Synchronization

Physical clocks

Logical clocks

Summary so far …

A distributed system is:

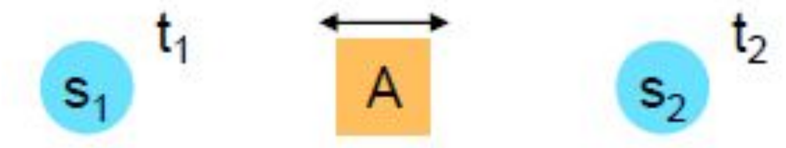
- a collection of *independent computers* that appears to its users as a *single coherent system*

Components need to:

- Communicate
  - Point to point: request-reply, RPC/RMI
  - Point to multipoint: multicast, epidemic
- Cooperate
  - Naming to enable some resource sharing
    - Naming systems for unstructured (flat) namespaces:
      - consistent hashing, DHTs
    - Naming systems for structured namespaces
  - **Event ordering**
  - **Synchronization**

An example

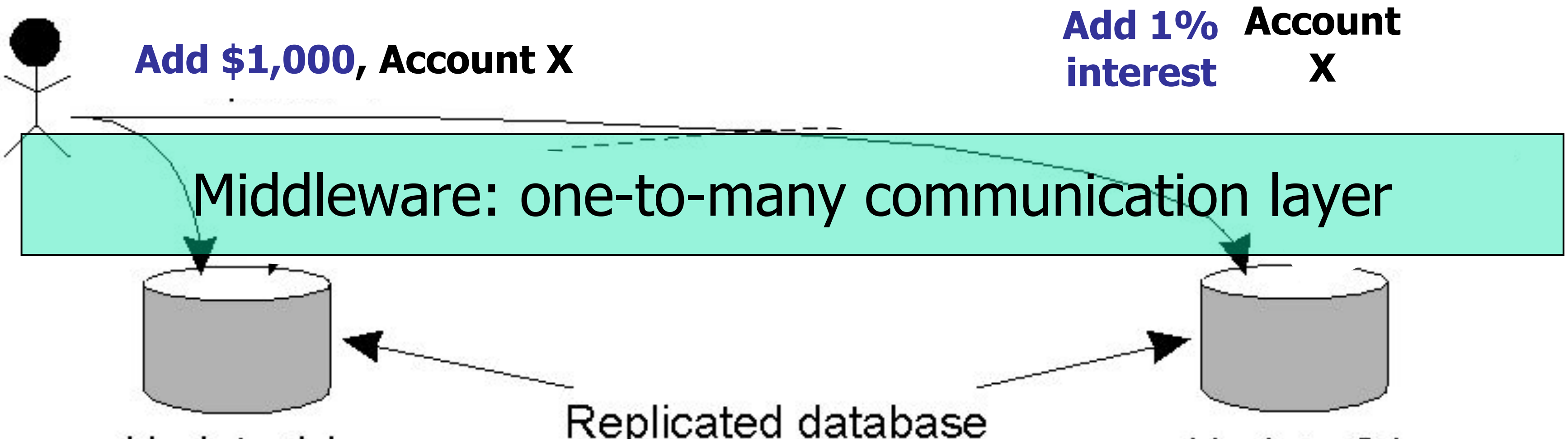
- Multiplayer game …
  - Distributed implementations generally use replicated state: each player ha
    its own view of the world.
  - Object ***A*** (the target) and the "world" are replicated as S_1 and S_2
- … players shoot the same target at about the same time
  - Players shot at local times t_1 and t_2 on two replicas S_1 and S_2
- Who gets the points?
  - Need to aggregate events into one consistent view.



Issues: Correctness. Overheads. Fairness.

Room for application-specific solutions

Example: Replicated State Machine



Add $1,000, Account X

Add 1% interest Account X

Middleware: one-to-many communication layer

Replicated database

Q: Do the updates need to be executed in the same order on the two replicas?

Issues:

- What kind of ordering: partial or total?
- Does 'real' (physical time) ordering matter for correctness?

Why event ordering is more complex than in a single-box system?

- No single physical clock
  - Likely multiple physical clocks,
  - Likely out of sync and drifting
- Need to aggregate a ‘global’ view
- Failures

What are your tools?

- Physical clocks
  - Provide [an estimate of] actual (real) time.
- ‘Logical clocks’
    - Where only ordering of events matters
  - Lamport clocks
  - Vector clocks (ability to trace event dependence)

Keeping track of time easily gets complex …

"Fifty-six standards of time are now employed by the various railroads of the country in preparing their schedules of running times"

-- New York Times, April 1883 [source]

Other Examples?

- The October with only 20 days (in 1582)
- Leap Second

Physical clocks (I)

- Problem: Achieve coordination on real time in a distributed system

- The standard: Coordinated Universal Time (UTC):
  - Atomic clocks: Based on the number of transitions per second of the cesium 133 atom
    - accurate but expensive
  - Leap second from time to time to compensate for days getting longer.

- UTC is broadcast through short wave radio and satellite.
  - Accuracy ± 1ms (but if weather conditions considered ±10ms)
  - Needs a receiver

Problem:

- Suppose we have a distributed system with a UTC-receiver somewhere in it.
- How do we:
  - distribute time to each machine, and
  - maintain a bound on how much local time differs from actual tim?

Internal mechanism (the “clock”) at each node

- Each machine has a *timer*
- Timer causes an *interrupt* *H* times a second
    - Interrupt handler adds 1 (a ‘*tick*’) to a software clock
- Software clock keeps track of the number of *ticks* since some agreed-upon time in the past.

Time is correct
Drift = 0

Clock does not deviate
Drift rate = 0

Clock drift and drift rate

Notation: Value of clock on machine *p* at real time ***t*** is ***$C_p(t)$***

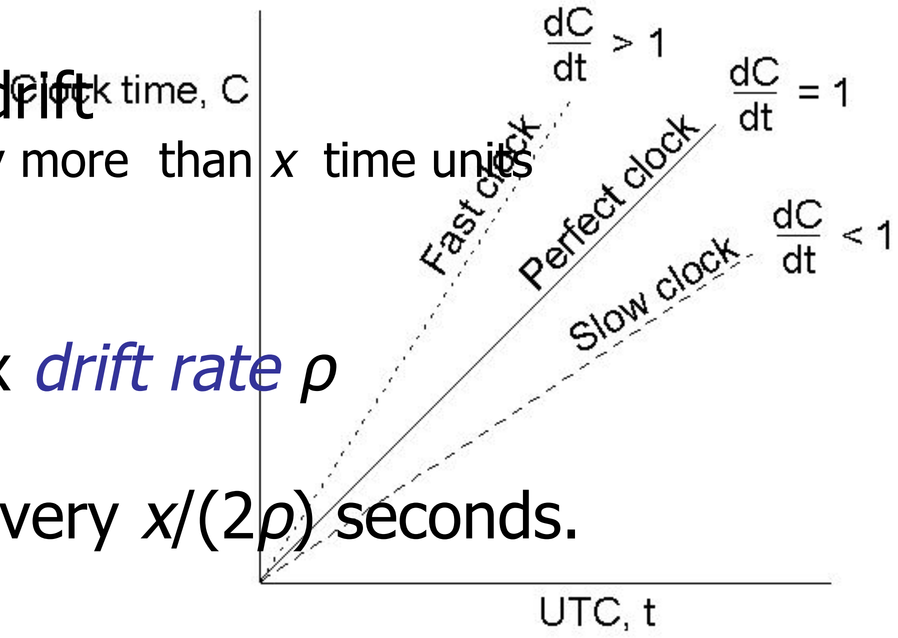
Clock drift **:** $|C_p(t) - t|$ ***Drift rate:*** $\frac{dC_p(t)}{dt} - 1$

Ideally: $C_p(t) == t$ $dC_p(t) = dt$

Goal: Guarantee on maximum drift
i.e., never let clocks on two nodes differ by more than *x* time units

How?

- Manufacturer guarantees max *drift rate* *ρ*
  $1 - \rho \leq (dC/dt) \leq 1 + \rho$
- Nodes synchronizes at least every *x*/(2*ρ*) seconds.

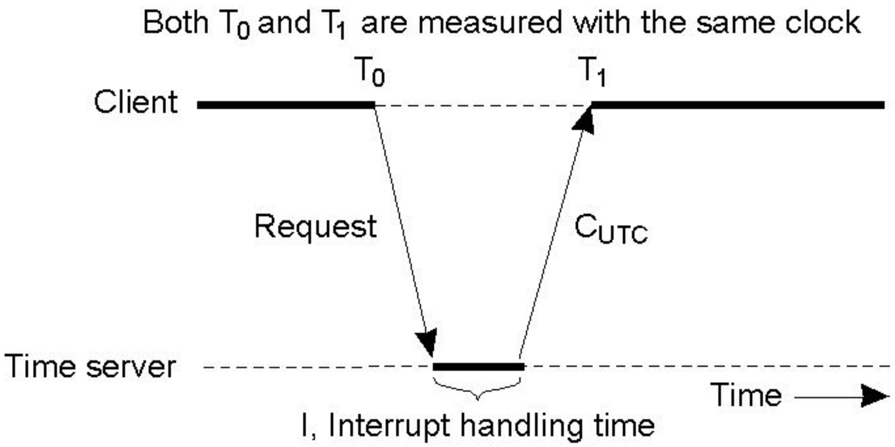


Building a complete system …

- Every machine asks a **time server** for the accurate time at least once every *x*/(2*ρ*) seconds

Client updates time to?

$$\mathbf{T_{new}=C_{UTC}+(T_1-T_0)/2}$$



Both T_0 and T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time server
Time
I, Interrupt handling time

Q: Problems?

- **Fundamental:** setting the time back is **never** allowed □ smooth adjustments.

Real world: Network Time Protocol (NTP)

- Stratum 1 NTP servers – receive time from external sources (cesium clocks, GPS, radio broadcasts)
- Stratum N+1 servers synchronize with stratum N servers and between themselves
  - Self-configuring network
- Survey (fairly old: 2006)
  - > 1M NTP servers
  - 0.2% of the NTP servers >128ms offset from synchronization peer
  - Excluding these: - median: 0.7ms
    - mean: 7ms

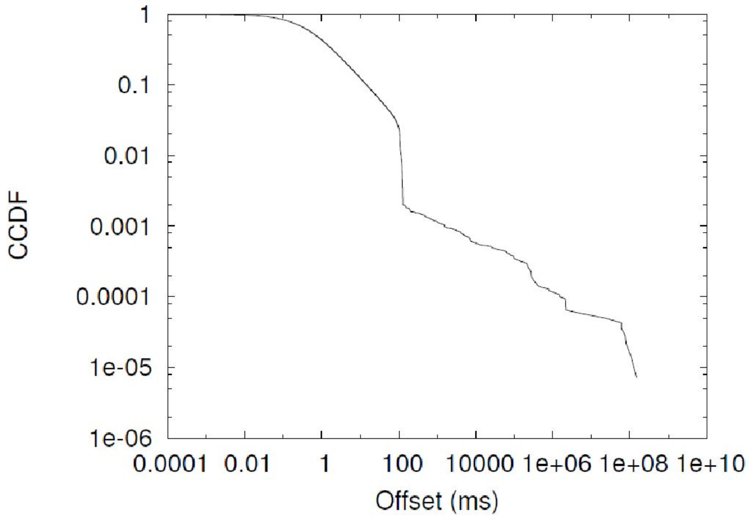


Fig. 1: CCDF of host–peer offsets

[Summary so far] Physical clocks

- Clock drift: time difference between two clocks

- Sources of errors (drift)
  - Variability in time to propagate radio signals. (±10ms)
  - Clocks are not perfect: Drift rates
  - Network latencies are not symmetric
  - Differences in speed to process messages

- System design to limit drift
  - One node holds the ‘true’ time
  - Other nodes contact this node periodically and adjust their clocks
    - How often?
    - How exactly the adjustment is done?

- We’ve established that clocks can not be *perfectly* synchronized (and atomic clocks are costly).

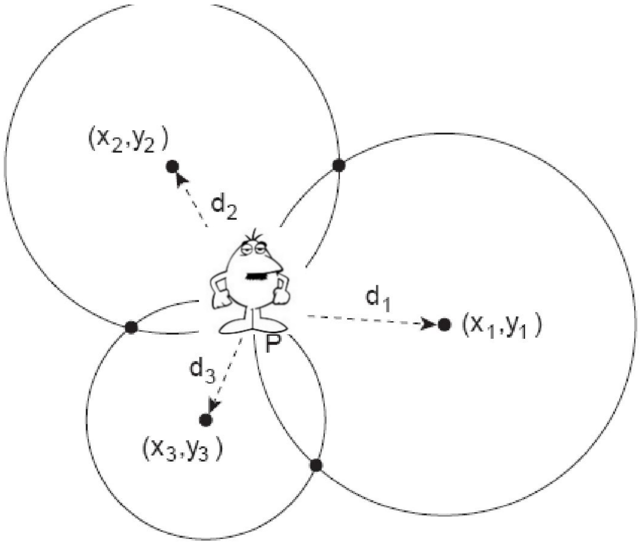
- What can one do in these conditions?
  - Get a better estimate of time by using different technology
    - e.g., use GPS to obtain time in your system
  - Expose uncertainty and design the system to take drift into account
    - Example 1: Google’s Spanner
    - Example 2: Server design to provide at-most-once semantics
  - Give up physical clocks!
    - Consider only event order - Logical clocks

GPS – Global Positioning Systems Intro

Basic idea: To estimate distance to a landmark (e.g., s satellite)

- Estimate signal *propagation time* between you and the landmark
- Multiply by signal speed

Strawman: Assume that the clocks of the satellites and receiver are accurate and perfectly synchronized:



But, in real world:

- 3D not 2D
- The receiver’s clock is definitely out of sync with the satellite

- Unknowns: x_r, y_r, z_r coordinates of the receiver.
- Known:
    - x_i, y_i, z_i coordinates of satellite i
    - T_i is the send timestamp on a message from satellite *i*
    - $\Delta I_i = (T_{now} – T_i)$ measured delay for message sent by satellite *i*
- Distance to satellite *i can be estimated:*
    - (1) Propagation time*: $d_i = c \text{ x } \Delta I_i$*
    - (2) Geometric distance:

$$d_i = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

- 3 satellites□ 3 equations in 3 unknowns. I'M DONE!
- … BUT: I assumed receiver clock is synchronized!

*the satellite has an atomic clock anyways so Ti is correct

- Unknowns: x_r, y_r, z_r coordinates of the receiver.
- Known: X_i, Y_i, Z_i - coordinates of satellite i
  - T_i - the send timestamp on a message from satellite *i*

$\Delta I_i = (T_{now} – T_i)$ propagation delay of message sent by satellite *i*

- Distance to satellite *i can be estimated:*
  - Propagation time*:* $d_i = c \times \Delta I_i$
  - Geometric distance:

$$d_i = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

- So far I assumed receiver clock is synchronized*!
  - What if it needs to be adjusted? $T_{real} = T_{now} + \Delta r$
  - $\Delta I_i = (T_{now} + \Delta r) – T_i$
  - Collect one more measurement from one more satellite!

*the satellite has a atomic clock anyways so Ti is correct

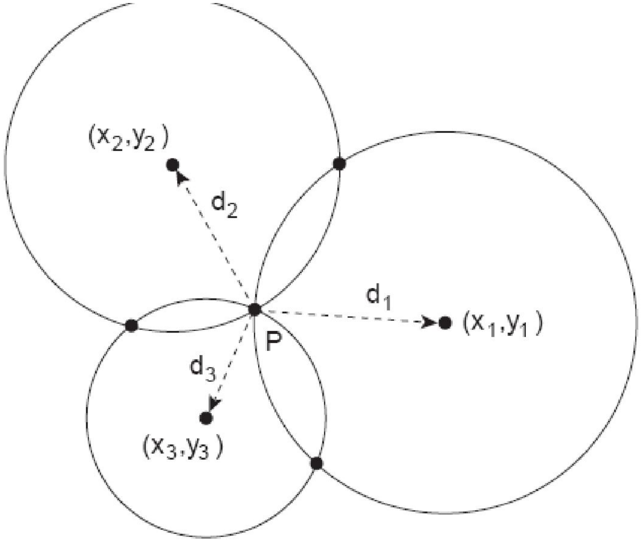
Two takeaways

- (1) Triangulation technique
  - Can be used in other contexts: e.g., computing geographical position in wired networks
- (2) Enough information to correct the clock drift at the receiver

Similar triangulation techniques work in other contexts: Computing geographical position in wired networks

Observation: a node *P* needs at least *k* + 1 landmarks to compute its own position in a *k*-dimensional space.

Consider two-dimensional case:



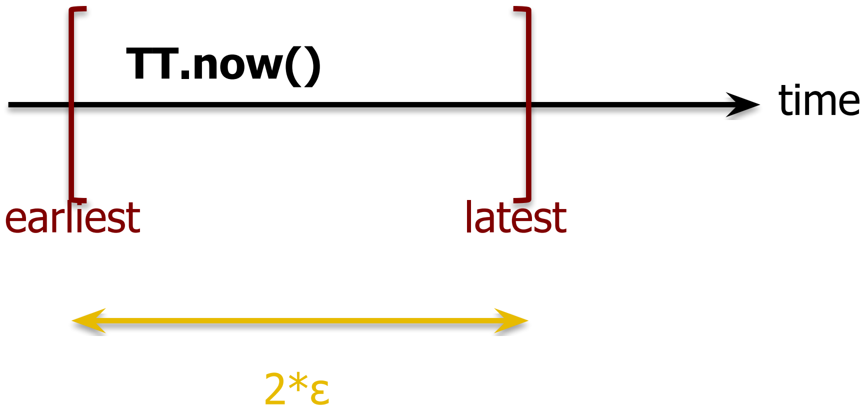
Solution: *P* needs to solve three equations in two unknowns (x_P,y_P)

- We’ve established that clocks can not be *perfectly* synchronized (and atomic clocks are costly).

