Time, Clocks and Ordering of Events

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

Overview

- Clock synchronization
- Logical clocks
- Vector clocks

Time on a single node

- Programs often need to determine or compare time
  - E.g., log timestamped events, set a timeout, profile programs
- Programs running on a node depend on a single physical clock to determine time
- But what about programs running across nodes?

Need for clock synchronization

- Each node has a separate physical clock
- Different clocks may have skew and drift
  - Skew: instantaneous time difference between two clocks
  - Drift: rate at which clock skew increases between two clocks
- Programs running on multiple nodes expect clocks to be synchronized, i.e., no skew (or drift)
  - E.g., order timestamped events in the log,
    handle crashed clients using timeouts, etc.
- To synchronize clocks, we need to first understand characteristics of physical clocks

Physical clocks

- Let’s look at three types of physical clocks
  - Computer clocks
  - Atomic clocks
  - UTC radio/satellite receivers

Computer clocks

- Typical computers have a battery-backed clock that uses a quartz crystal that oscillates at a defined frequency
  - Oscillator is cheap (<$1) but sensitive to temperature, age, vibration, radiation
  - May drift by as much as a second per day!

Atomic clocks

- Caesium atomic clocks provide high accuracy
  - Best accuracy is roughly 1 sec per 100 million years!
- Compact chip-scale atomic clock (CSAC) available commercially, roughly 4cm x 4cm x 1cm
  - Expensive, currently available for $5000

UTC radio/satellite receivers

- Universal Coordinated Time (UTC) is a time standard that uses the weighted average of hundreds of atomic clocks installed worldwide
- UTC is broadcast from radio stations, GPS satellites
- A radio/satellite receiver can acquire UTC
  - GPS receivers can provide accuracy to ~1 microsecond
  - But high-end receivers are expensive and don’t work indoors

Physical clocks: recap

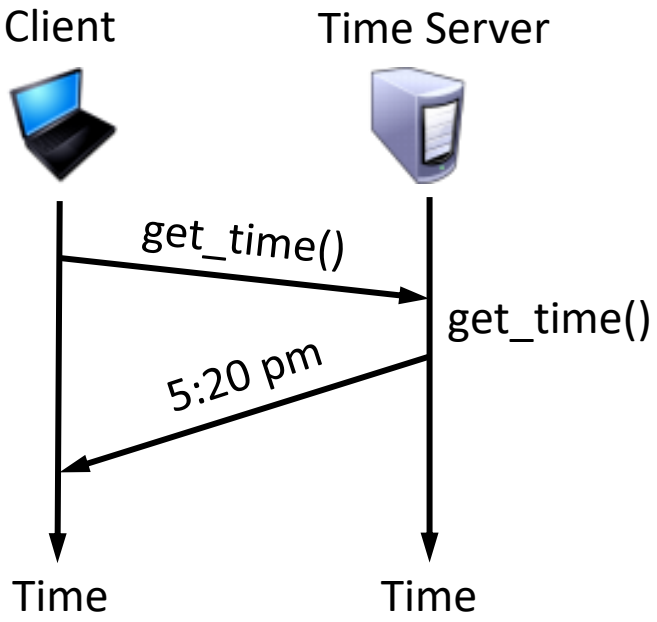
- Let’s look at three types of physical clocks
  - Computer clocks: cheap but high drift
  - Atomic clocks: very accurate, but expensive
  - UTC radio/satellite receivers: accurate, but still expensive

Key idea for synchronizing clocks

- Use a small number of time servers
  - Fitted with atomic clock or UTC receiver hardware
- Use a distributed protocol to synchronize the clock on a node with the time servers

