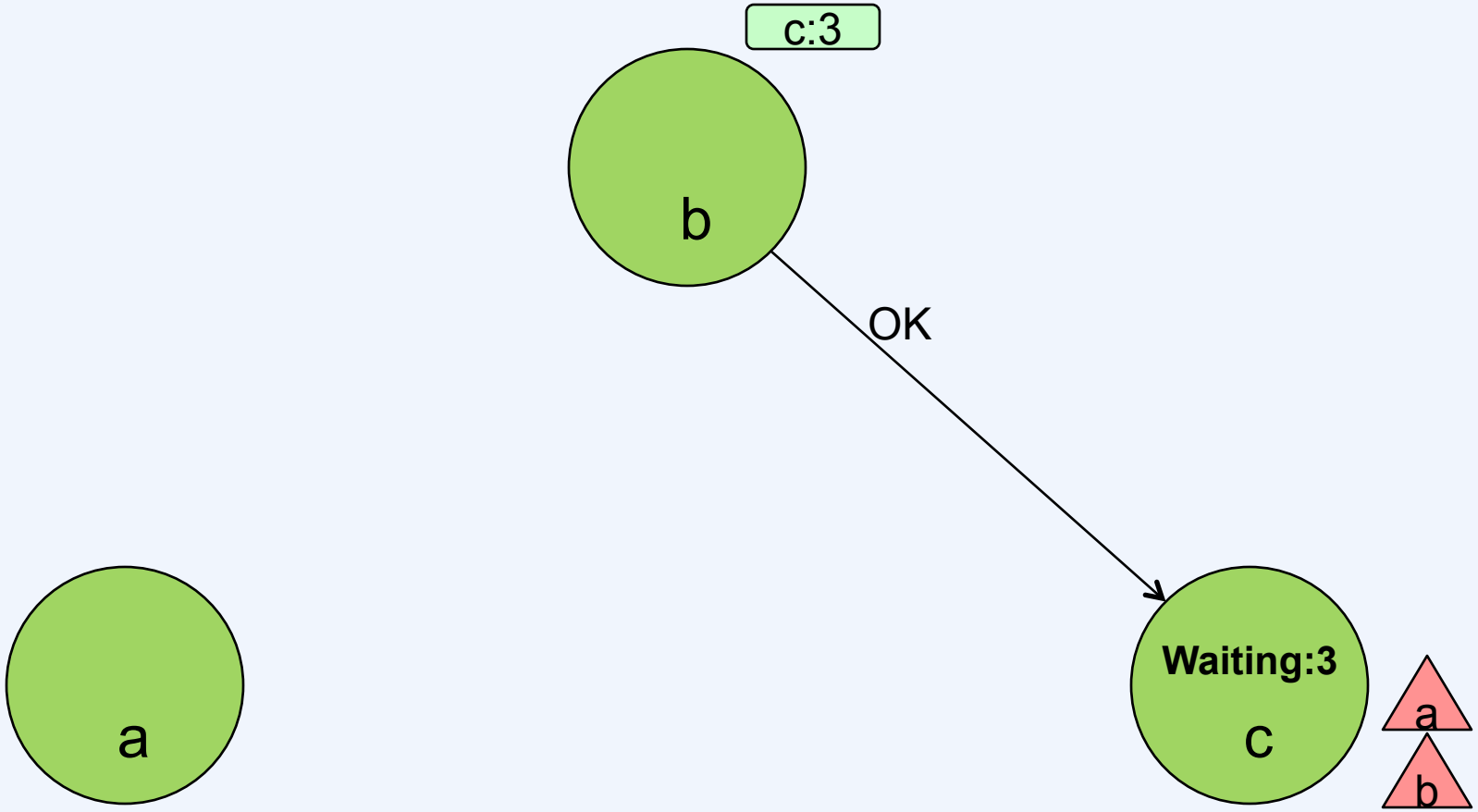
Mutex Exclusion (6)



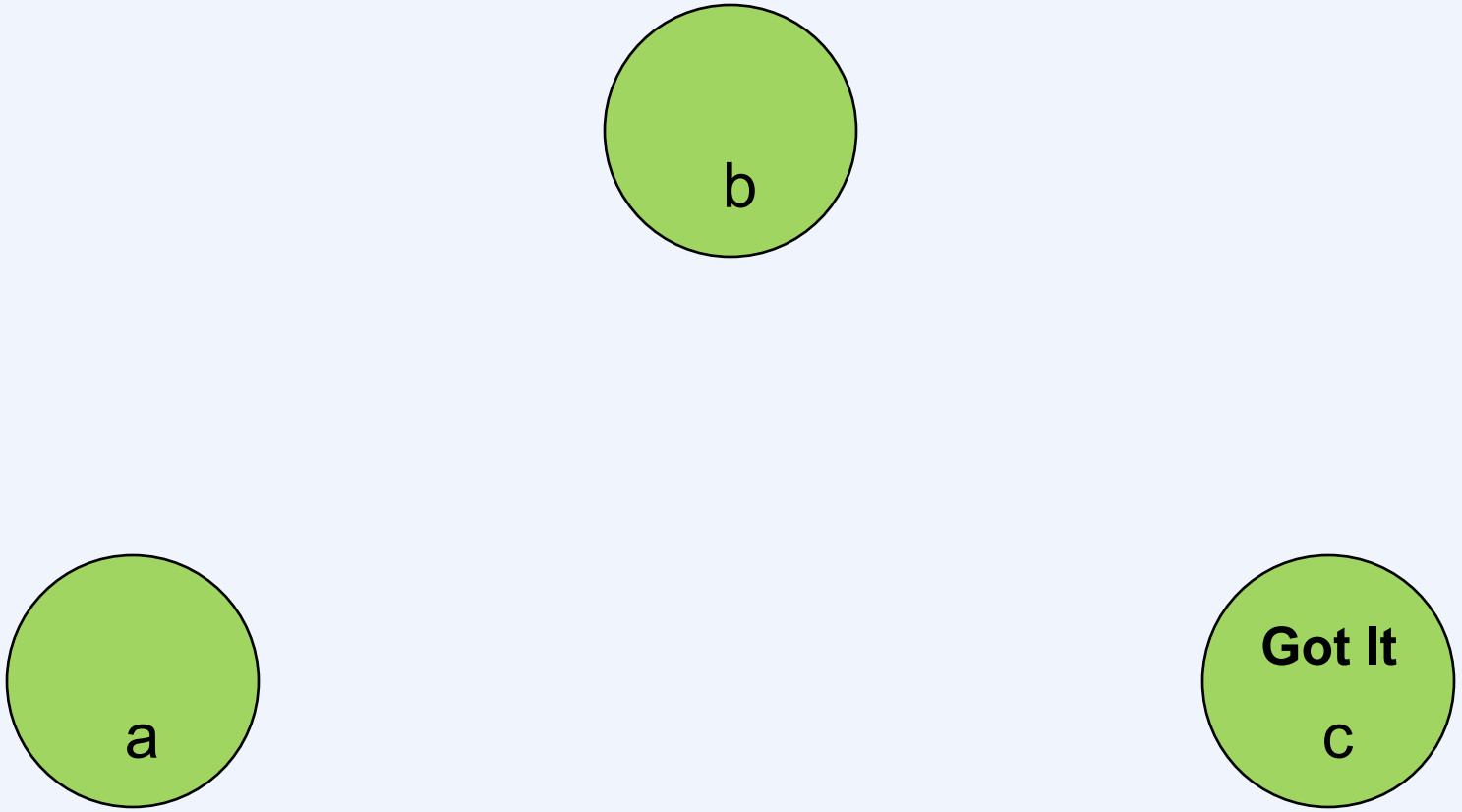
Mutex Exclusion (7)