- What can one do in these conditions?
  - Get a better estimate of time by using new technology
    - e.g., use GPS to obtain time in your system
  - Expose uncertainty / design the system to account for drift
    - Example 1: Google’s Spanner – Extended time API
    - Example 2: Server design to provide at-most-once semantics
  - Give up physical clocks!
    - Consider only event order - Logical clocks

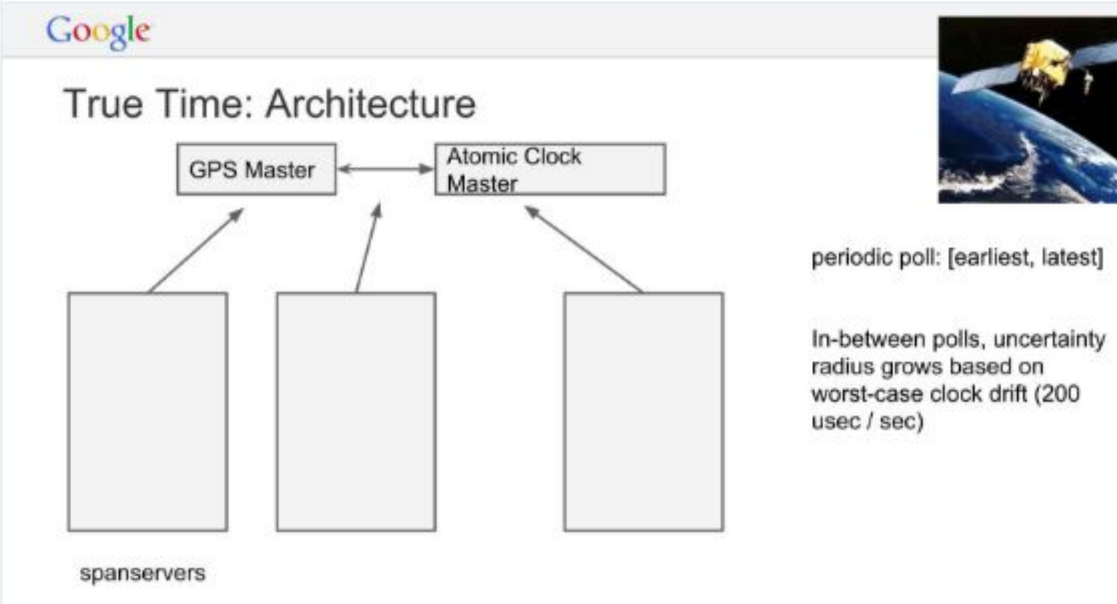
Google’s Spanner

- “Global wall-clock time” with bounded uncertainty
  - Time estimate: TT.now() □ [earliest, latest]
  - Guaranteed interval



Spanner: Google’s Globally-Distributed Database - internally ‘11, OSDI 12, product ‘17

TrueTime implementatior

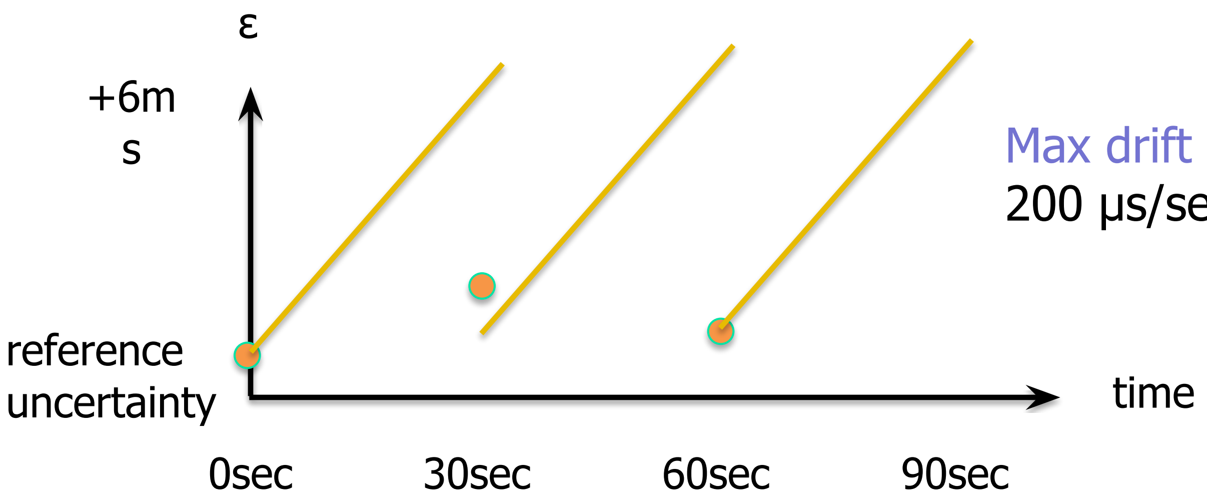


True Time: Architecture

periodic poll: [earliest, latest]

In-between polls, uncertainty radius grows based on worst-case clock drift (200 usec / sec)

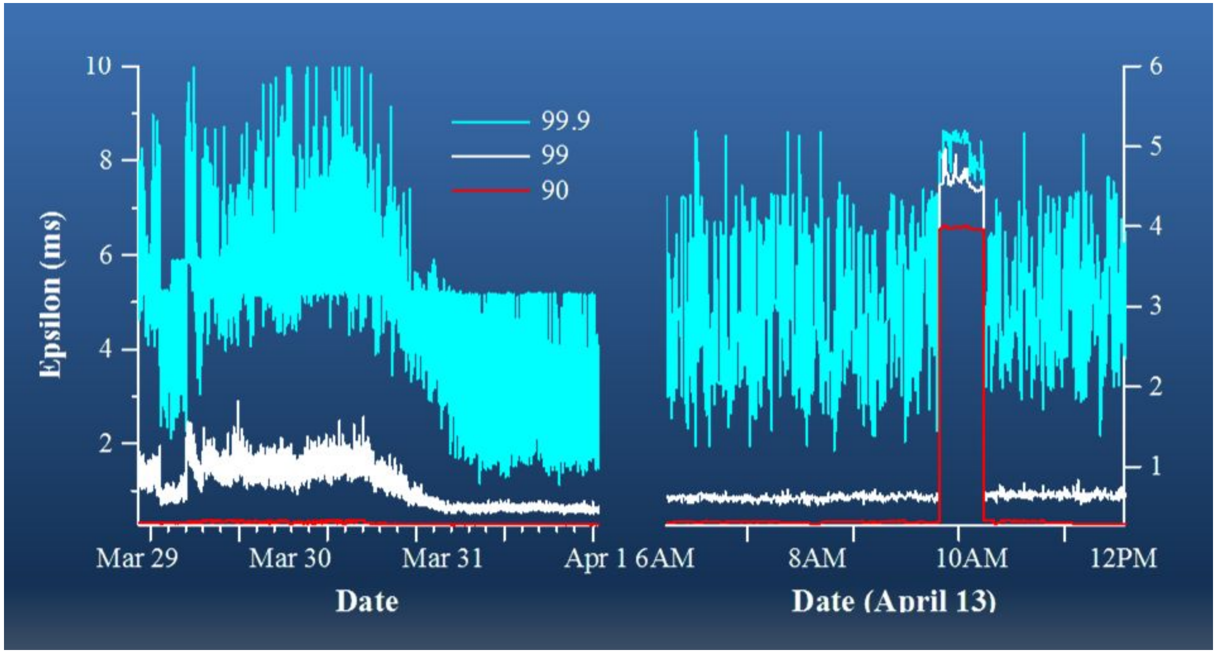
ε = reference ε + worst-case local-clock drift



Max drift rate: 200 µs/sec □ Max drift: 6ms (for a 30s sync interval)

Practical Experience @Google

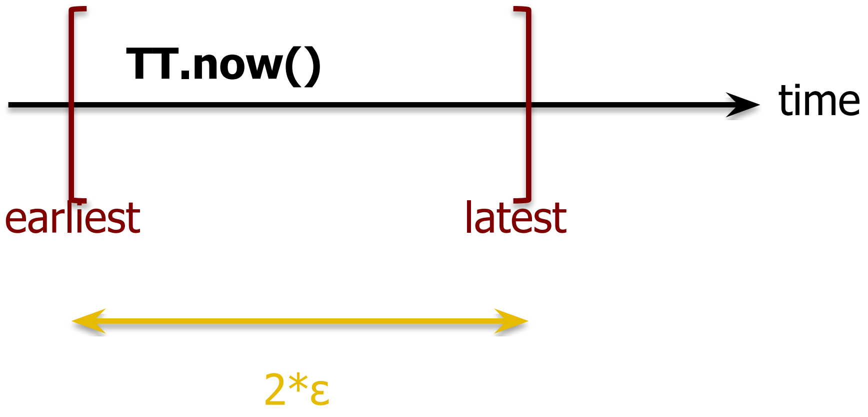
- (1) Is *ε* truly a bound on clock uncertainty?
  - Violations of guarantees [drift rate] unlikely: Bad CPUs 6 times more likely than bad clocks (based on 1y of data)
    - "As a result, we believe that TrueTime's implementation is as trustworthy as any other piece of software upon which Spanner depends"
- (2) How bad does *ε* get?
  - Network-induced uncertainty



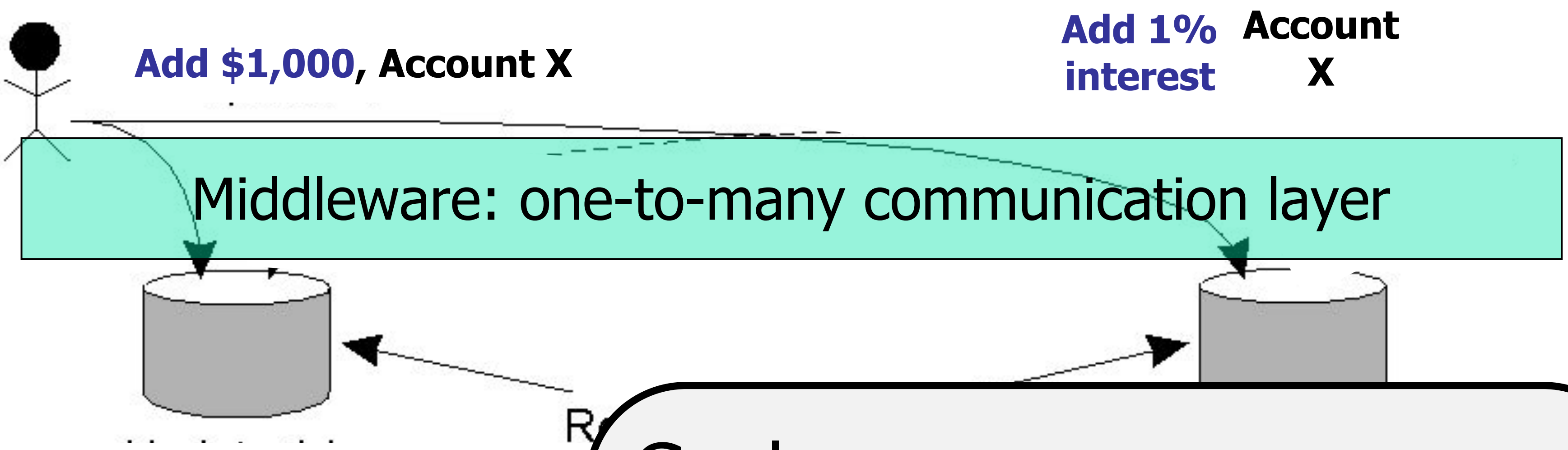
ε sampled at timeslave daemons immediately after polling the time masters

Recap: expose uncertainty of time estimates

- "Global wall-clock time" with bounded uncertainty
  - Time estimate
  - Guaranteed interval



How would Spanner/TrueTime help here ….



Goal (for next few slides):

aggregate data from transaction initiators to construct a correct ordering of events

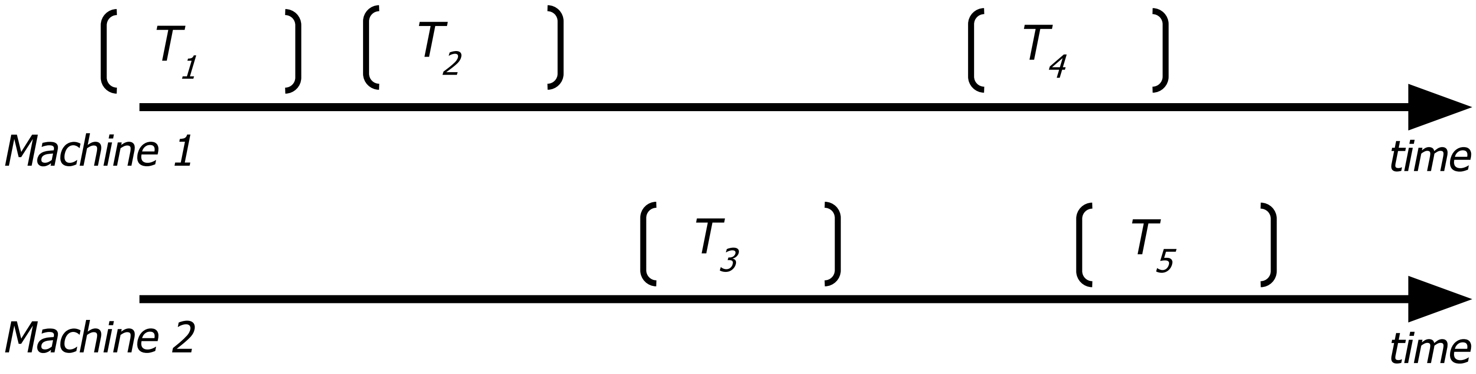
[NB: This is a over-simplified view of spanner]

Q: Do the updates r
same order on the t

Issues:

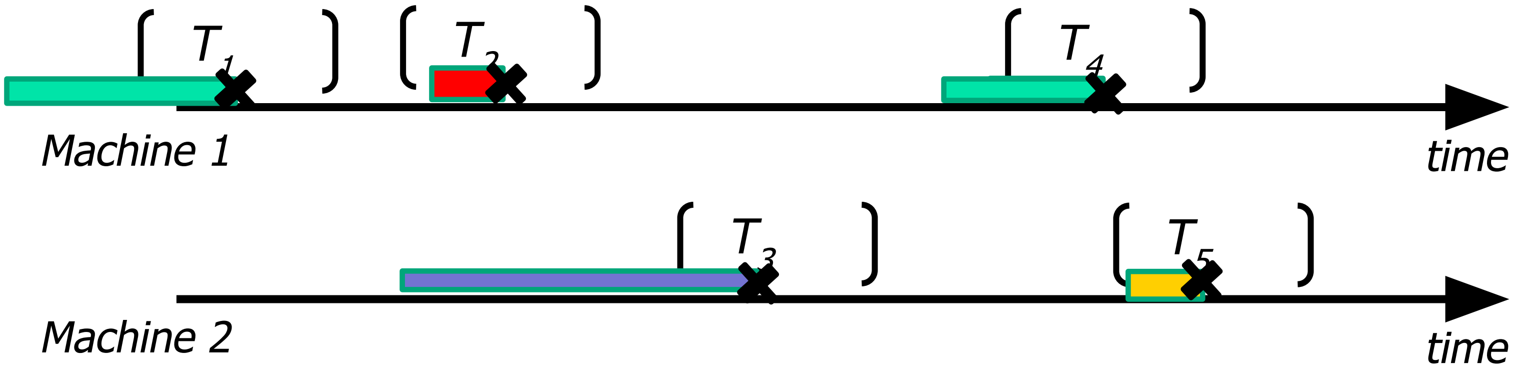
- What kind of ordering
- Does ‘real’ (physical time) ordering matter for correctness?

Ok – so how does this help with event ordering?



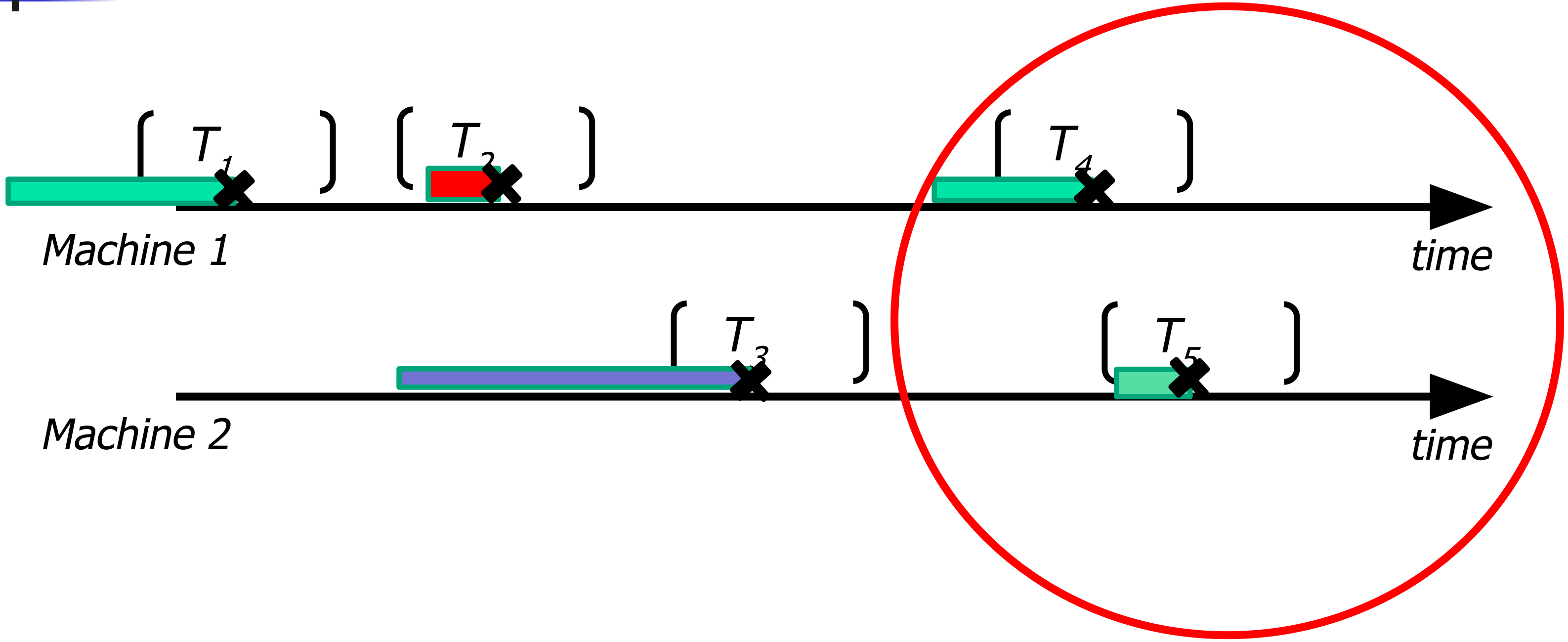
- Events get timestamped (with uncertainty intervals)
- Use the timestamps to recreate a global view for the order in which events occurred.
  - Order unambiguous as long as the uncertainty intervals do not overlap

Ordering transactions



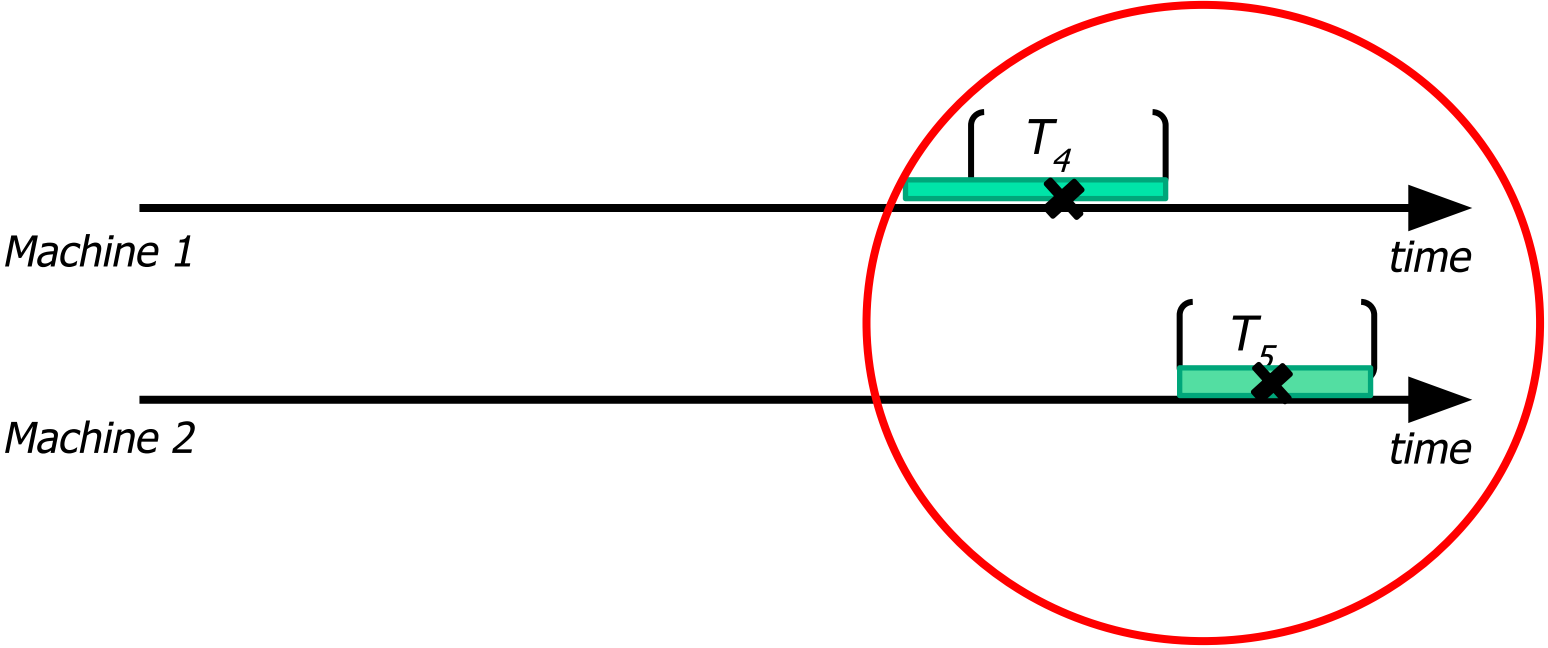
- Assume that T_x are transaction timestamps.
  - Timestamps taken at commit time
- Goal is to recreate the global transaction order
  - If two transactions are not conflicting: order does not matter
  - Else: need to get the order right

Ordering transactions



- Assume that T_x are transaction commit timestamps.
- Goal is to recreate the global transaction order
  - If two transactions are not conflicting: order does not matter
  - Else: need to get the order right

Spanner solution



- Move timestamp within the transaction
  - So that full uncertainty interval is within the transaction
- If needed (i.e., transaction is too short) extend the transaction
  - i.e., delay releasing the resources
- This way conflicting transactions have timestamps that can be ordered without uncertainty

- We’ve established that clocks can not be *perfectly* synchronized (and atomic clocks are costly).
- What can one do in these conditions?
  - Get a better estimate of time
    - e.g., use GPS to extract time in your system /
  - Expose uncertainty and design the system to take drift into account
    - Example 1: Google’s Spanner – Extended time API
    - Example 2: Server design to provide at-most-once semantics
  - Give up physical clocks!
    - Consider only event order - Logical clocks

Efficient at-most-once message delivery

Goal : Server has to identify previously served requests to implement at-most-once semantics

[Old] Assumptions:

- Message propagation time bounded
- Physical clocks not used
  - (so no need to make assumptions about drift rates)
- Client may resend messages for up to *MaxLifeTime*

Issues

- 1: For how long to maintain ‘transaction’ data?
- 2: How to deal with server failures?
  - (minimize the state that is persistently stored, when to restart)

Issue1: For how long to maintain transaction data at server?