Strawman for clock synchronization

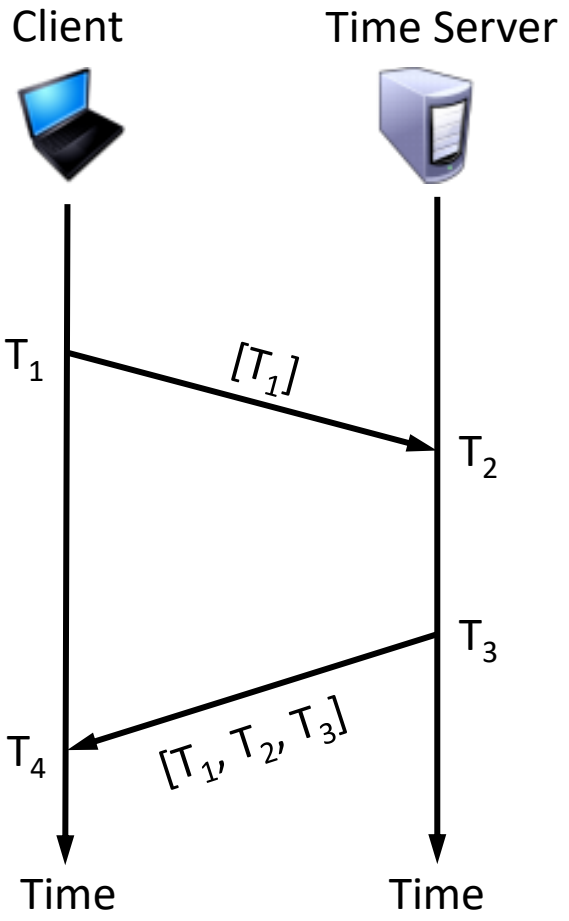
- Client issues RPC to get current time from time server
- Sets computer clock to time received from time server



- Problem: receiving a messages has (varying) delay, response has outdated time

Cristian's algorithm: RPC message

1. Client sends request packet, timestamped with its clock time T_1
2. Server timestamps its receipt of request with its clock time T_2
3. Server sends response packet with its clock times T_3 and T_2
4. Client timestamps its receipt of response with its clock time T_4

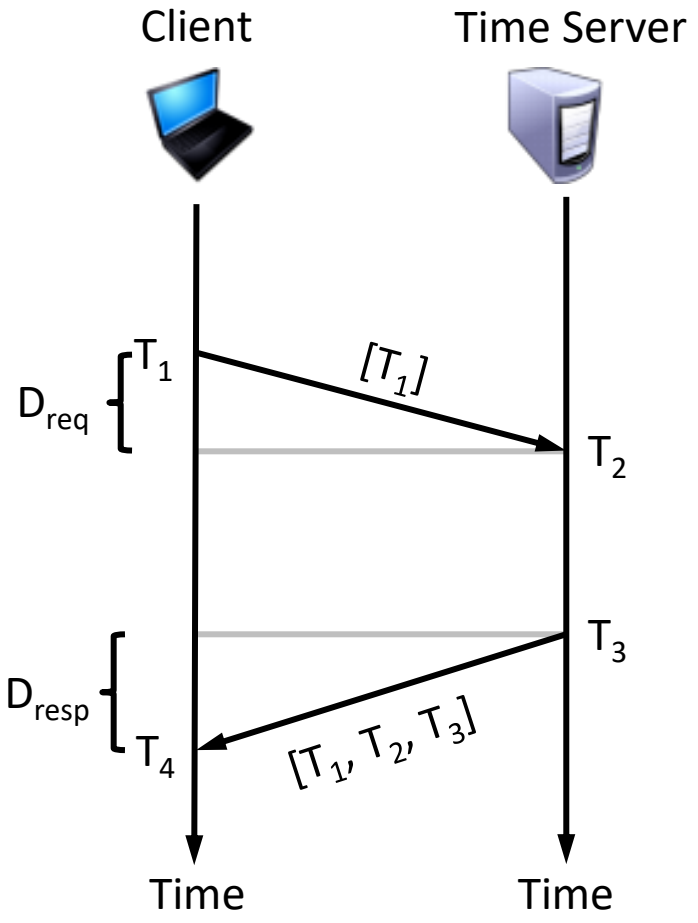


Then client uses the $[T_1,T_2,T_3,T_4]$ tuple to synchronize its clock with time server