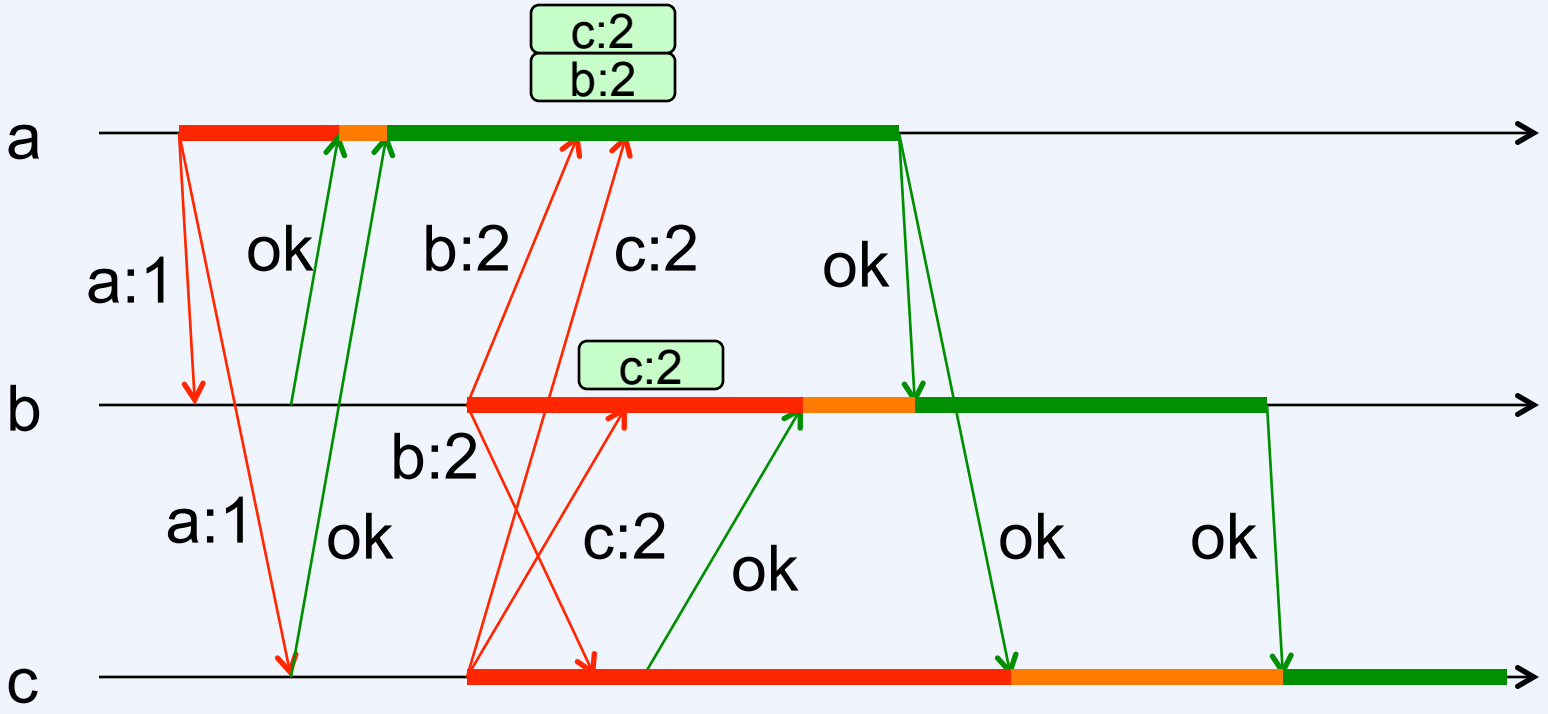
Mutex Exclusion (8)



Mutex Exclusion (9)