- Server's goals:
  - 1. Identify message duplicates – at-most-once delivery
  - 2. While trying to avoid storing too much state

- Context:
  - No bound on message propagation time
  - Any two clocks in the system differ by at most *MaxClockDrift*
  - Client may resend messages for up to *MaxLifeTime*

- Mechanism idea: Server discards messages that have been generated too far in the past.
  - Client protocol: client sends transactionID and physical timestamp
  - Server computes: $G = T_{now} - MaxLifeTime – MaxClockDrift$
    - … and maintains transaction data only for the interval $[G..T_{\text{now}}]$
  - Discards messages with $msg_{timestamp}$ older than G
  - Ignores (or delays) messages that arrive in future: $msg_{timestamp} > T_{current}$

Efficient at-most-once message delivery

Design goal : Server has to identify previously served requests to implement at-most-once semantics

[New] Assumptions:

- No bound on message propagation time
- There is a bound on clock drift
  - any two clocks in the system differ by at most *MaxClockDrift*
- Client may resend messages for up to *MaxLifeTime*
  - *after that reports error (timeout) to the application*

Issues

- 1: For how long to maintain 'transaction' data?
- 2: How to deal with server failures? What to persist across server failures? When to restart?
  - (minimize the state that is persistently stored, minimize downtime)

Efficient at-most-once message delivery (II)

- Issue 2: What to persistently store across server failures?
  - Strawman #1: Store nothing persistently.
    - Incorrect if server reboots quickly
  - Strawman #2: Persistently store ALL transactions.
    - Costly
  - Strawman #3: Store nothing persistently, and wait *MaxLifeTime* after reboot before starting to process messages
    - Correct but lowers availability.

Efficient at-most-once message delivery (II)

- Issue 2: What to persistently store across server failures?

- Towards a solution: need to approximate failure time
  - Write current time (*CT*) to disk every *ΔT*
  - At recovery read it
    - Failure time is approximated as $G_{failure}$ from last saved *CT*

  - After recovery – when a new message arrives
    (let's ignore clock drift for now and assume perfect clocks)
    - Discard messages with timestamp older than $G_{failure} + \Delta T$
      - *Reason:* the server might have seen these in the past (but lost the cache)
    - Process messages with timestamp newer than $G_{failure} + \Delta T$
      - *Reason: surely not seen before failure*

[Quiz-like question: Change the formulas to consider clock drift]

Quiz-like question

- Previous solution
    - no assumption on maximum message propagation time
    - uses physical clocks (with a bound on max clock drift)

- You can now assume an upper bound *B* on message propagation time
    - Design a solution that does not use physical clocks?

- We’ve established that clocks can not be *perfectly* synchronized (and atomic clocks are costly).

- What can I do in these conditions?
  - Get a better estimate of time by using new technology
    - GPS systems
  - Expose uncertainty / Design the system to take drift into account
    - Example: Server design to provide at-most-once semantics
  - Give up physical clocks!
    - Consider only event order - Logical clocks

Logical clocks -- Time Revisited

- What’s important?
  - The precise time two events occurred?

OR

  - The order in which the two events occured?

(1) Alice intends to quit her employer

- She removes her boss as a friend
- Posts *"I'll quit tomorrow! Here is the story: … "*

  (visible to friends only)

Alice expects her boss will not be able to see her post

(2) Bob, a friend of Alice,

- Reads Alice's post
- Messages Charlie, a common friend:

  "Wow! Alice just posted that she'll quit! Read her story!" *

(3) Charlie, a friend of both

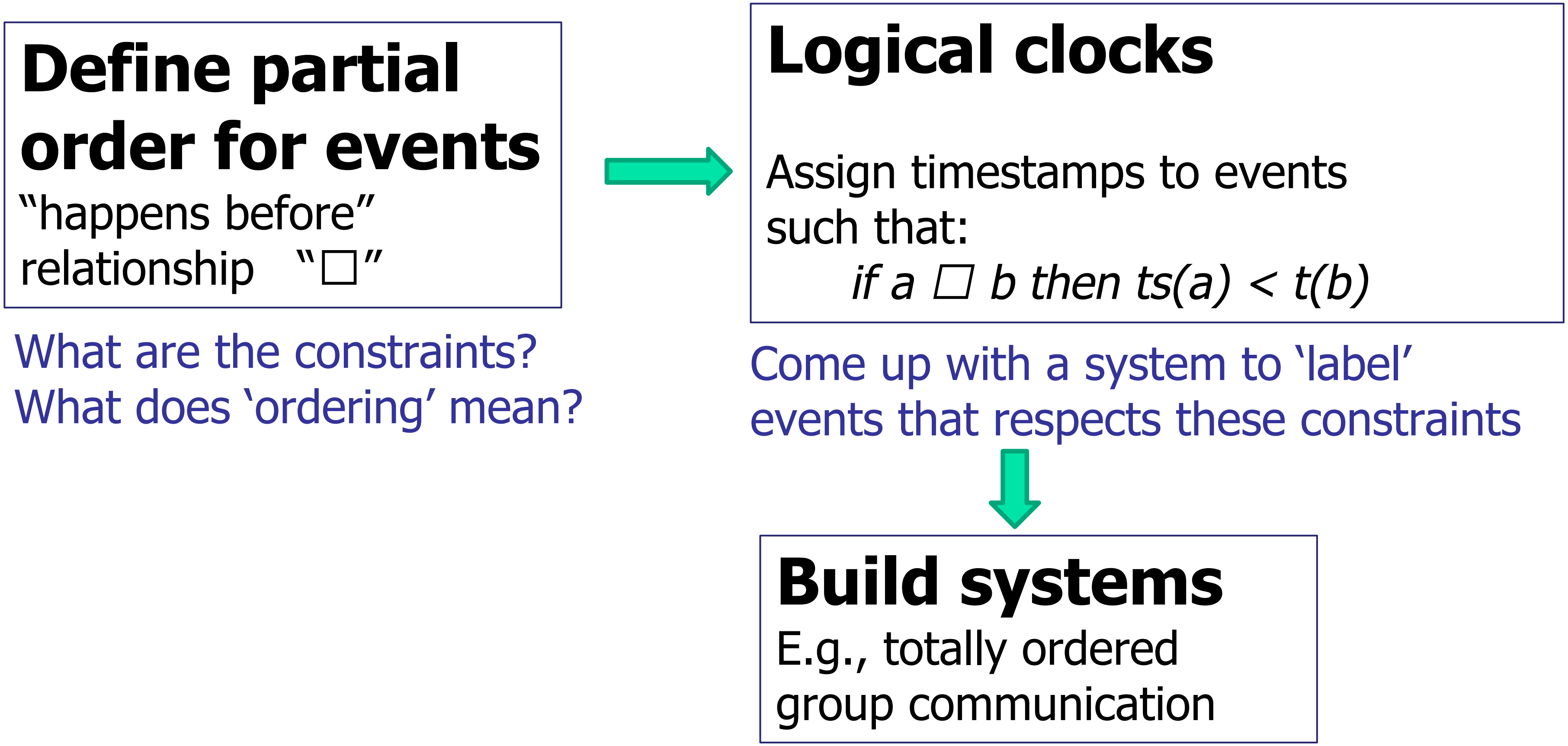
- Reads Bob's message
- Goes to Alice's timeline to read the story

Charlie expects to see Alice's post

Common expectation: process events in the order they occurred

* We are in 2003, post sharing has not been invented

Logical clocks: ROADMAP



Define partial order for events

"happens before" relationship "□"

What are the constraints?

What does 'ordering' mean?

Logical clocks

Assign timestamps to events such that:

if a □ b then ts(a) < t(b)

Come up with a system to 'label' events that respects these constraints

Build systems

E.g., totally ordered group communication

How is this used?

What are the constraints?
What does ‘order’ mean?

Need to introduce a notion of ordering (before we can order anything).

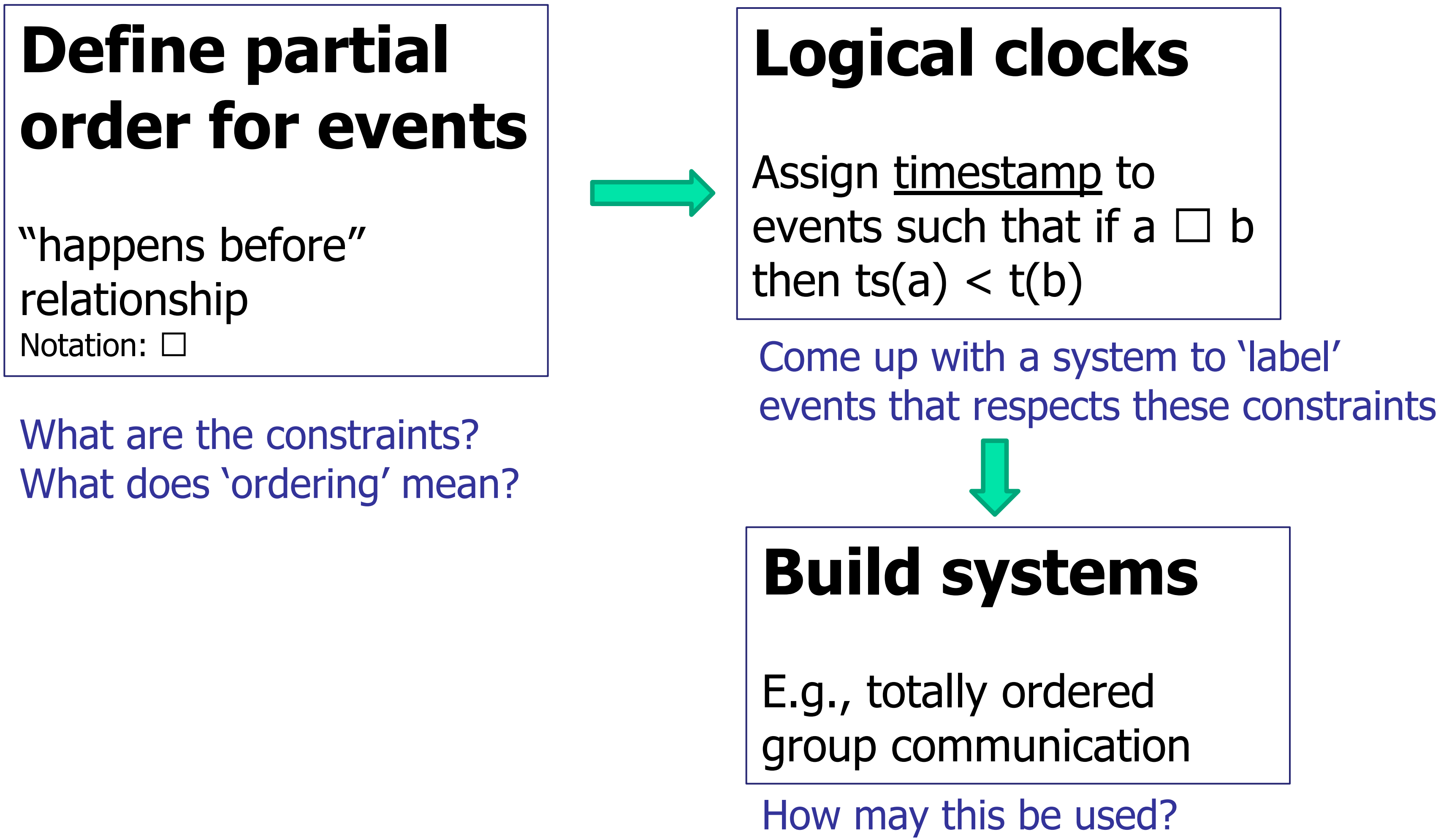
The **happened-before** relation (notation: “→”) on a set of events in a distributed system

- if *a*, *b* are events in the same process, and *a* occurs before *b*, (in physical time) then *a* → *b*
- if *a* is the event of sending a message by a process, and *b* receiving same message by another process then *a* → *b*

Property: Transitive: if a□b and b□c then a□ c

Two events are *concurrent* if nothing can be said about the order in which they happened (i.e. happens-before is a partial order)

Logical clocks: ROADMAP



Define partial order for events

"happens before" relationship

Notation: □

What are the constraints?
What does 'ordering' mean?

Logical clocks

Assign timestamp to events such that if a □ b then ts(a) < t(b)

Come up with a system to 'label' events that respects these constraints

Build systems

E.g., totally ordered group communication

How may this be used?

Logical clocks

Objective: Build a view on the system's behavior that is consistent with the 'happened-before' relation

Attach a timestamp *ts*(*e*) to each event *e*, such that:

- **P1:** If *a* and *b* are events in the same process, and *a* happened before in physical time *b*, then we demand that *ts*(*a*) < *ts*(*b*).
- **P2:** If *a* corresponds to sending a message, and *b* to the receipt of that message, then also *ts*(*a*) < *ts*(*b*).

- **Problem:** How to attach timestamps to all events in the system (consistent with the rules above) when there's no global clock
  - maintain a **consistent** set of logical clocks, one per process.

Problem: Need to attach timestamps to all events in the system

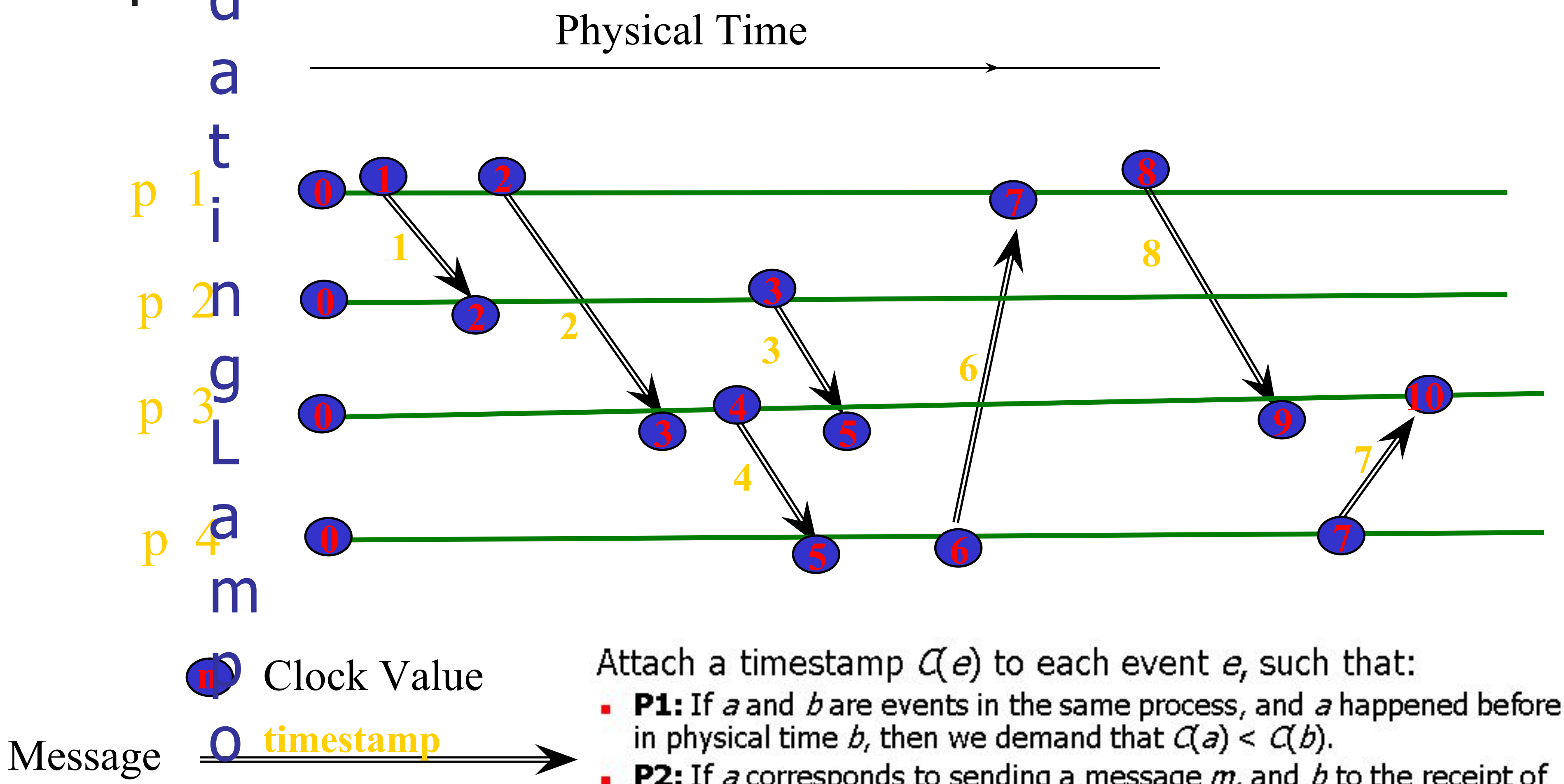
- maintain a consistent set of logical clocks, one per process.
- there's no global clock

Solution (Lamport)**:** Each process P_i maintains a **local** counter C_i and adjusts it as follows:

1) For any two successive events that take place within P_i the counter c_i is incremented by 1.
2) Each time a message m is sent by process P_i the message is timestamped $ts(m) = c_i$
3) Whenever a message m is received by a process P_j, P_j adjusts its local counter c_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

- Property **P1** is satisfied by (1); Property **P2** by (2) and (3).

Updating Lamport’



Attach a timestamp *C*(*e*) to each event *e*, such that:

- **P1:** If *a* and *b* are events in the same process, and *a* happened before in physical time *b*, then we demand that *C*(*a*) < *C*(*b*).
- **P2:** If *a* corresponds to sending a message *m*, and *b* to the receipt of that message, then also *C*(*a*) < *C*(*b*).

Note: C must only increase in each process

Quiz-like question

Notation: ***timestamp(a)*** is the Lamport logical clock associated with event *a*

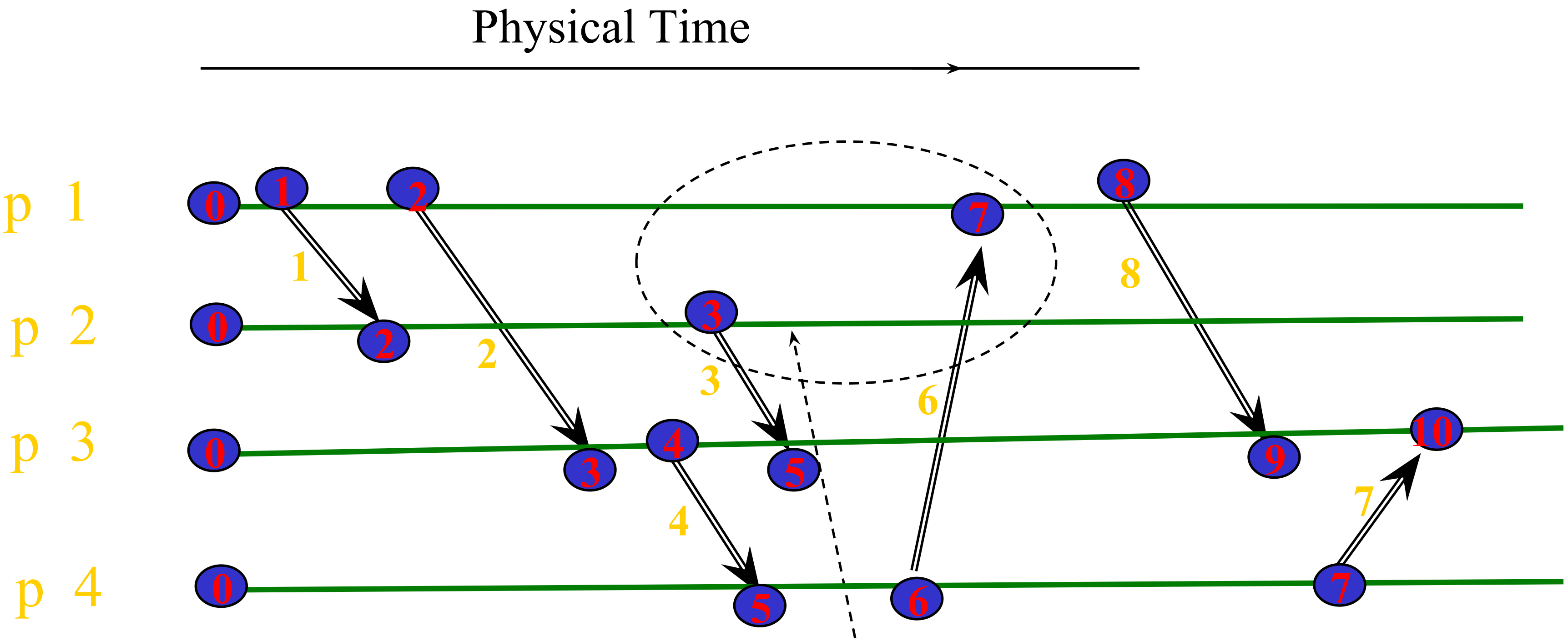
By construction if ***a*** □ ***b*** => ***timestamp(a) < timestamp(b)***
(if ***a*** happens before ***b***, then ***timestamp(a) < timestamp(b)***)

Q: is the converse true?
That is: if timestamp(a) < timestamp(b) => ?✗ ***a*** □ ***b***

No. If ***timestamp(a) < timestamp(b),*** it does NOT imply that ***a*** happens before ***b***

Q: Do I know anything timestamp(a) < timestamp(b)?
A: Only that b□ a is FALSE (i.e.. b surely did NOT happen before a, more concrete:
a and b concurrent OR a happened before b)

Example



n Clock Value

timestamp Message

concurrent events can not be detected based on Lamport clock info

Note: Lamport Timestamps: 3 < 7, but event with timestamp 3 is concurrent to event with timestamp 7, (events are not in ‘happen-before’ relation).