Cristian's algorithm: clock sync

- Client should set its clock to T_3+D_{resp}
- Round trip delay $D = D_{req}+D_{resp}$

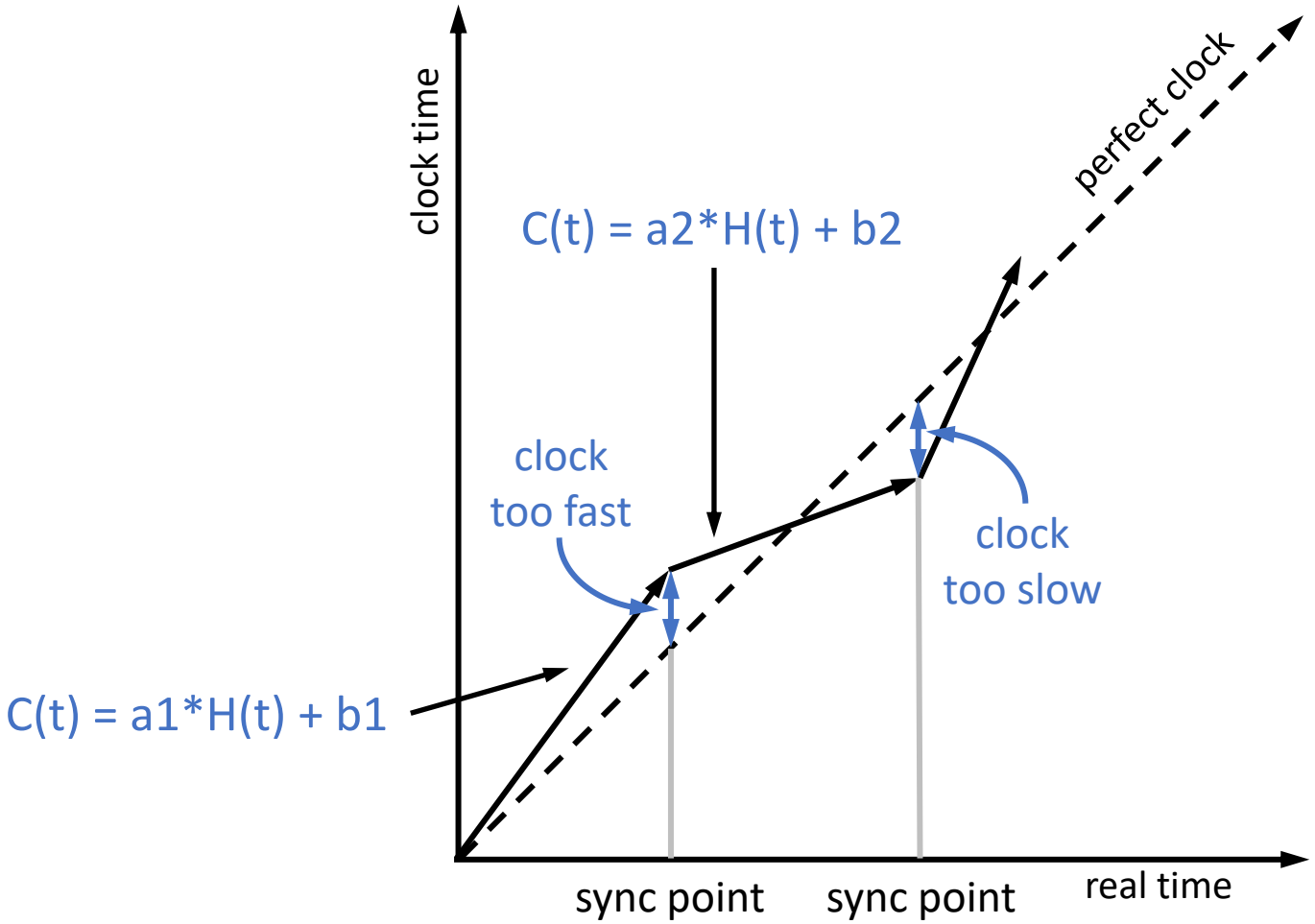
  $= (T_4-T_1)-(T_3-T_2)$
- Client samples round trip delay
- Client doesn't know D_{req}
- Approximates $D_{req} \approx D_{resp}$
  so $D_{resp} = D/2$
- Client sets local clock to $T_3+D/2$