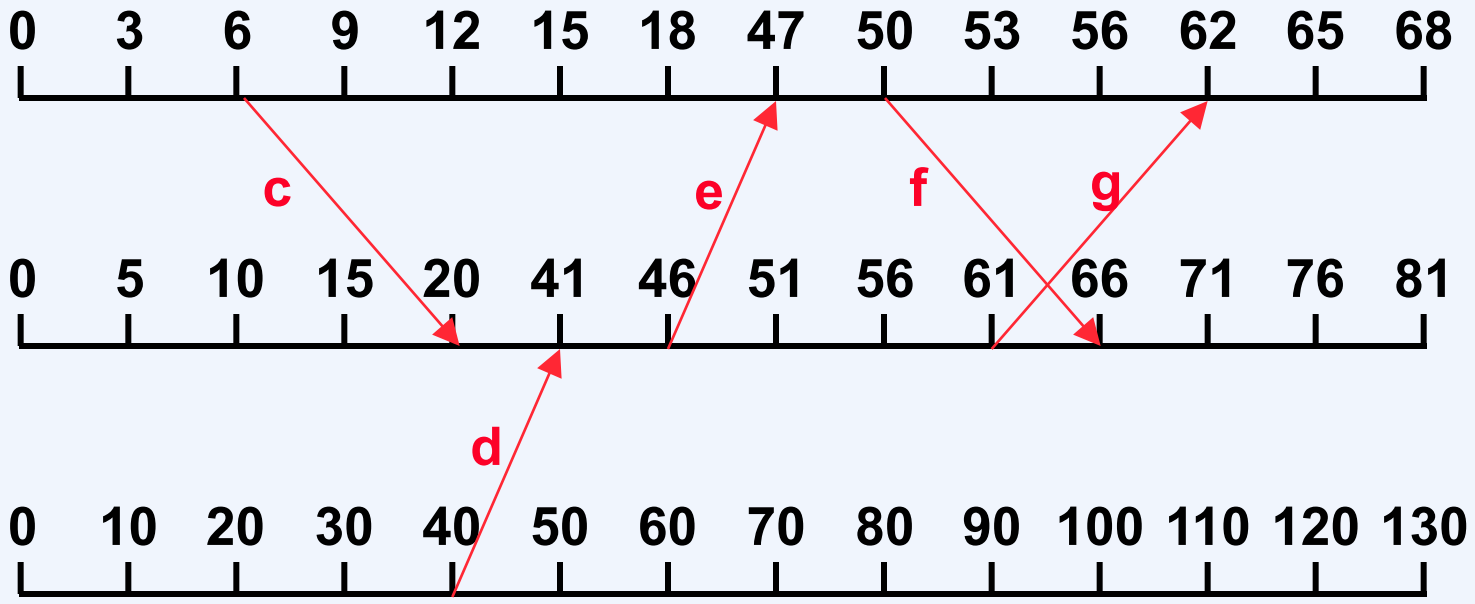
Why Total Order is Important



"if waiting for resource, respond OK if request's timestamp is lower than own, otherwise queue request"

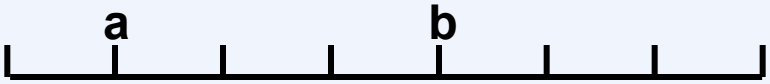
b:2 < c:2

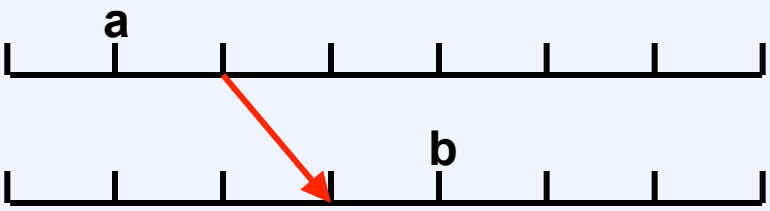
Causal Ordering



Causality

- How can event *a* have a causal effect on *b*?





Concurrent Events

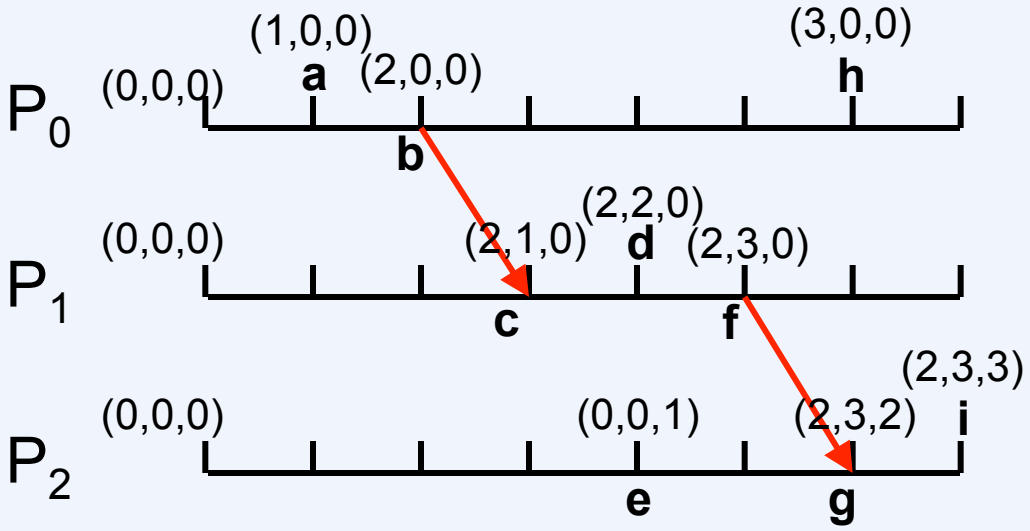
- **Events a and b are *concurrent* if neither of the following are true:**

$$a \rightarrow b$$

$$b \rightarrow a$$

Vector Clocks

- $VC_i[i]$
  - number of events so far at process P_i
- $VC_i[h] = k$
  - process P_i is aware of the first k events at P_h