Last Time: Logical time

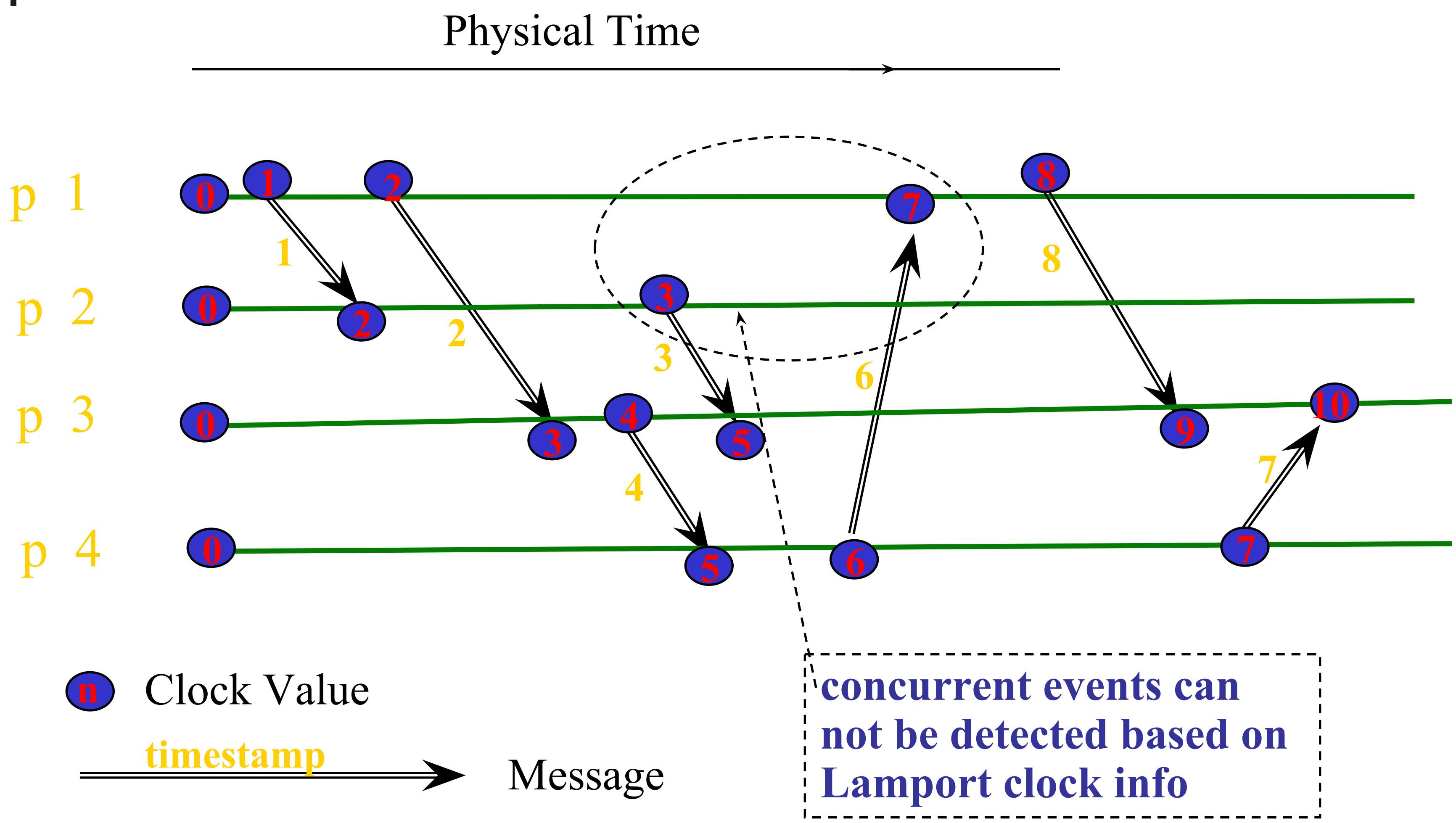
What is that?

- Discrete assignment of sequence numbers to events, which preserves "happens-before" order

How do Lamport clocks work?

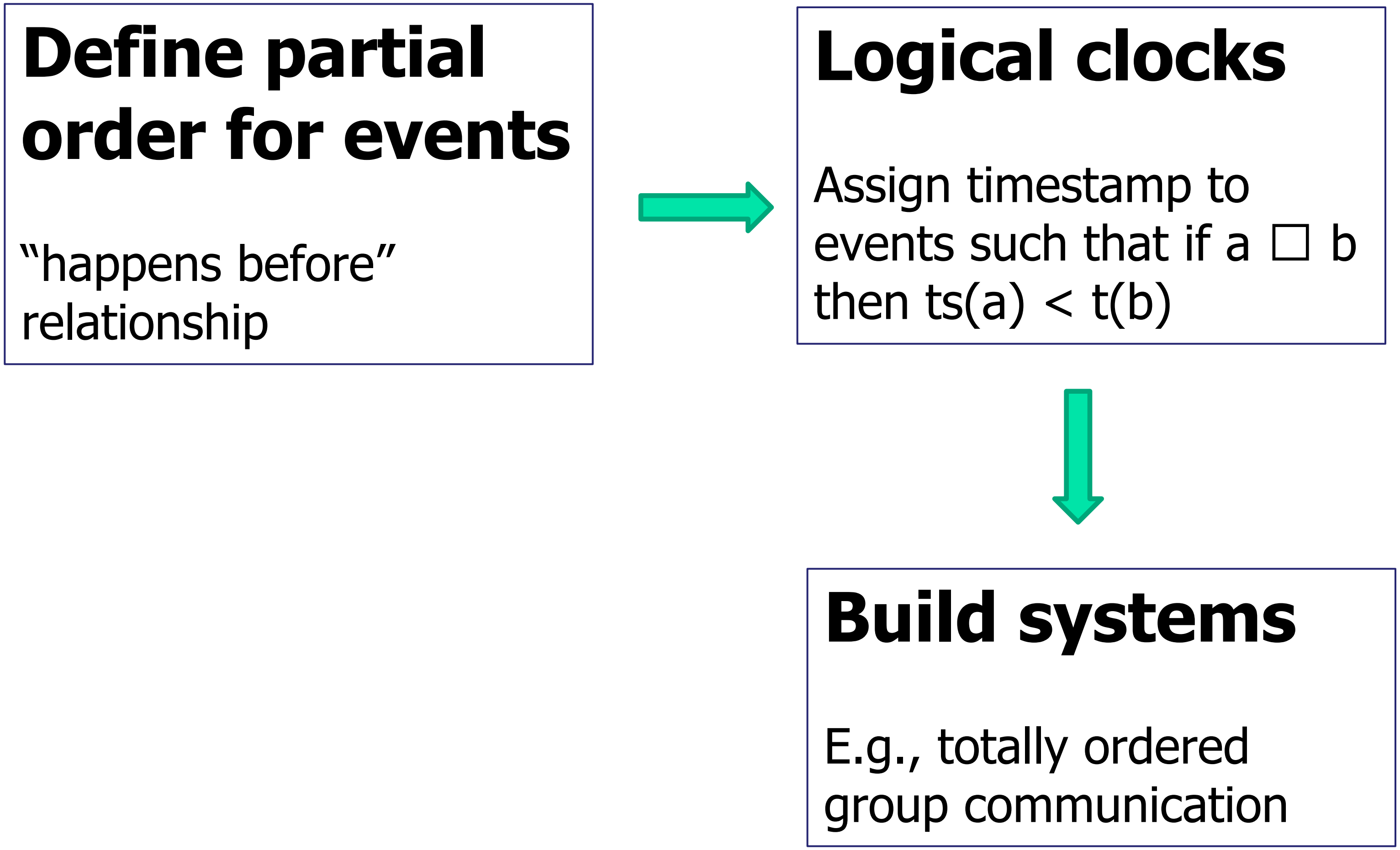
- Processes increment their clocks upon receiving/sending new messages and based on other processes' clocks (carried with

Example



concurrent events can not be detected based on Lamport clock info

Note: Lamport Timestamps: 3 < 7, but event with timestamp 3 is concurrent to event with timestamp 7, (events are not in 'happen-before' relation).



Define partial order for events

"happens before" relationship

Logical clocks

Assign timestamp to events such that if a □ b then ts(a) < t(b)

Build systems

E.g., totally ordered group communication

One example use

\big_detour{start}

Mutual exclusion

[see separate slide set]

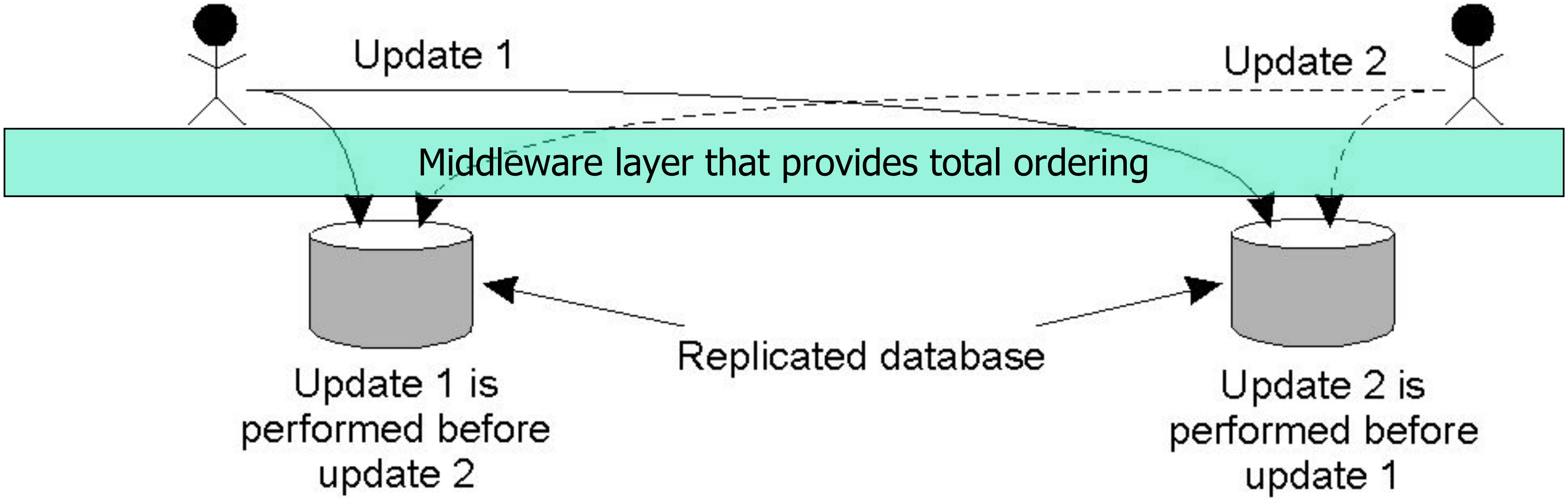
\big_detour{end}

Example II – replicated state machine

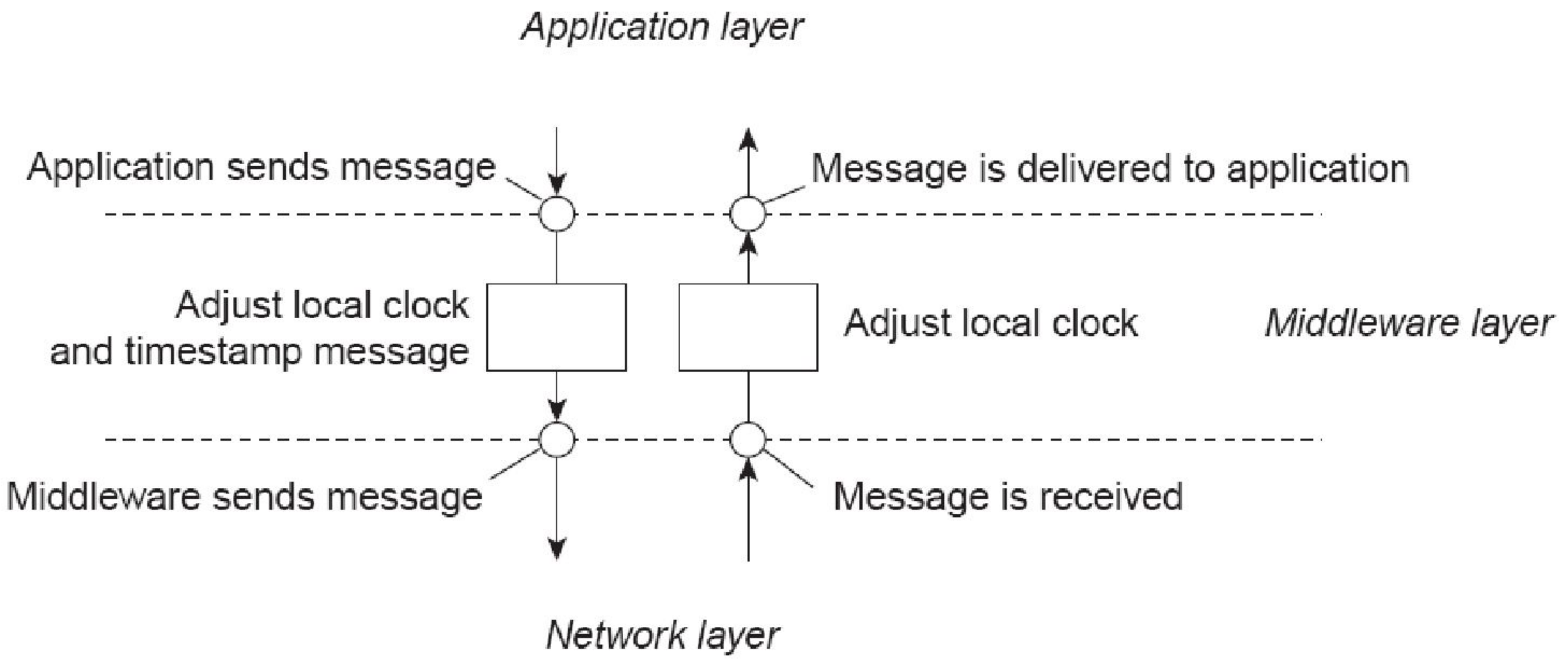
Two accounts:

- Initial state: $100 account balance
- Update 1: add $100
- Update 2: add 1% monthly interest

Updates need to be performed in the same order at the two replicas!


Update 1
Update 2
Middleware layer that provides total ordering
Replicated database
Update 1 is performed before update 2
Update 2 is performed before update 1

Architectural view



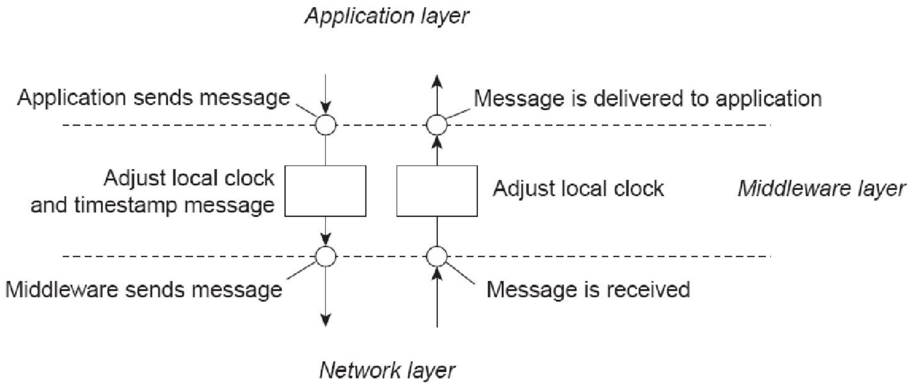
Middleware layer in charge of:

- Local management of logical clocks
  - i.e., stamping messages with (logical) clock times, updating timestamps at message receival,
- Message ordering: e.g. by delaying delivery/buffering (if needed)

Totally ordered group communication (cont)

Setup (for simplicity)

- All members are both senders and receivers
- A one-to-many communication substrate
- FIFO / Reliable Channels
- No Side Channels



Sketch of a solution:

- Middleware at each member maintains an ordered queue of received messages that have not yet been delivered to the application.
- **Main issue**: when to deliver to application such that, at all endpoints, messages are delivered in the same order.
- Each message is timestamped with local logical time then multicasted
  - When multicasted, also message logically sent to the sender itself
- When receiving a message, the middleware layer
  - Adds message to local queue (ordered by sending timestamp)
  - Acknowledges (using multicast) the message
  - Delivers from top of queue to application only when **all** acks for message on top have been received (or optimization: see next slide)

Totally Ordered Multicast – Algorithm

Sending / Receiving

- Process P_i sends timestamped message msg_i to all others. The message itself is put in the local $queue_i$ as well
- Any incoming message msg_i *received* at P_k is queued in $queue_k$ according to its sent timestamp, and acknowledged to every other process.

Deliver to application

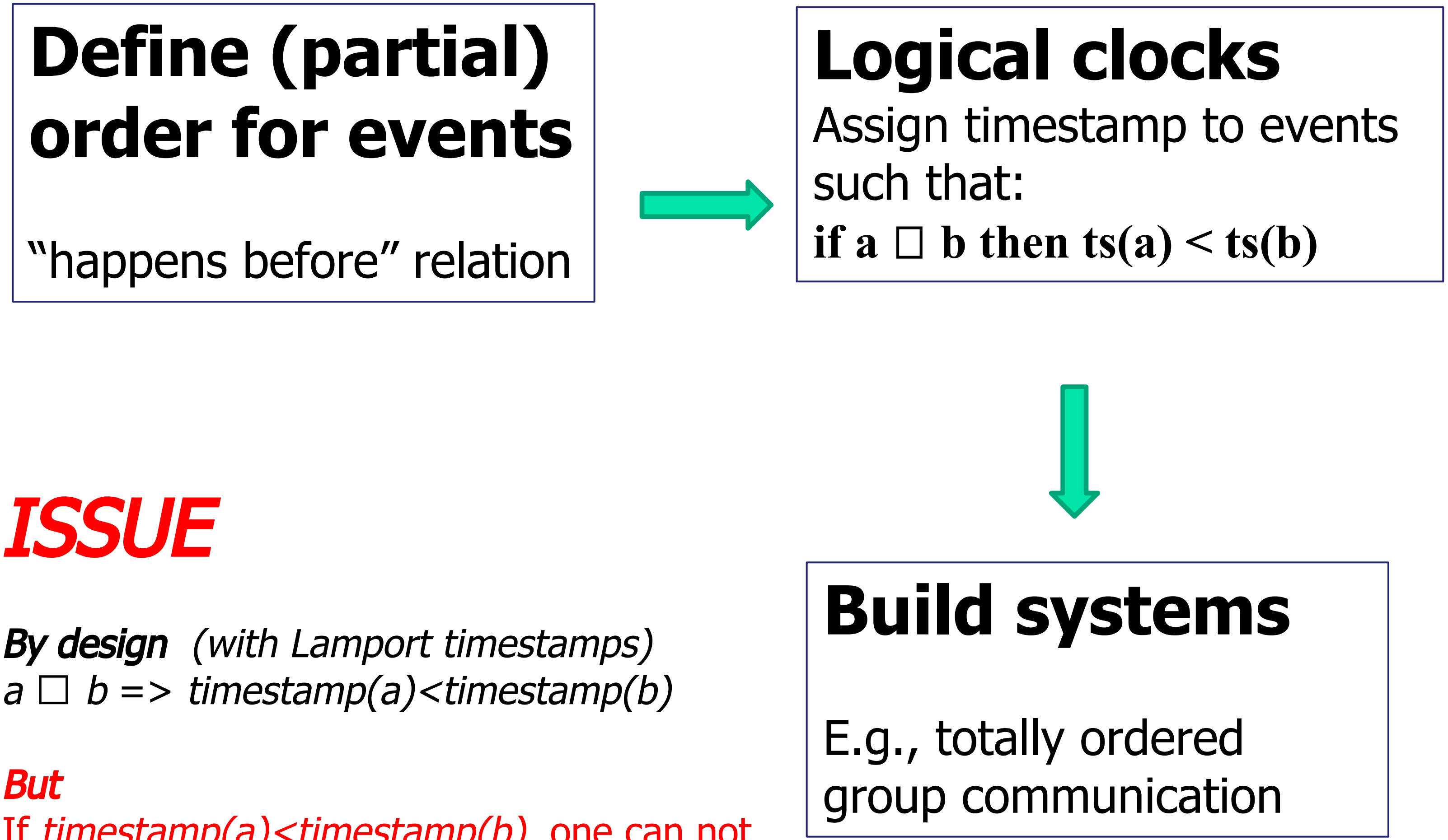
- P_k delivers a message msg_i to its application if:
  - msg_i is at the head of $queue_k$ and
  - for each process P_x, there is an ack or a message in $queue_k$ with a larger sending timestamp.