Adjusting clock time

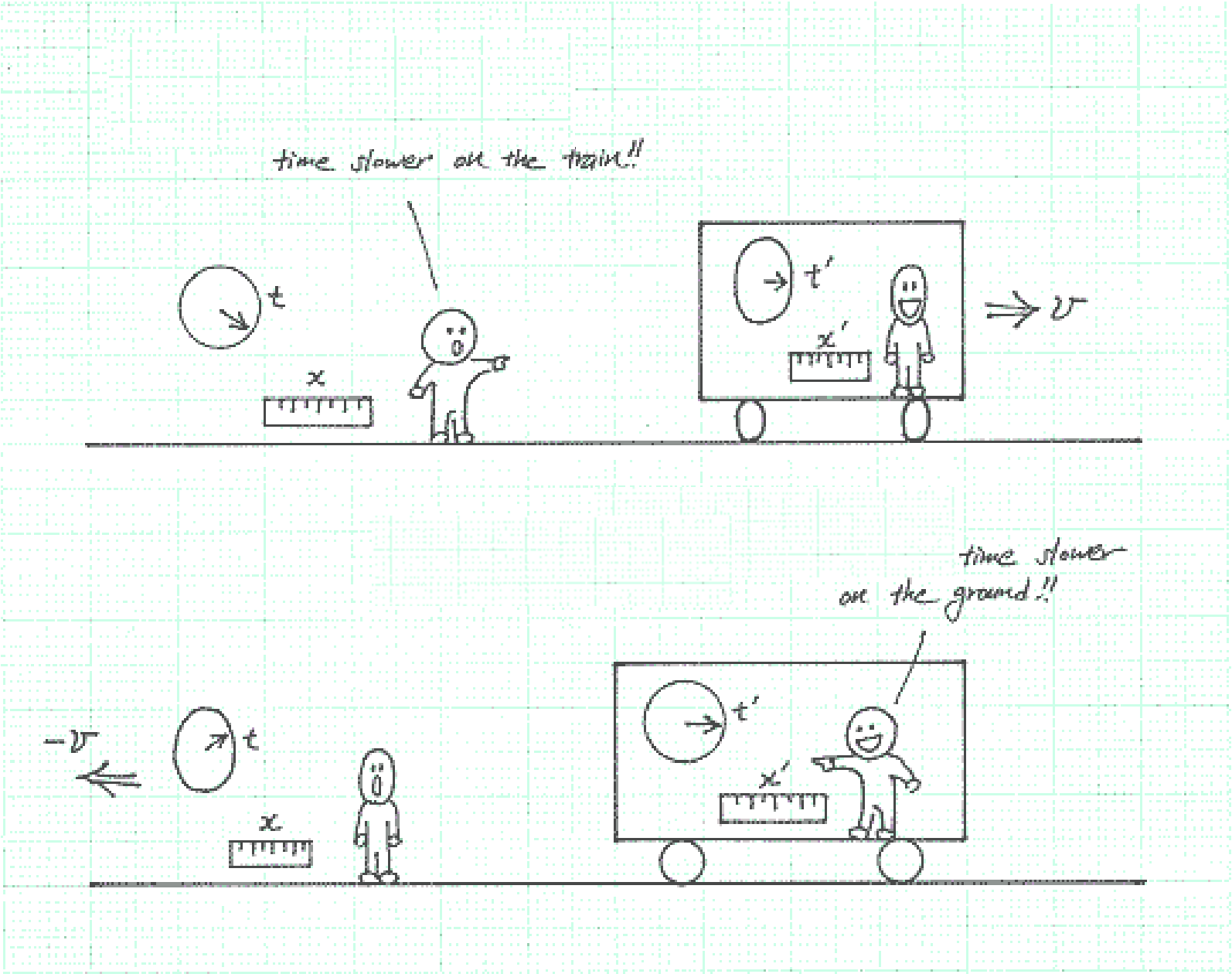
- Client needs to ensure that its clock time increases monotonically, i.e., it doesn't go backward when it synchronizes with a time server
  - Otherwise, programs may not work correctly, e.g., make
- OS periodically reads hardware clock time: H(t)
- Calculates clock time: C(t) = a*H(t) + b
  where a (slope) and b (offset) are constants
- Monotonicity requirement: $t' > t \Rightarrow C(t') > C(t)$
- Can achieve monotonicity by adjusting a and b