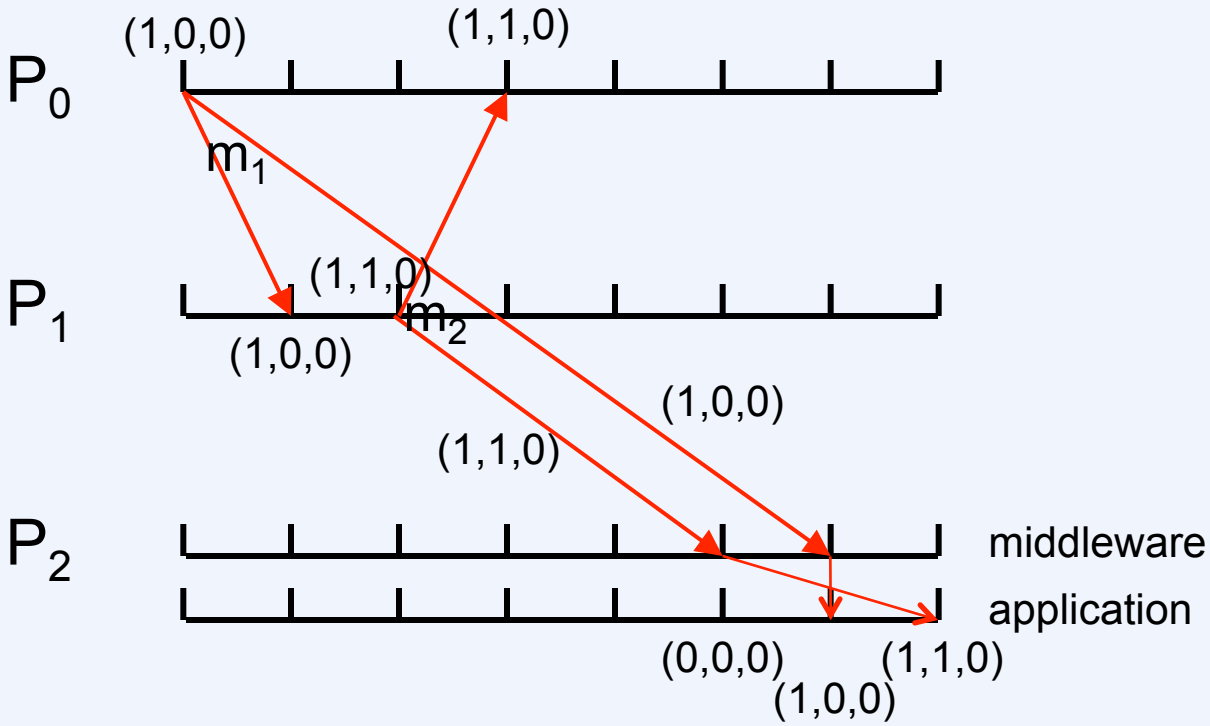
Rules …

1) Initially, $VC_i[j] = 0$, for i, j = 0, … , n-1
2) Just before P_i timestamps an event, it sets $VC_i[i] = VC_i[i]+1$
3) P_i includes the current timestamp in every message it sends
4) When P_i receives a timestamp t in a message, it sets $VC_i[j] = \max(VC_i[j], t[j])$, for j = 0, … , n-1
5) Comparing clocks:
    1) If all components of $VC_i <= VC_j$ and at least one component is smaller ⇔ i happens before j
    2) i and j are concurrent iff neither $VC_i <= VC_j$ and $VC_j <= VC_i$

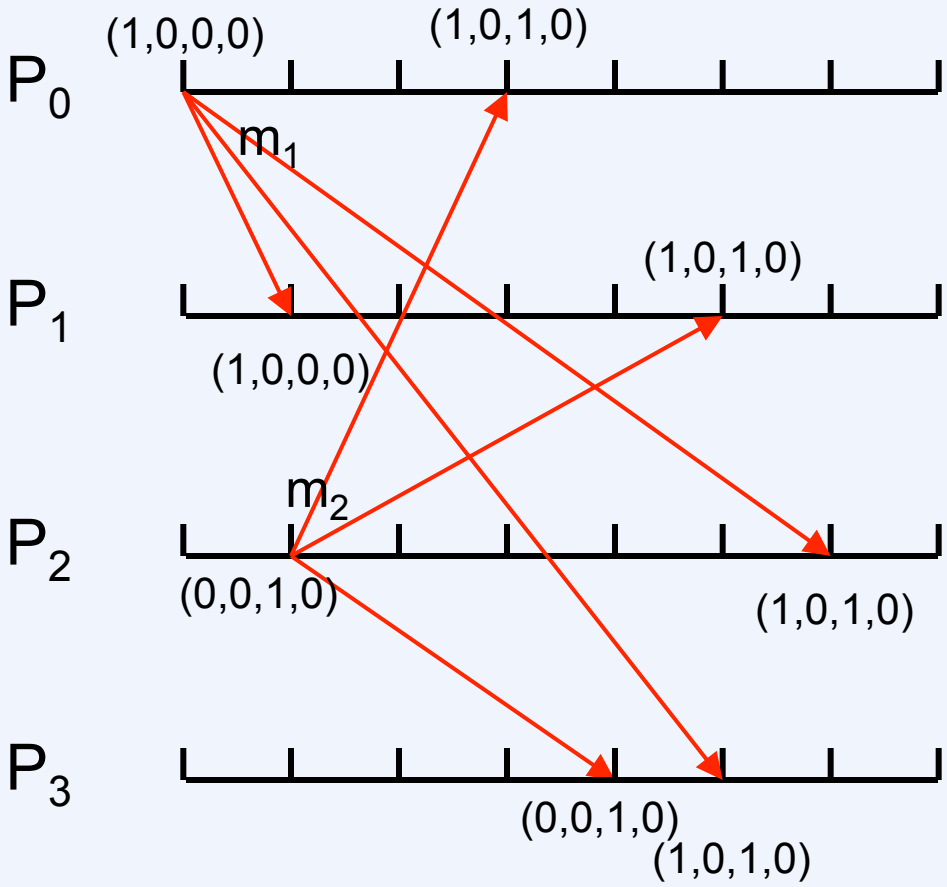
Causally Ordered Multicast

- Application of vector clocks
  - the only event is 'sending message'
  - all messages are multicast to all
  - messages carry "timestamp" = sender's VC
- Strategy
  - P_h receives multicast message *m* from P_i
  - deliver *m* to application when:
    - timestamp(*m*)[i] = VC_h[i] + 1
      - next expected message from P_i
    - timestamp(*m*)[k] ≤ VC_h[k], for all k≠i
      - P_h has seen all events P_i had seen when it sent the message

Causally Ordered Multicast (1)



Causally Ordered Multicast (2)



Not covered in class

- These won't be in the exams, but are another example of when you can use timestamps