Guarantee: all messages are delivered in the same order at all destinations

Note: We assume that communication is reliable and FIFO ordered.

Quiz-Like Questions

- What’s the complexity of the protocol in terms of number of messages
- What happens if we drop channel reliability assumption?
    - Does the protocol still work? If it fails, explain how.
- What happens if we drop channel FIFO assumption?
    - Does the protocol still work?
    - How would you change the previous protocol to still work correctly without this assumption?
- Assume you have a bound on message propagation time in the network. Design a protocol that provides total ordering (and generates less traffic)



Define (partial) order for events

"happens before" relation

Logical clocks

Assign timestamp to events such that:

if a □ b then ts(a) < ts(b)

Build systems

E.g., totally ordered group communication

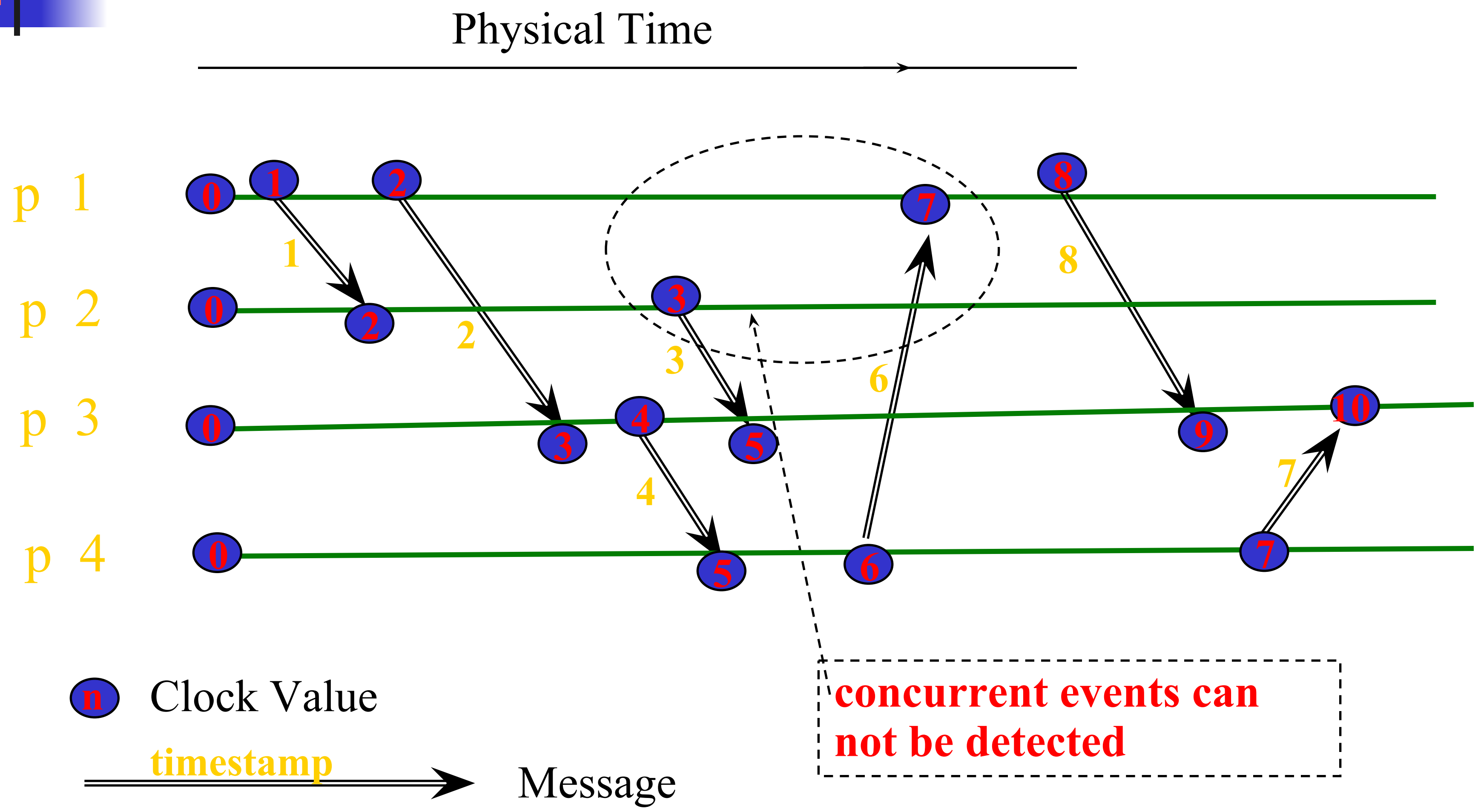
ISSUE

By design *(with Lamport timestamps)*
a □ b => timestamp(a)<timestamp(b)

But
If *timestamp(a)<timestamp(b)* one can not reason about relative ordering of *a* and *b*

[the only thing you know is that b□a is FALSE]

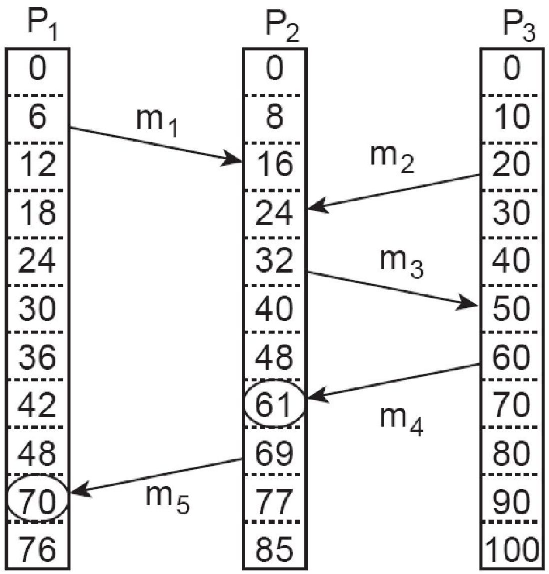
Example



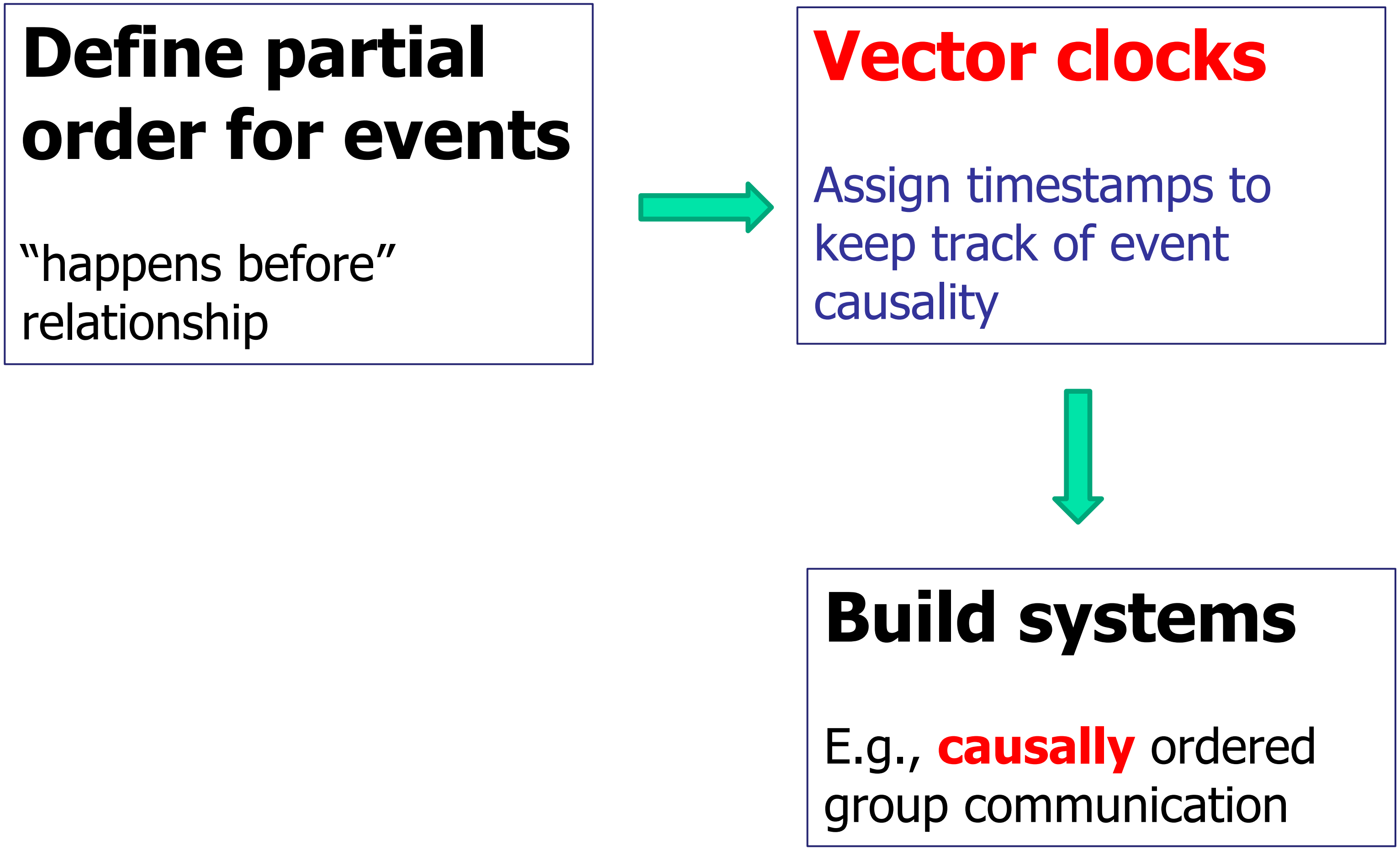
concurrent events can not be detected

Issue: Lamport Timestamps 3 < 7, but event with timestamp 3 is concurrent to event with timestamp 7, i.e., events are not in ‘happen-before’ relation.

Causality



- Isue: Lamport timestamps don't properly capture causality
  - Introduce more ordering than necessary
- Applications often need to reason about (i.e., order similarly) only causally related messages.
  - Example: news postings have multiple independent threads of messages.
    - What are the constraints on ordering?
- To model causality – **vector** timestamps
  - **Intuition**: each item in vector logical clock for one causality thread.



Define partial order for events

"happens before" relationship

Vector clocks

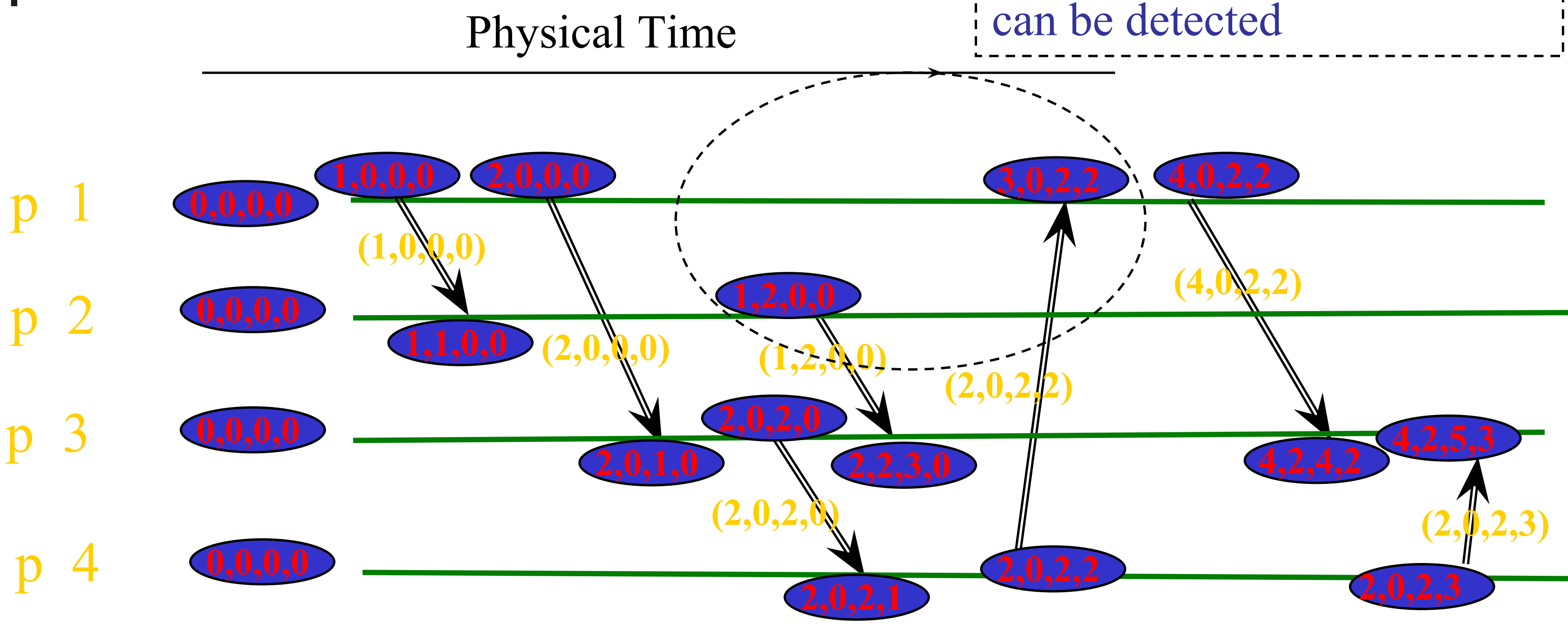
Assign timestamps to keep track of event causality

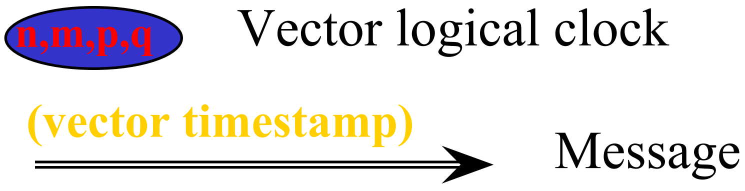
Build systems

E.g., **causally** ordered group communication

Example: Vector Timestamsps

logically concurrent events can be detected





Notation

- $VT_1 < VT_2$,

  iff forall j ($1 \leq j \leq n$) such that $VT_1[j] \leq VT_2[j]$

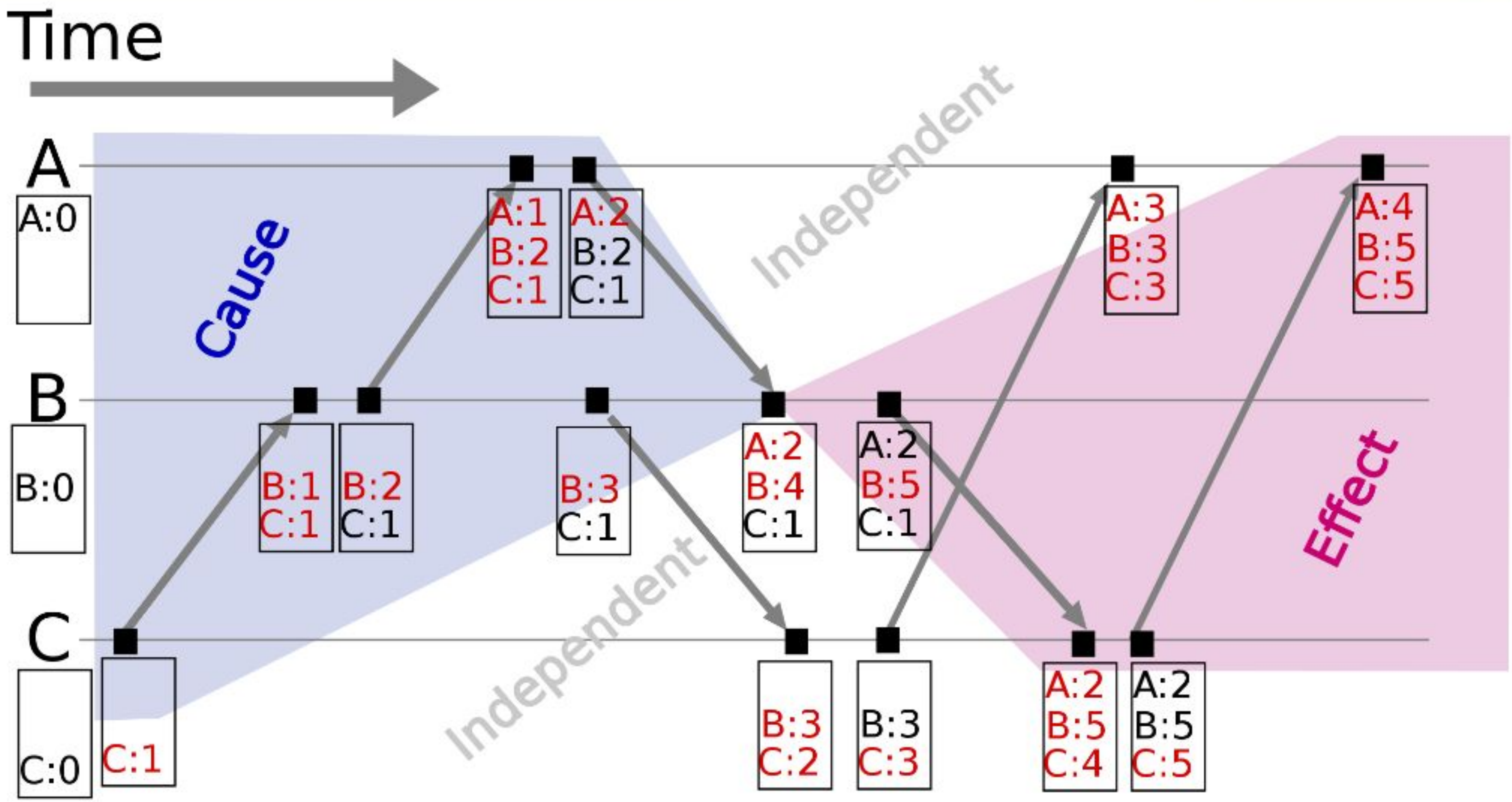
  and

  exists j ($1 \leq j \leq n$) such that $VT_1[j] < VT_2[j]$

- VT_1 is concurrent with VT_2

  iff (not $VT_1 < VT_2$ AND not $VT_2 < VT_1$)

Vector clocks/timestamps enable reasoning about causality



Events in the blue region are the causes leading to event B4, whereas those in the red region are the effects of event B4

Vector clocks: the formal definition

- Each process P_i has an array $VC_i\,[1..n]$ of clocks (all initially at 0)
  - $VC_i\,[j]$ denotes the number of events that process P_i knows have taken place at process P_j.
- P_i increments $VC_i\,[i]$: when an event occurs
  - local event, message sending, message receiving
  - timestamp of the event is vector value

- When sending
  - Messages sent by VC_i includes a **vector timestamp** $vt(m)$.
  - Result: upon arrival, recipient knows P_i's timestamp.
- When P_j receives a msg from P_i with vector timestamp $ts(m)$:
  - for k ≠ j: update each $VC_j\,[k]$ to $\max\{VC_j\,[k], ts(m)[k]\}$

Note: vector timestamps require a static notion of system membership

Comparing vector timestamps

Notation

- $VT_1 < VT_{2}$,
  iff forall j ($1 \leq j \leq n$) such that $VT_1[j] \leq VT_2[j]$
  and
  exists j ($1 \leq j \leq n$) such that $VT_1[j] < VT_2[j]$

- VT_1 is concurrent with VT_2
  iff (not $VT_1 < VT_2$ AND not $VT_2 < VT_1$)

Quiz like problem

Show that:

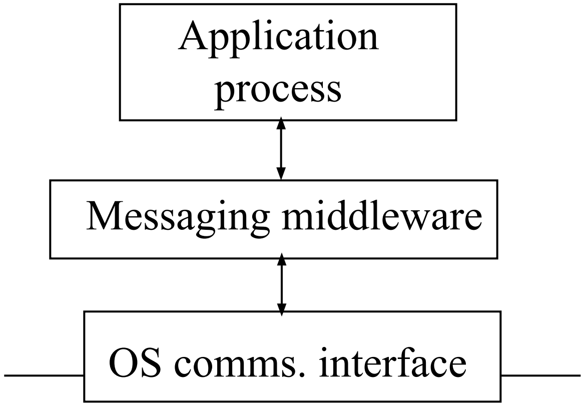
a □ b if and only if ***vectorTS(a) < vectorTS(b)***

Extending group communication

ASSUMPTIONS

- messages are multicast to (named) process groups
- reliable and FIFO channels
- processes don't crash
- processes behave as specified (i.e., we are not considering Byzantine behaviour)

Architectural view



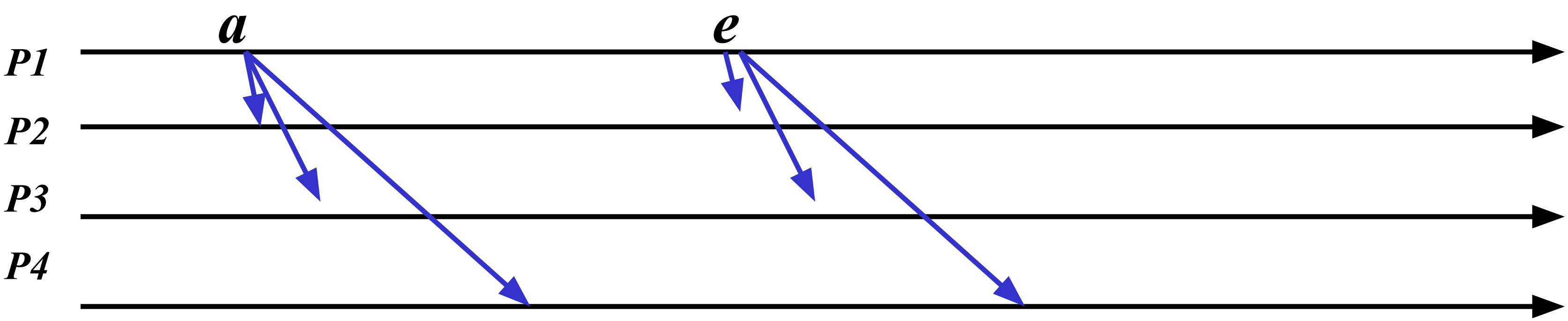
<u>Application:</u> chooses delivery order of message service
e.g. **total order, FIFO order, causal order**
(last time we looked at 'total order')

Middleware: may reorder/delay message delivery to application by buffering messages to implement reordering policy

Basic network layer: provides reliable FIFO from each source (done at lower levels)

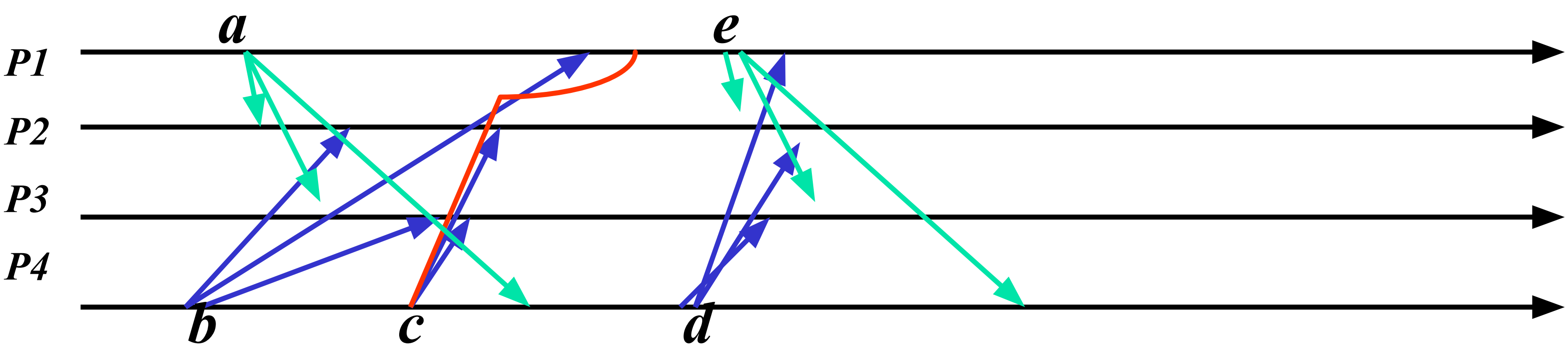
FIFO multicast

- *FIFO* (or *sender ordered*) multicast: *Messages sent by any single sender are received in the same order*
  - Nothing guaranteed for relative ordering of messages sent by two different senders



FIFO multicast

- *FIFO* (or *sender ordered*) multicast: *Messages sent by any single sender are received in the same order*
  - Nothing guaranteed for relative ordering of messages sent by two different senders



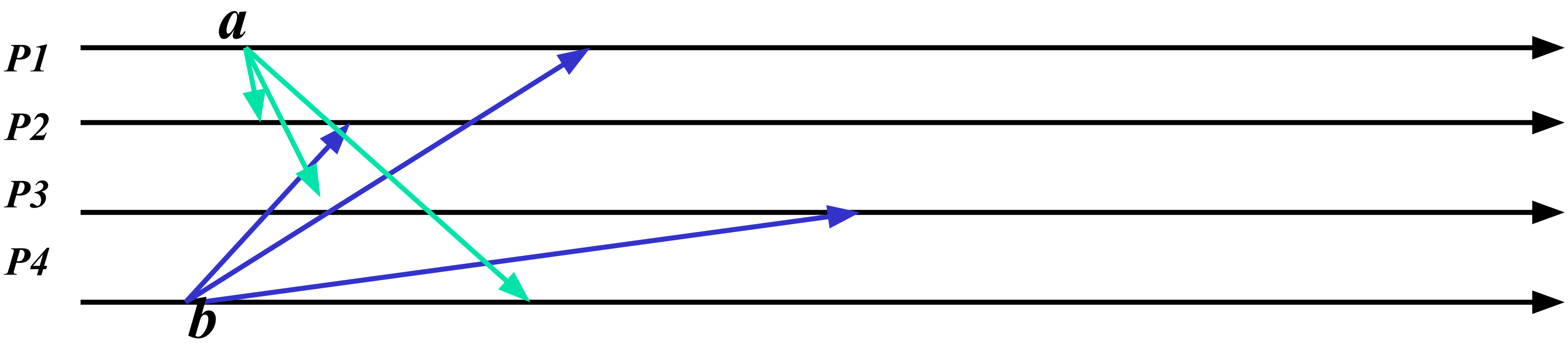
- *delivery of c to P1 is delayed until after b is delivered*
- *d and e are received in different order at P2 and P3*

Implementing FIFO multicast

- Basic reliable (i.e., no message loss) multicast has this property
  - Without failures all we need is to run it on FIFO channels (like TCP)
  - [Later: dealing with node failures]

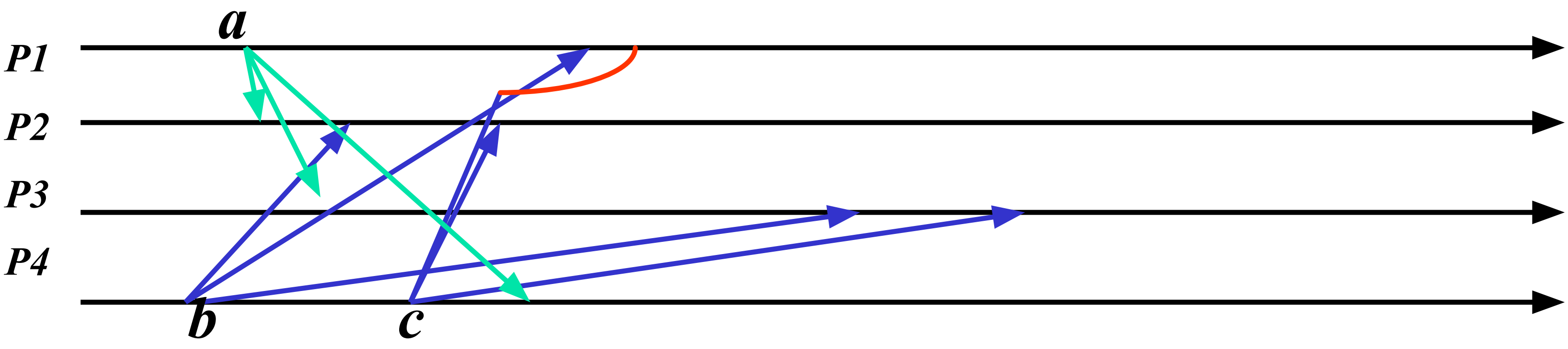
Causal multicast

- *Causal* (or *happens-before* ordering)
- *If send(a) → send(b) (i.e., there was some causal relationship)*
  - *then deliver(a) occurs before deliver(b) at common destinations*



Ordering properties: Causal

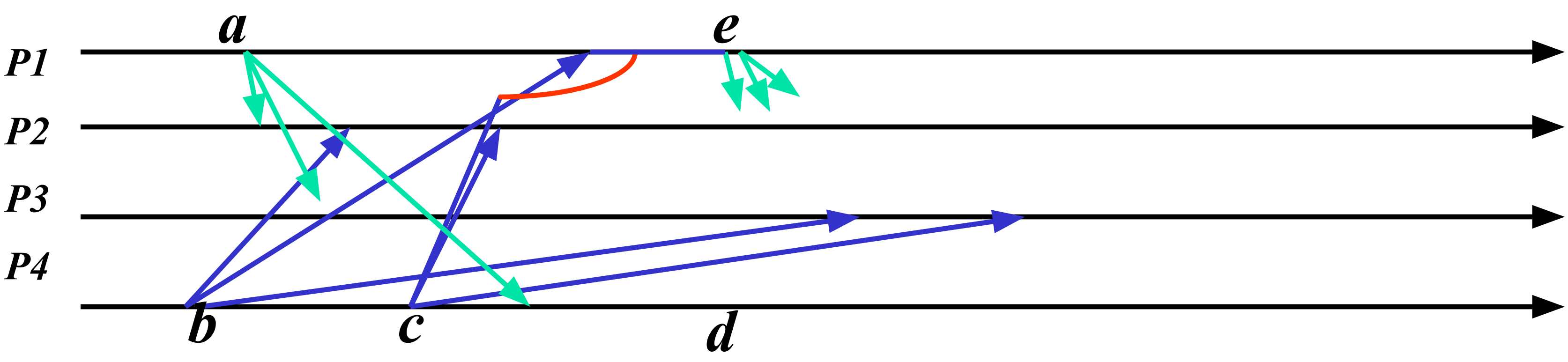
- *Causal* (or *happens-before* ordering)
- *If send(a) → send(b) (i.e., there was some causal relationship)*
  - *then deliver(a) occurs before deliver(b) at common destinations*



delivery of c to P1 is delayed until after b is delivered

Ordering properties: Causal

- *Causal* (or *happens-before* ordering)
- *If send(a) → send(b) (i.e., there was some causal relationship)*
  - *then deliver(a) occurs before deliver(b) at common destinations*



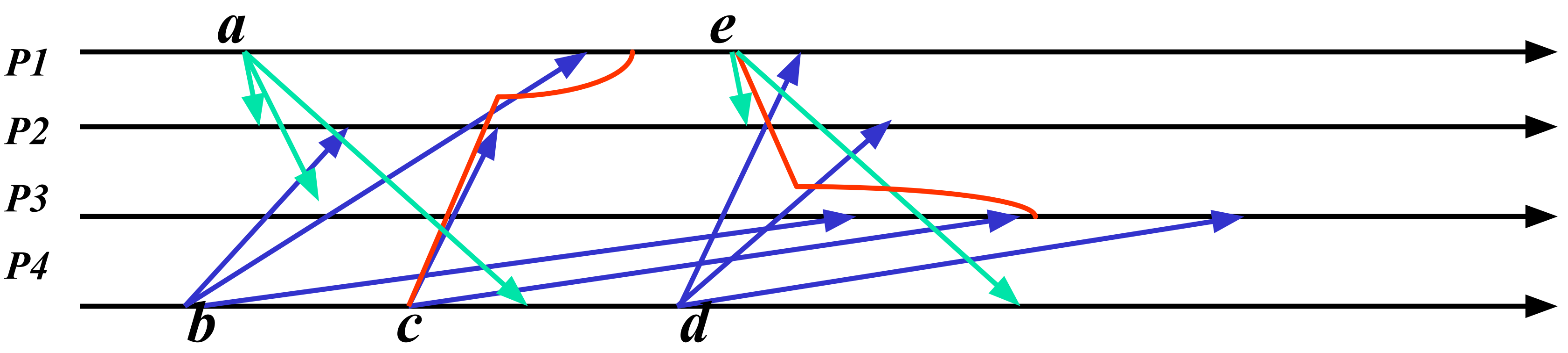
e is sent (causally) after b and c

e is sent concurrently with d

Ordering properties: Causal

- *Causal* (or *happens-before* ordering)
- *If send(a) → send(b)*

  (i.e., if there was some causal relationship) between a and b)
  - *then deliver(a) occurs before deliver(b) at common destinations*



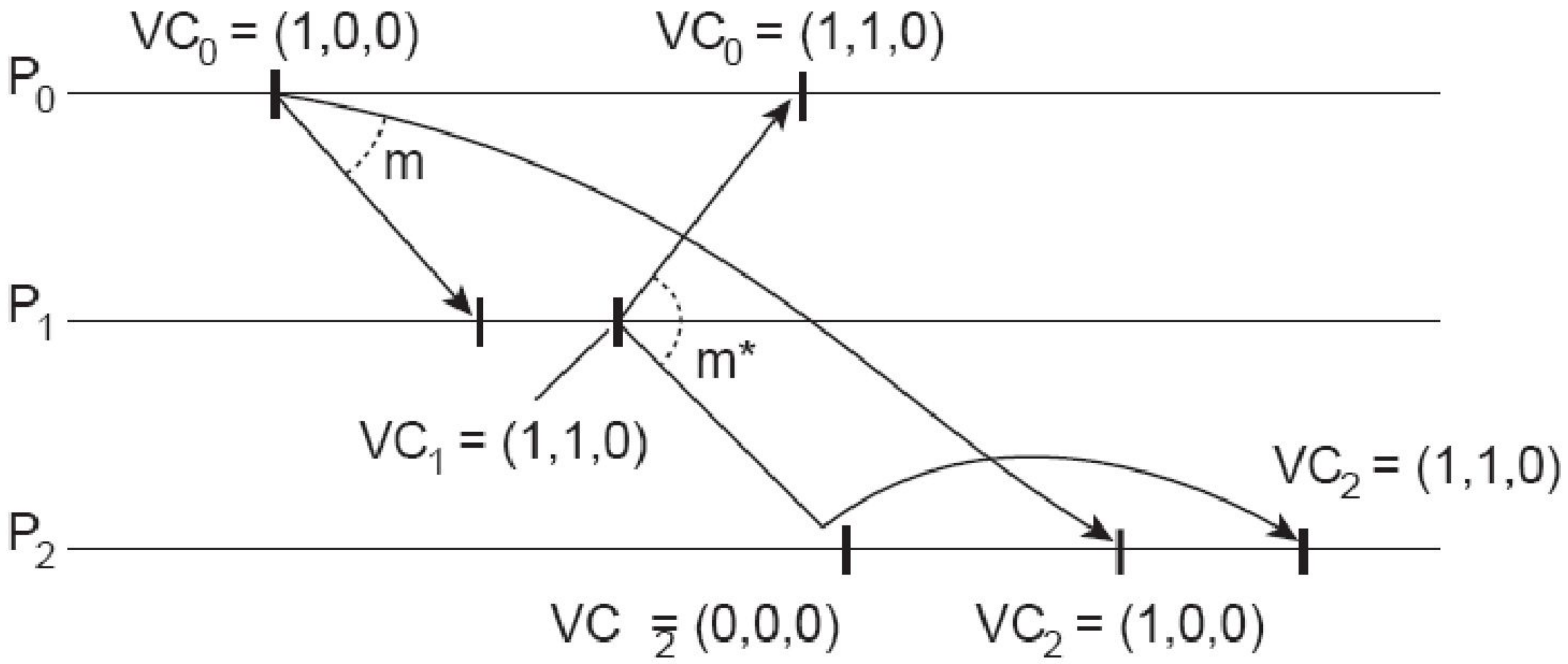
delivery of c to P1 is delayed until after b is delivered

delivery of e to P3 is delayed until after b&c are delivered

delivery of e and d to P2 and P3 in any relative order (concurrent)

Implementing causally ordered multicast

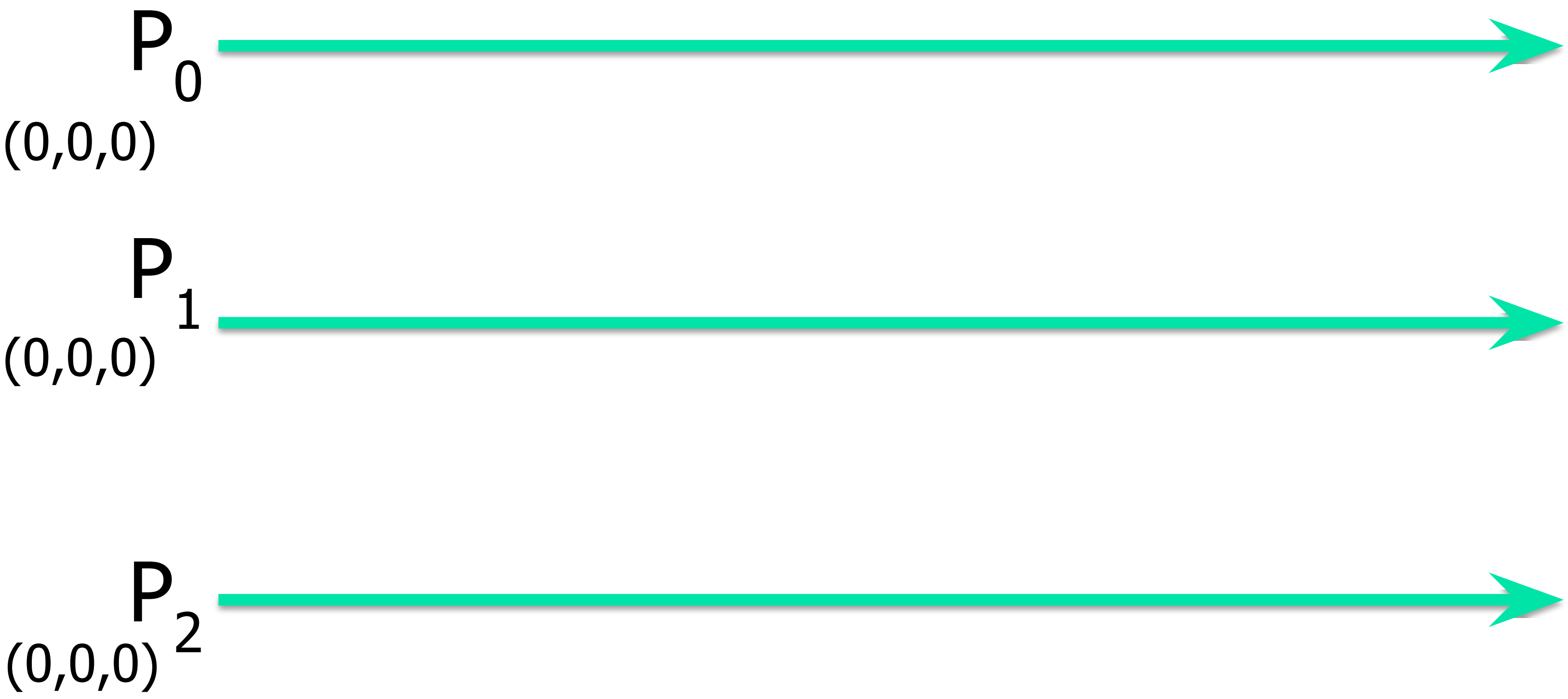
[note slightly different update rule for vector clocks to enable counting the number of messages sent by each machine]