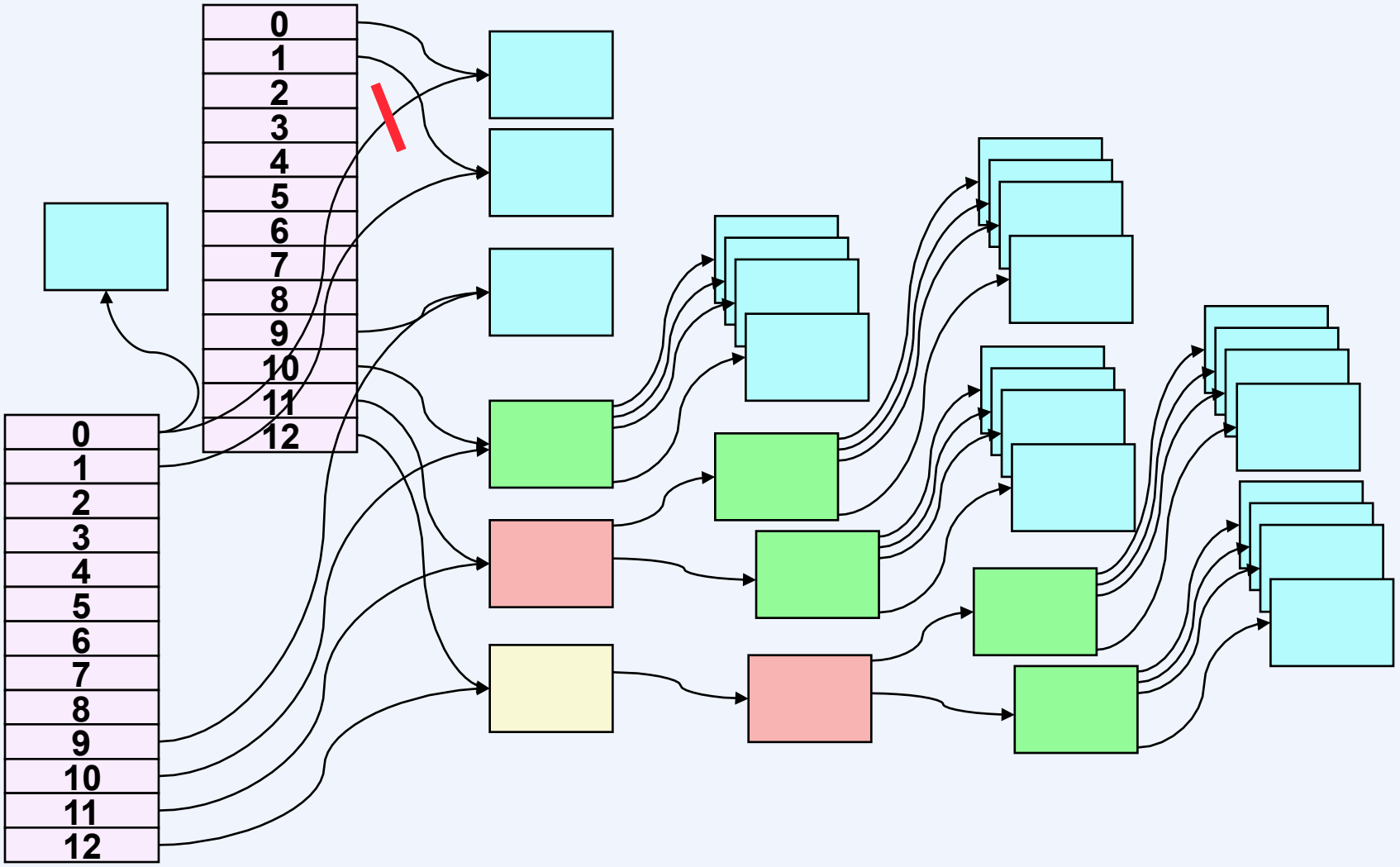
File Transactions

- Open file
- Write first item
  - allocate space
  - update inode
  - write data
- Write second item
  - allocate space
  - update inode
  - write data
- Close file

Example Strategies

- Shadow inodes
- Logging
  - Ext3, NTFS
- Optimism

Shadow Inodes



Optimistic Concurrency Control

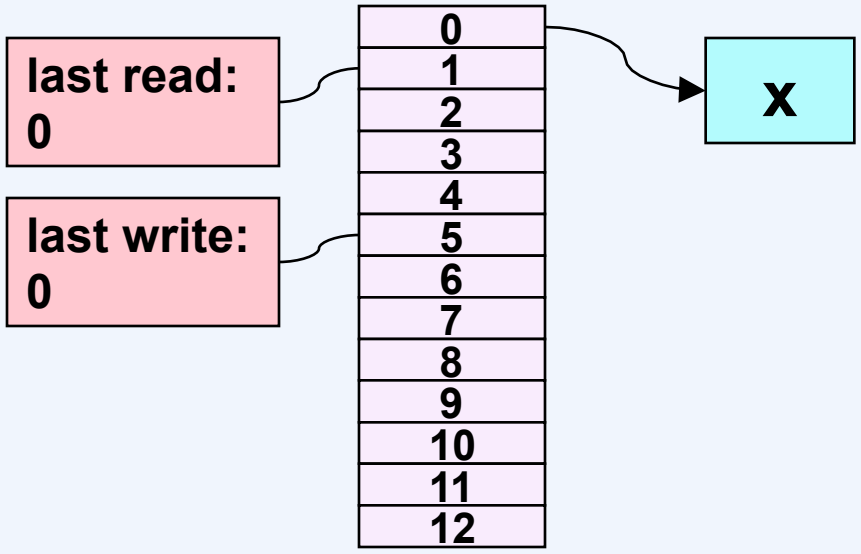
" 'tis better to ask forgiveness than permission"

- Perform transaction without bothering with concurrency control
- Check for conflicts afterwards:
  - if none, commit transaction
  - otherwise abort transaction (and start over)
- Example:
  - shadow inodes without mutually exclusive opens
  - if another shadow with modifications exists on close, abort without applying changes