Now suppose that P_j receives a message m from P_i with (vector) timestamp $ts(\mathrm{m})$. The delivery of the message to the application layer will then be delayed until the following two conditions are met:

1. $ts(\mathrm{m})[i] = \mathrm{VC}_j[i] + 1$
2. $ts(\mathrm{m})[k] \leq \mathrm{VC}_j[k]$ for all $k \neq i$

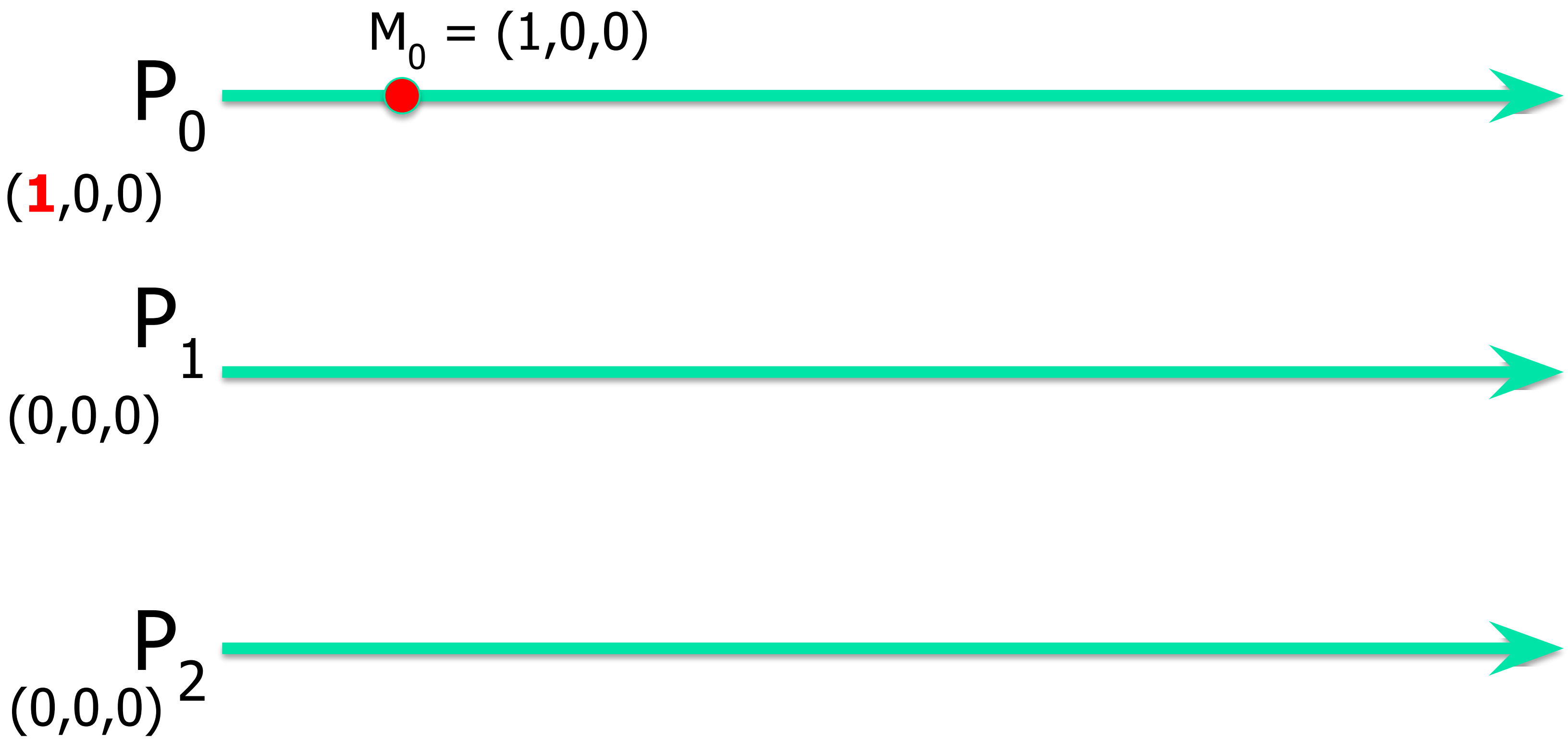
Implementing causally ordered multicast



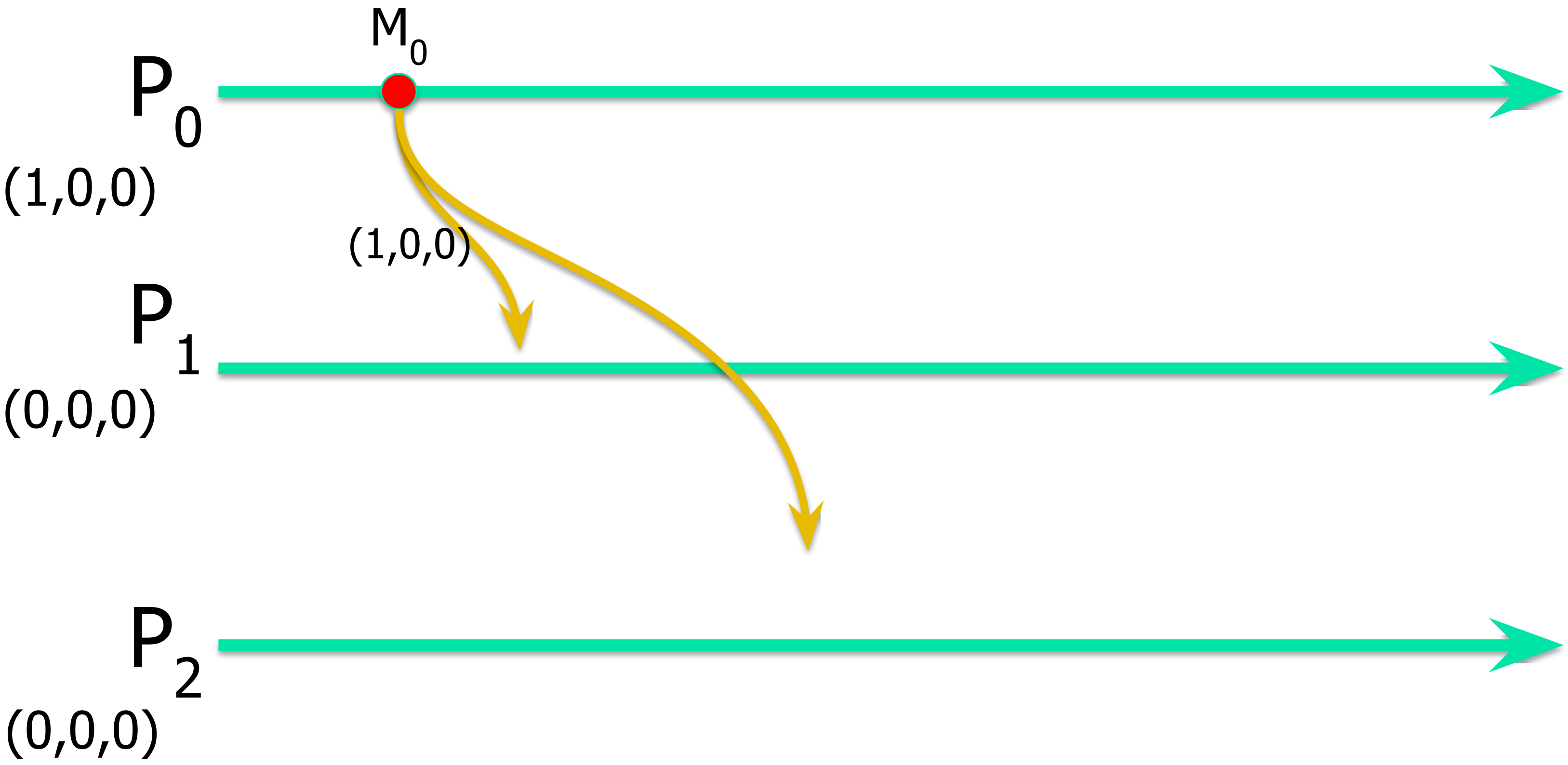
Vector clocks on each process all initialized at (0,0,0)

At process $\boldsymbol{P_i}$ position i will count how many messages $\boldsymbol{P_i}$ has sent

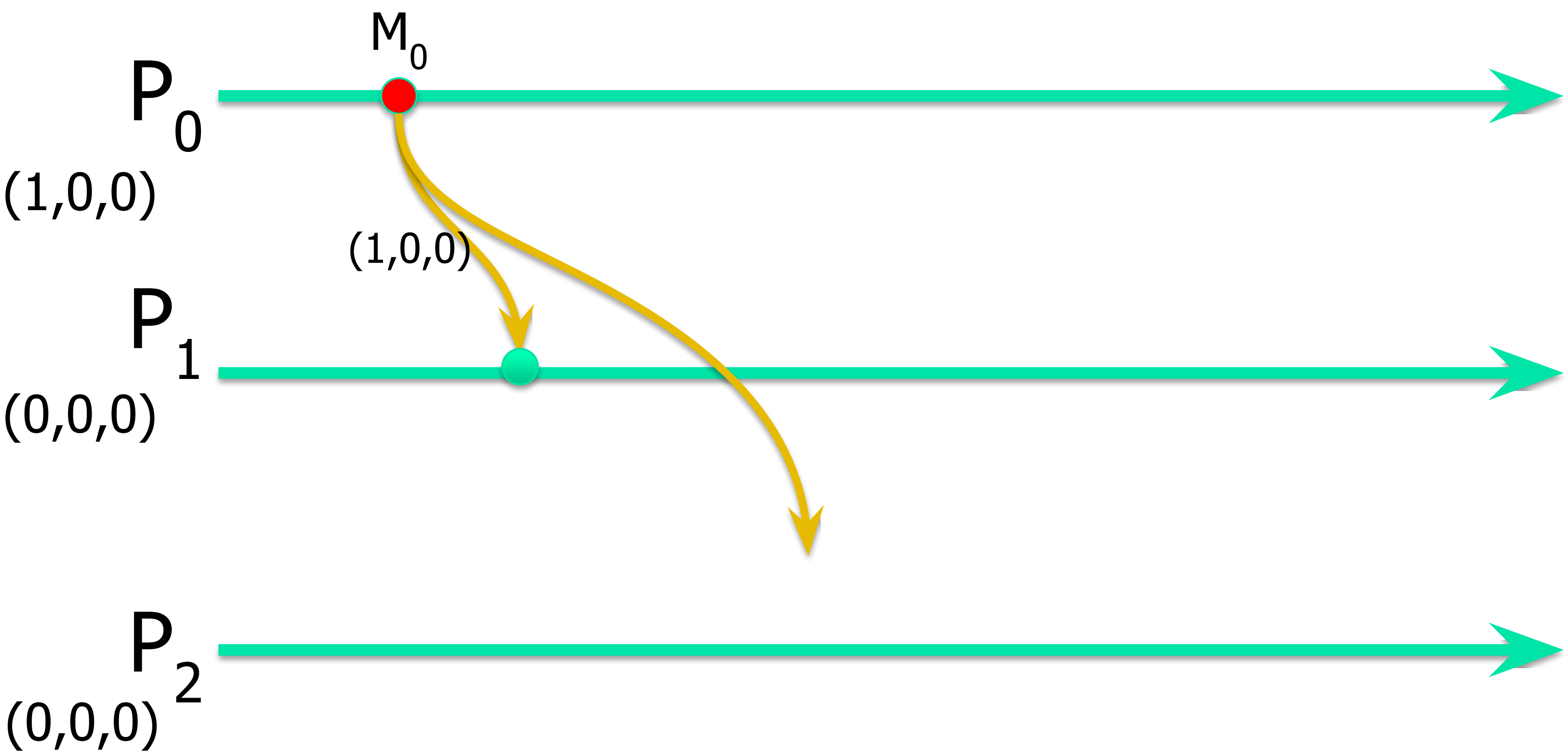
position $k \neq i$ will count how many messages $\boldsymbol{P_i}$ has received from $\boldsymbol{P_k}$



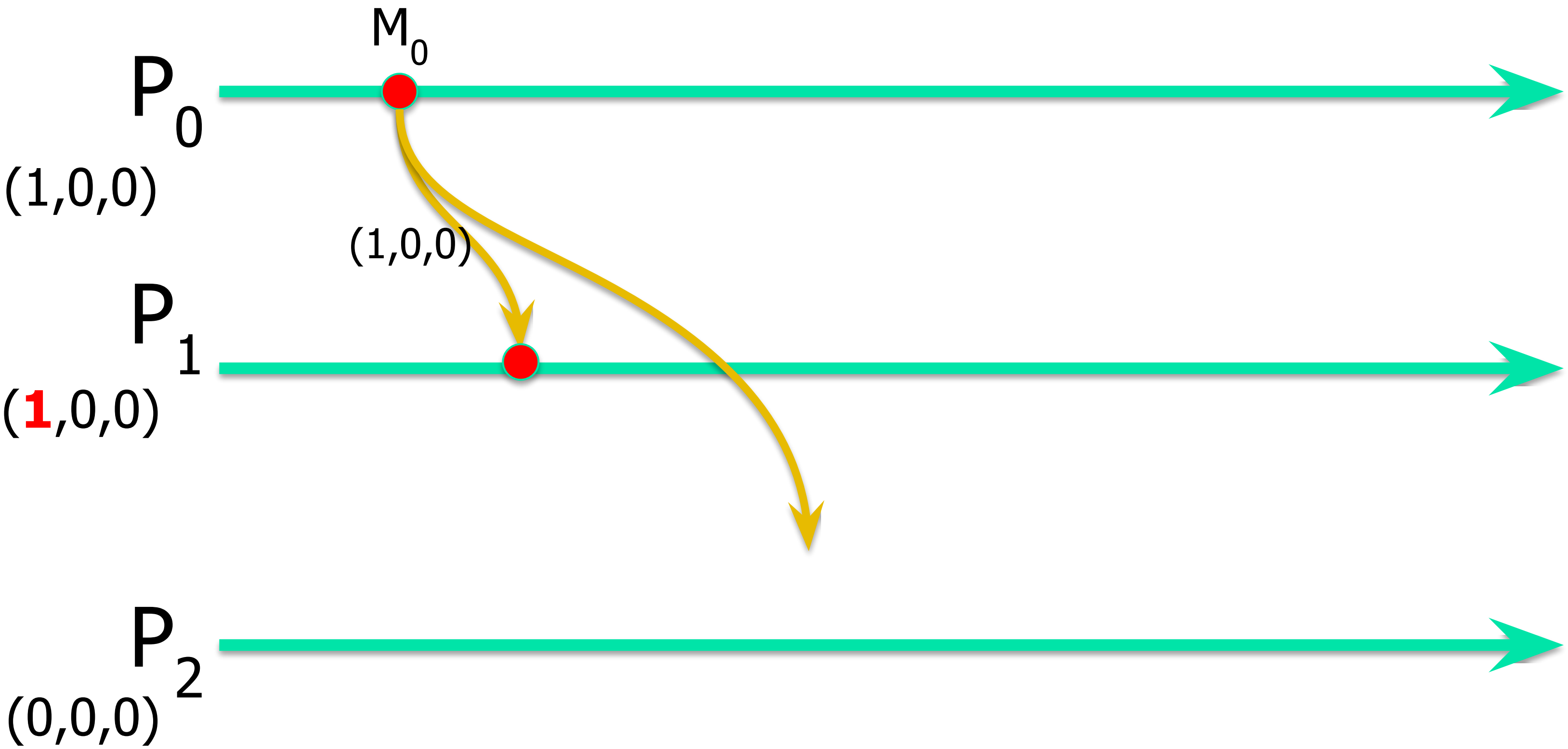
- Message Mo is initiated at Po and can be delivered to application locally.
- Vector clock at Po is updated.
- Mo is sent with its vector clock



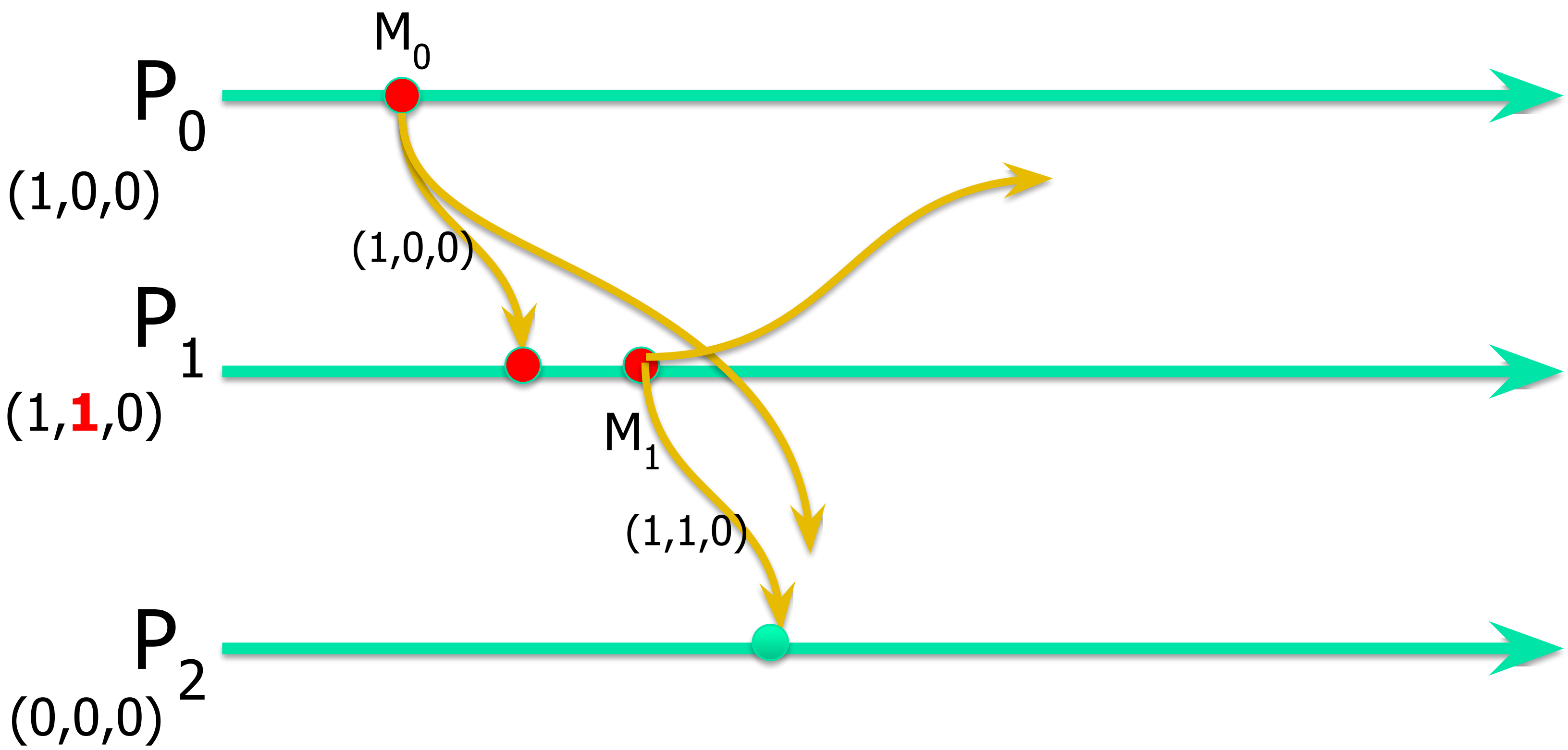
- Message Mo is initiated at Po and can be delivered to the application locally.
- Vector clock at Po is updated.
- Mo is sent with its vector clock to the other processes