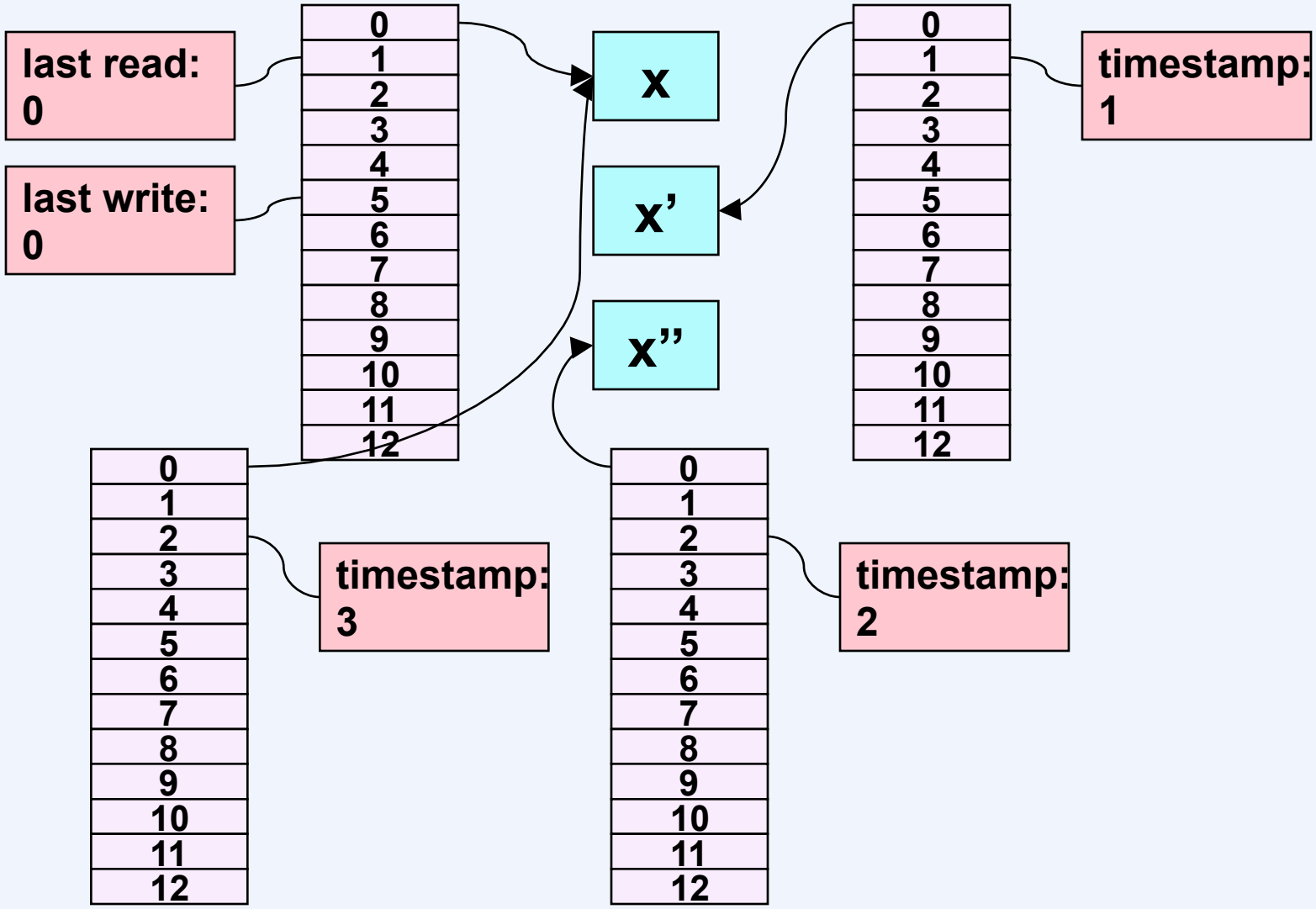
Optimistic Concurrency Control with Timestamps

- Each transaction, when started, is assigned the current timestamp; transactions are ordered by their timestamps
- Each file has last-read and last-write timestamps identifying the last committed transaction that read or wrote it
- If things are correctly ordered, whenever a transaction accesses a file, the file's timestamps will be earlier than the transaction's
- There's a problem if the transaction is writing and the file's read or write timestamp is later than the transaction's, or if the transaction is reading and the file's write timestamp is later than the transaction's
- In case of problem, abort

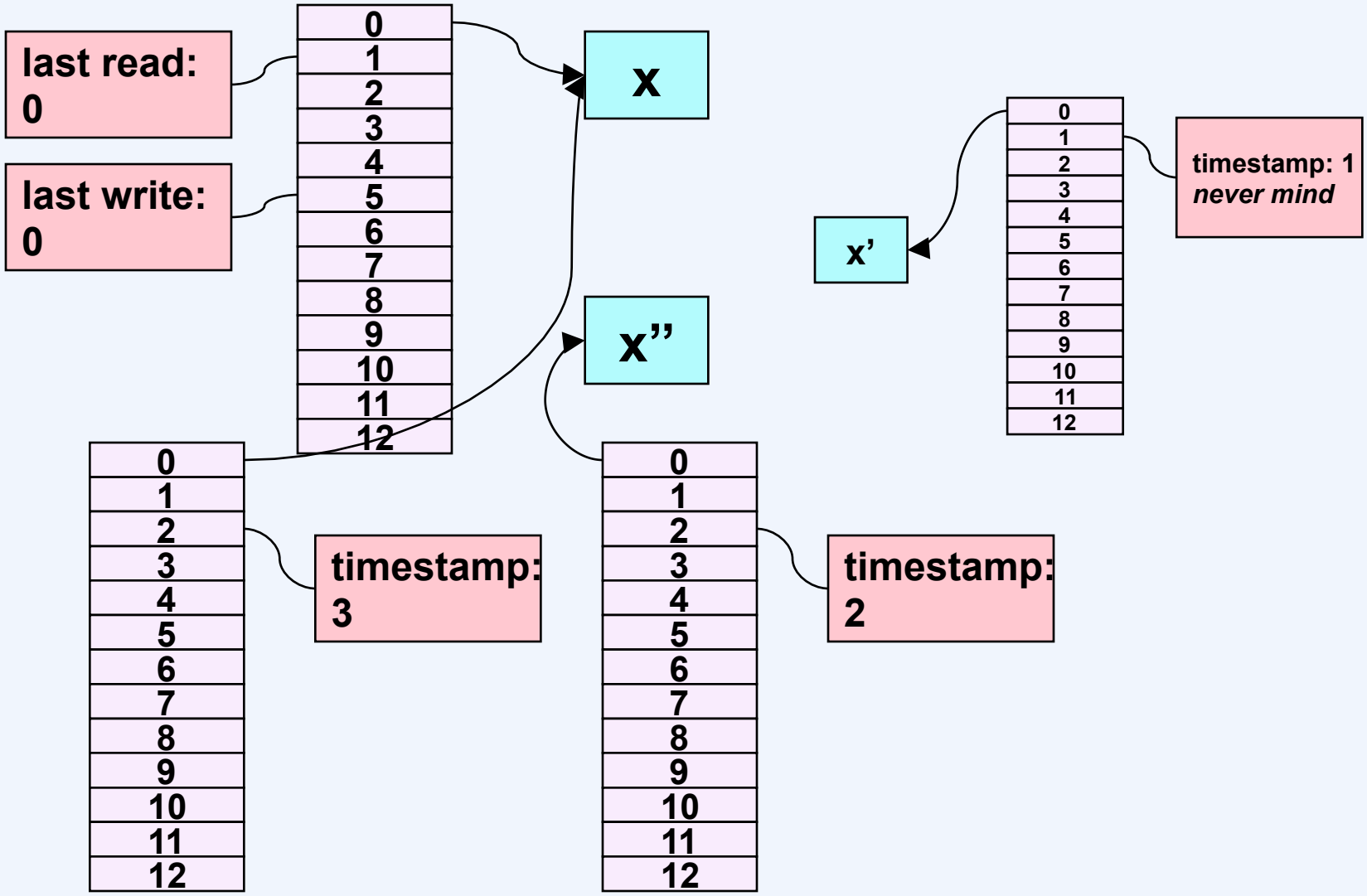
Example (1)



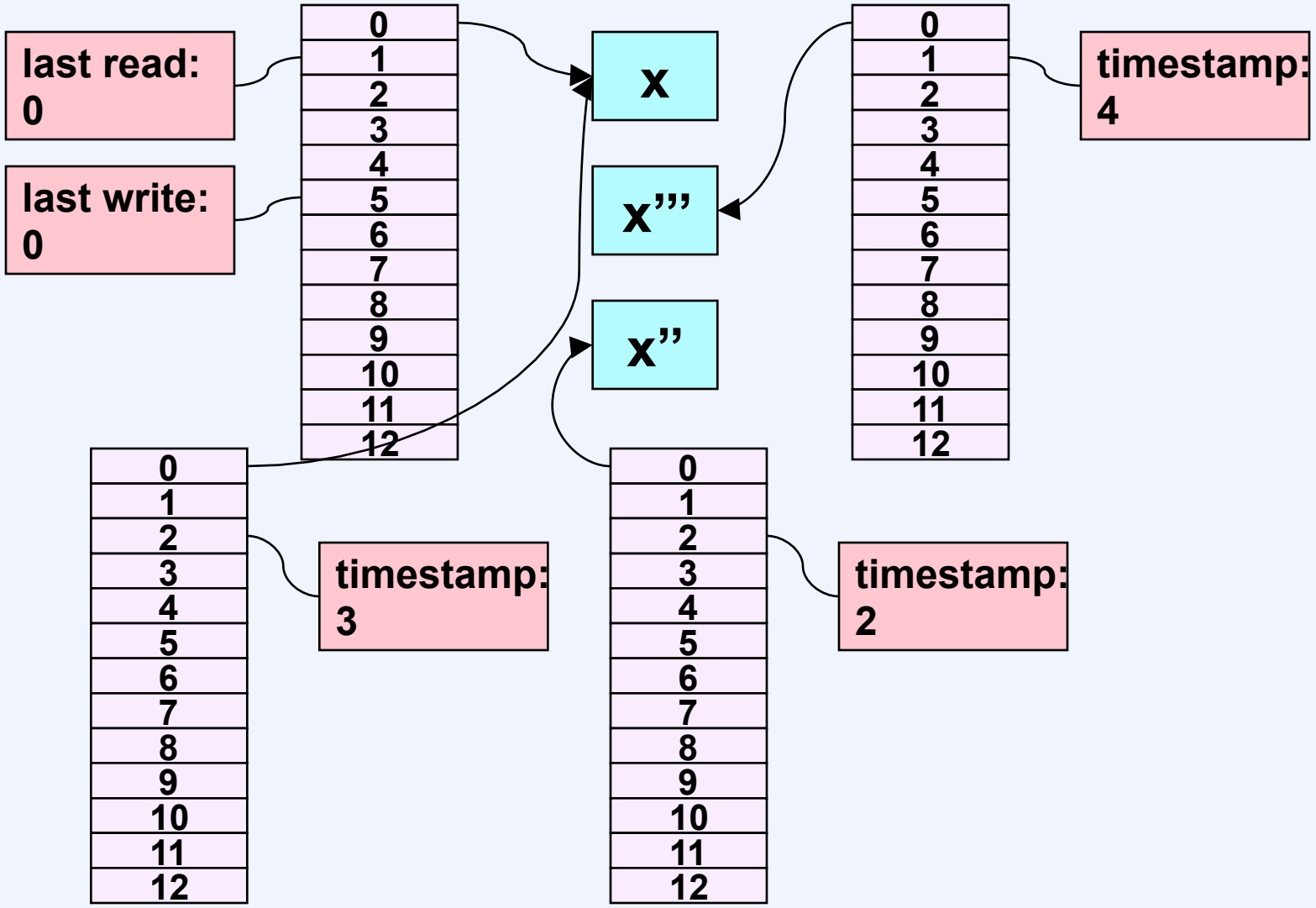
Example (2)