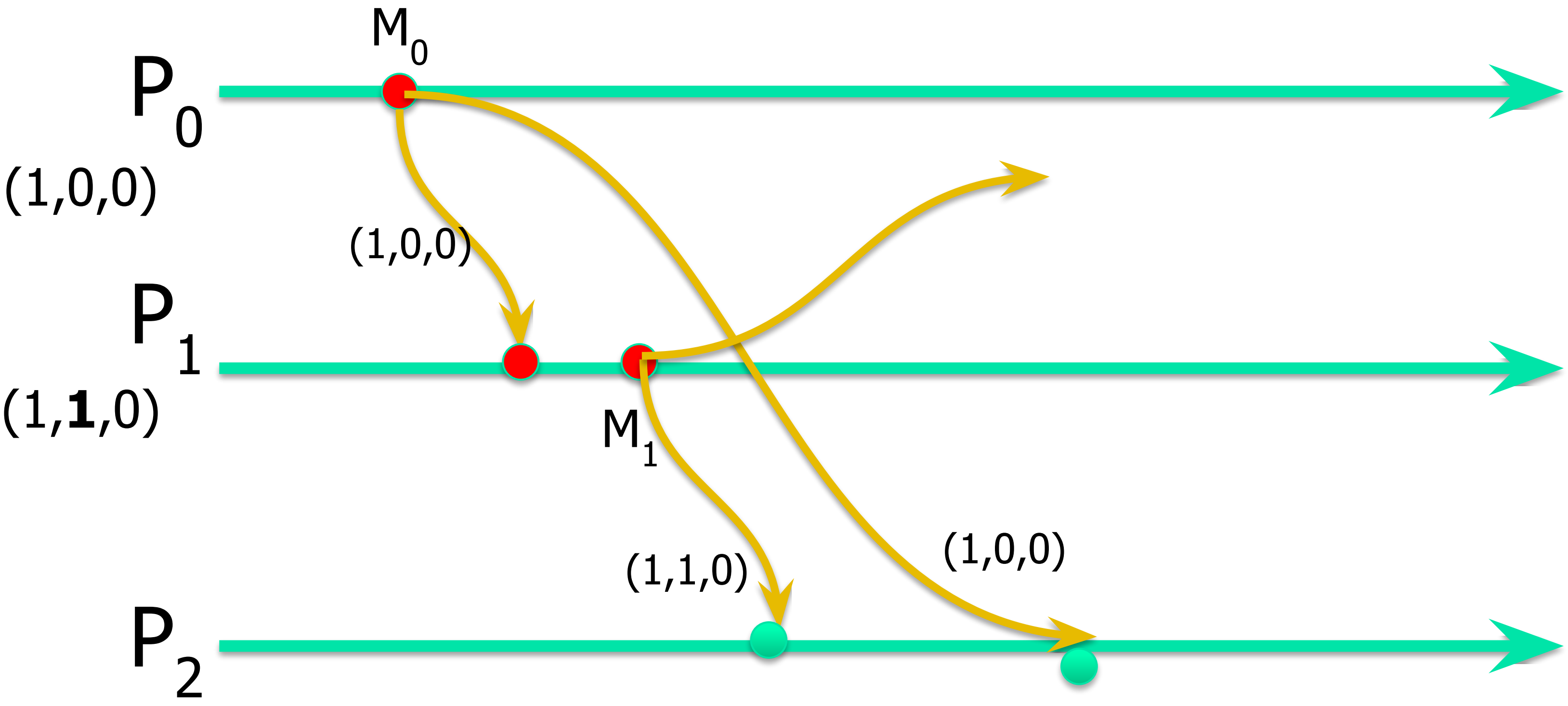
- Message Mo arrives at P1
- Message Mo can be delivered to the application at P1: the check - all messages that have potentially been received at Po before generating Mo have been seen by P1 as well



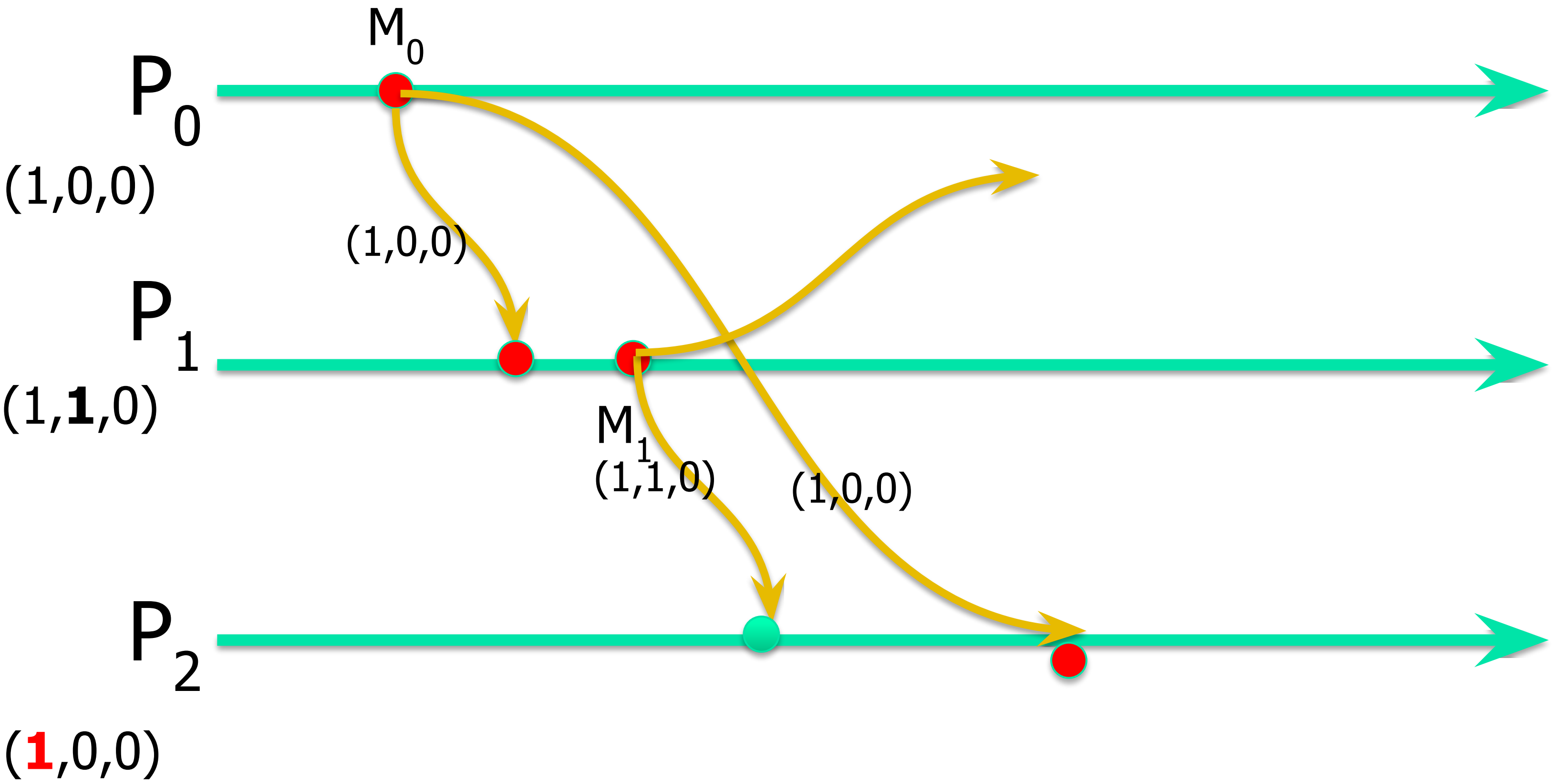
- Message Mo is delivered at P1
- P1 updates its vector clock



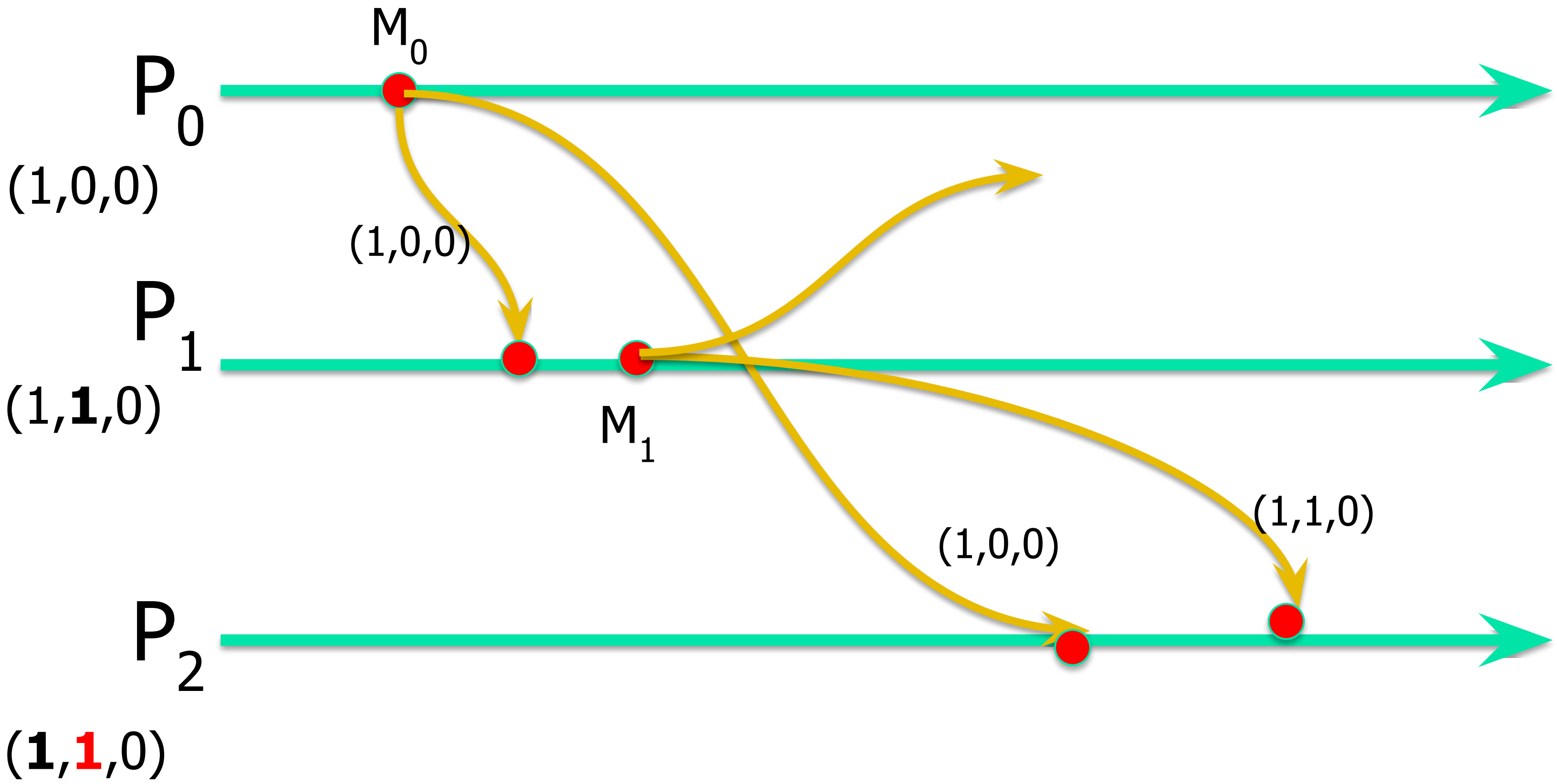
- Message M1 is created by P1. The message can be delivered to local application at P1. P1 updates its vector clock. This timestamp (1,1,0) is carried by M1.
- Message M1 arrives at P2. Can not be delivered to P2 since, M1's vector clock at position 0 implies that, when M1 was created it has one message from Po and P2 has not yet seen that message.



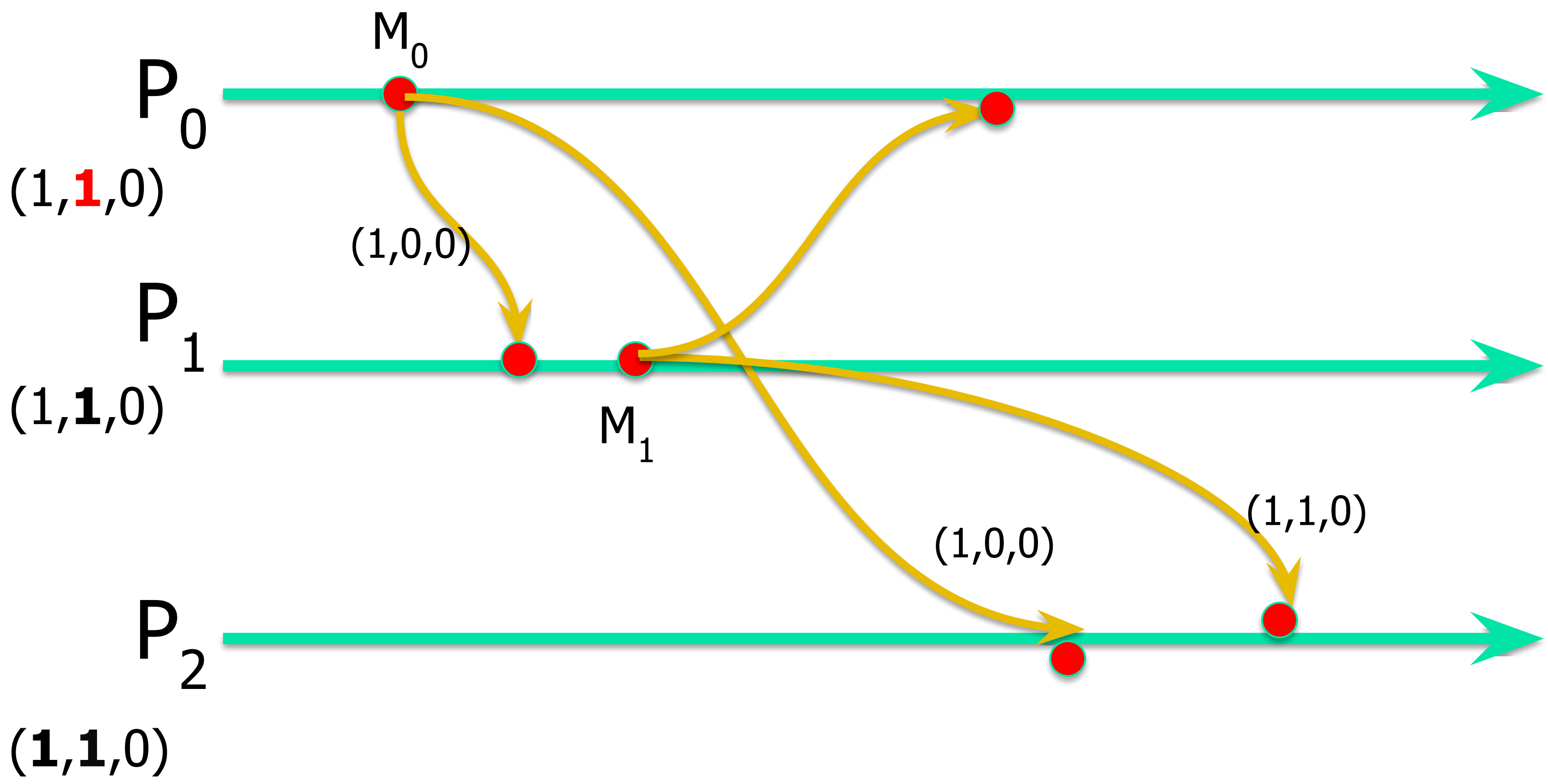
- Message Mo arrives at P2. Can be delivered to P2 (same argument)



- Message Mo is delivered to P2. P2’s vector clock is updated.
- Now P2 can look at the other messages it has in its queue



- Now Mo can be delivered to application at P2. P2’s vector clock is updated.



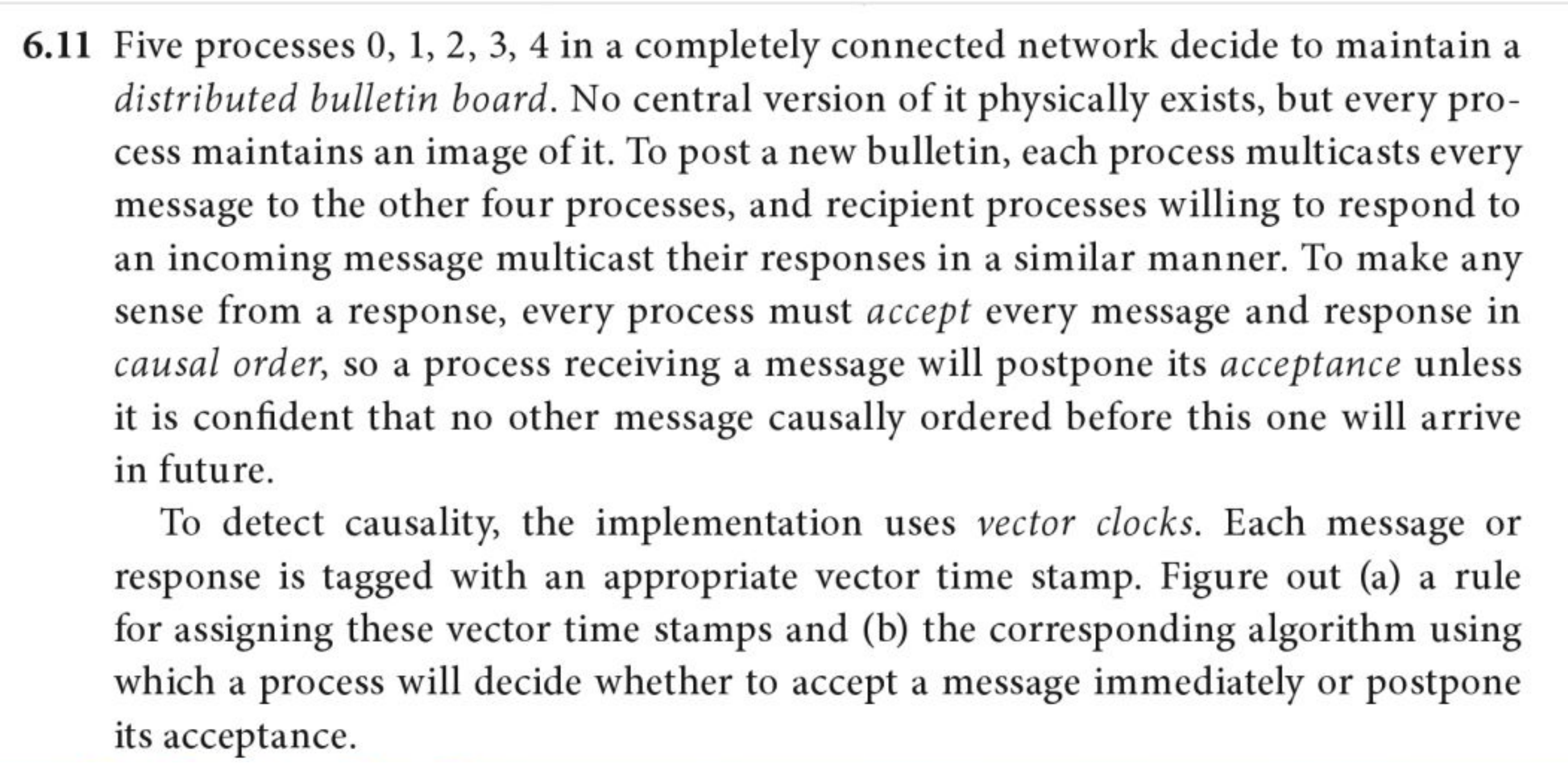
- M1 arrives at Po and can be delivered to application. Po updates its vector clock

Implementing causal order multicast

- Start with FIFO multicast
- Strengthen this into a causal multicast by adding vector time

- **Advantages** (compared with totally ordered multicast)
  - No additional messages needed!
    - Lower overhead
  - causal multicast (as well as FIFO multicast) are *asynchronous:*
    - Sender doesn’t get blocked and can deliver a copy to itself without “stopping” to learn a safe delivery order

Sample quiz question



6.11 Five processes 0, 1, 2, 3, 4 in a completely connected network decide to maintain a *distributed bulletin board*. No central version of it physically exists, but every process maintains an image of it. To post a new bulletin, each process multicasts every message to the other four processes, and recipient processes willing to respond to an incoming message multicast their responses in a similar manner. To make any sense from a response, every process must *accept* every message and response in *causal order*, so a process receiving a message will postpone its *acceptance* unless it is confident that no other message causally ordered before this one will arrive in future.

To detect causality, the implementation uses *vector clocks*. Each message or response is tagged with an appropriate vector time stamp. Figure out (a) a rule for assigning these vector time stamps and (b) the corresponding algorithm using which a process will decide whether to accept a message immediately or postpone its acceptance.

So far …

- Physical clocks
  - Two applications
    - Provide at-most-once semantics
    - Global Positioning Systems
- ‘Logical clocks’
  - Where only ordering of events matters

- Next:
  - Replication
  - One application that puts everything together