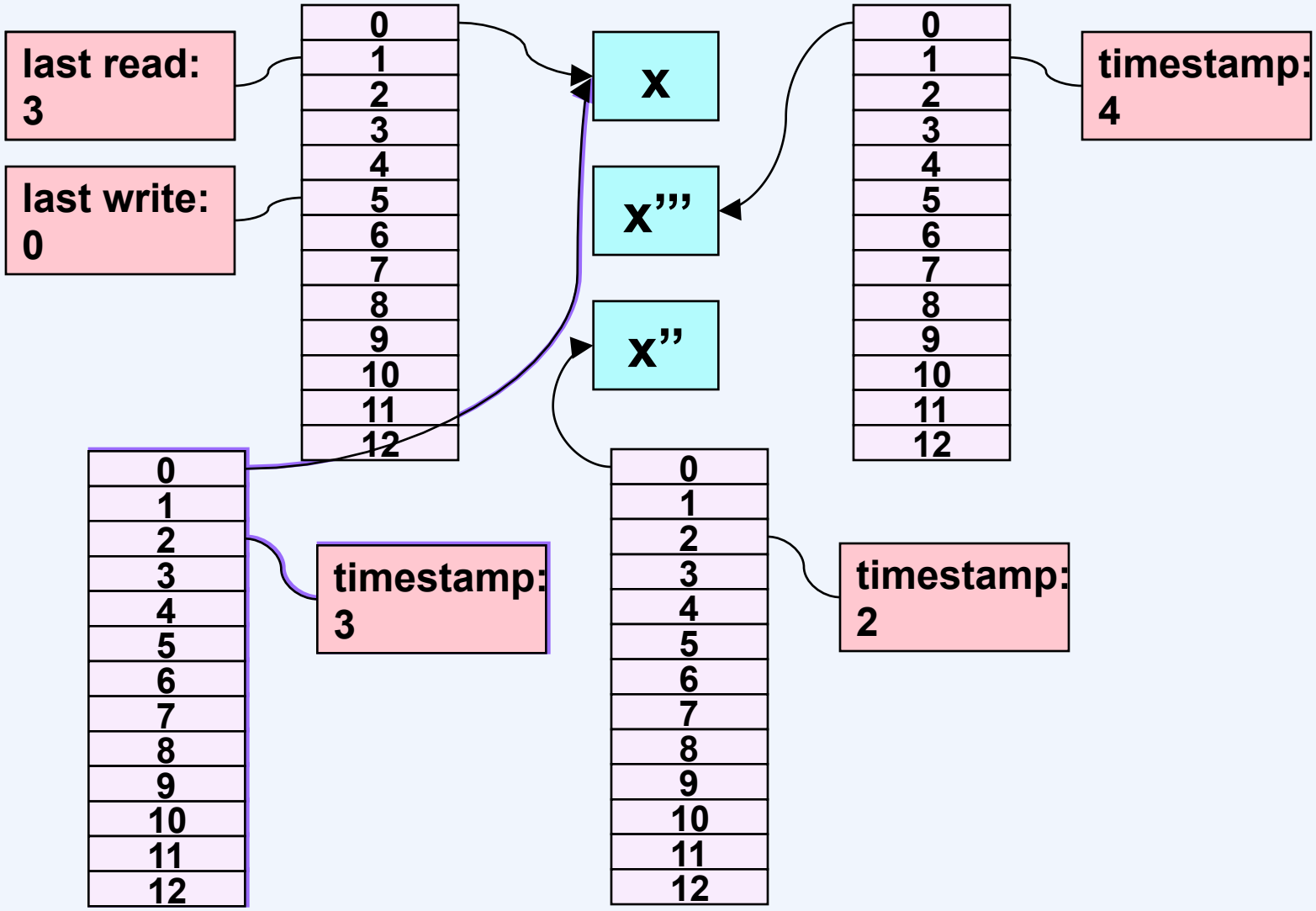
Example (3)



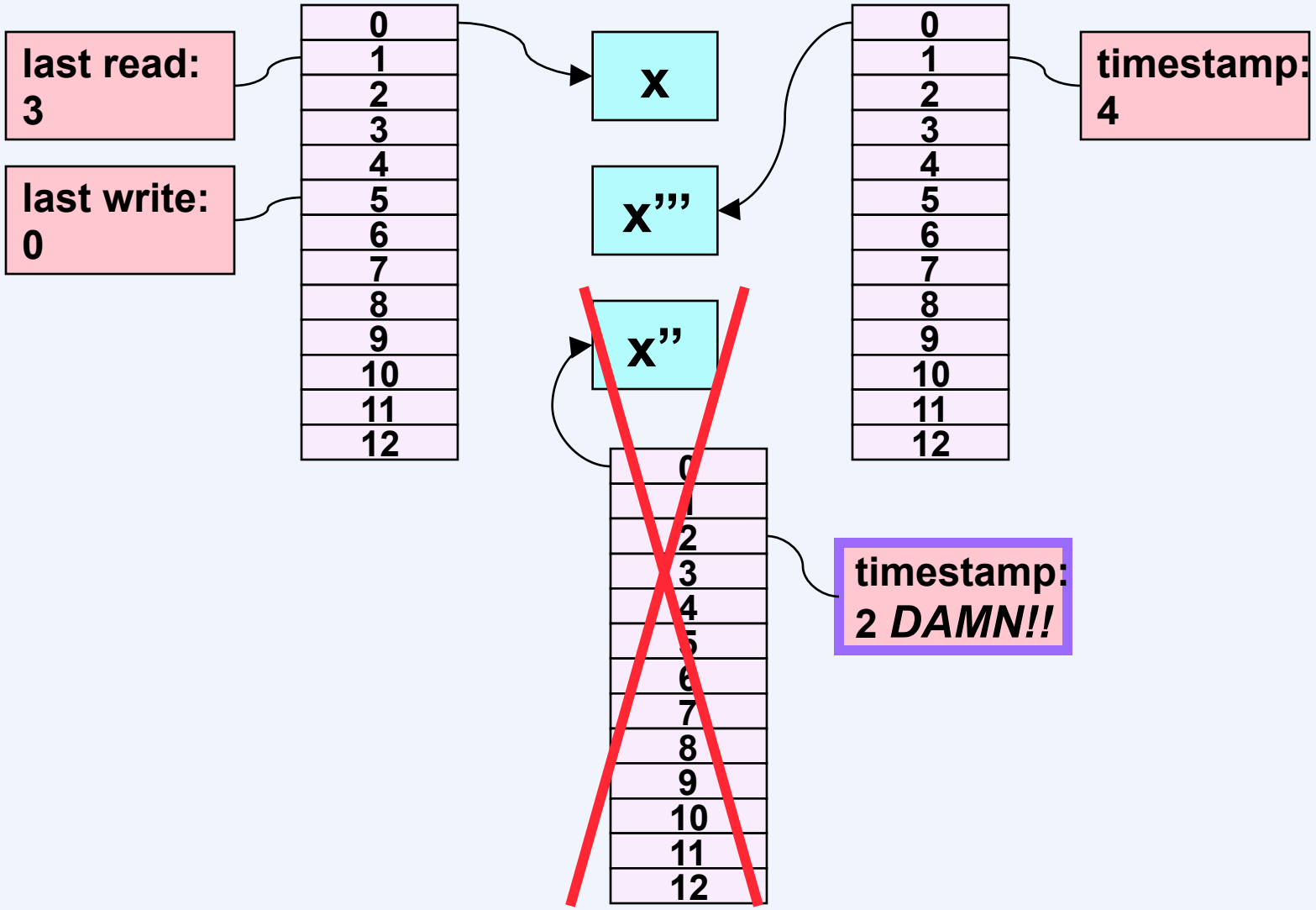
Example (4)



Example (5)



Example (6)



Example (7)