Clock adjustment in practice



Clock synchronization: takeaways

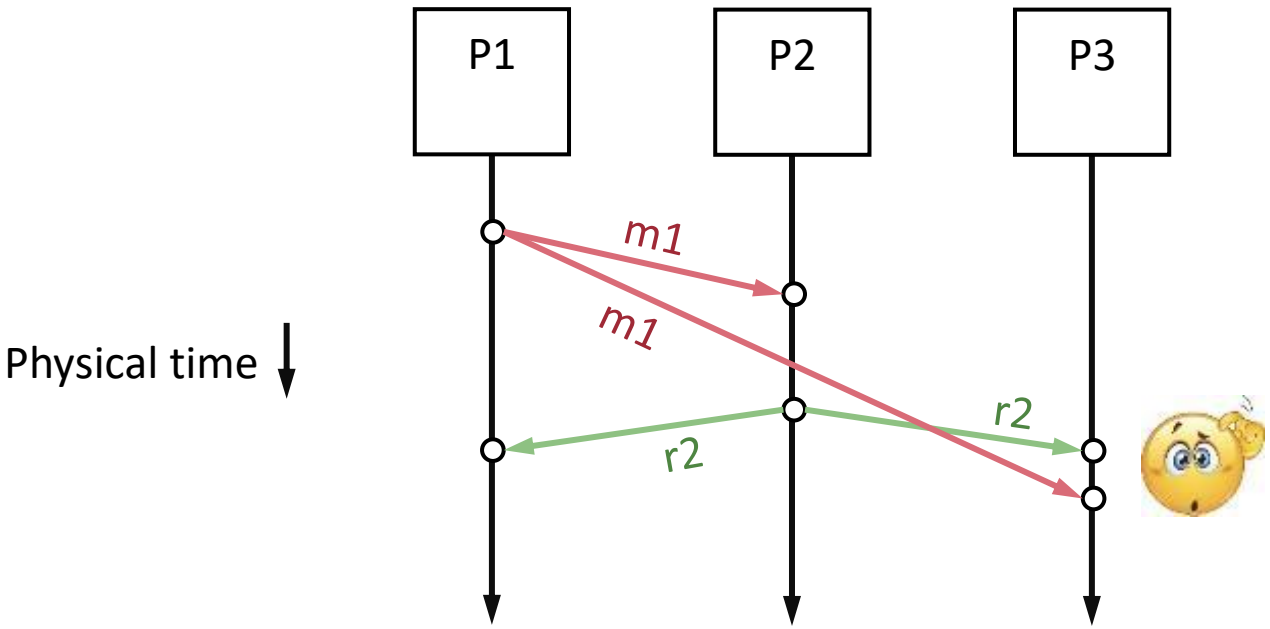
- Today, Network Time Protocol (NTP) is used for synchronizing clocks to UTC
  - Uses some of the methods we have discussed
  - Maintains time to within 10s of milliseconds on the Internet
  - Maintains time to within 1 millisecond in LAN
- However, clocks are never exactly synchronized
  - Thus, challenging to reason about precise ordering of events


time slower on the train!!
t
x
t'
x'
v
time slower on the ground!!
-v
t
x
t'
x'

Logical Clocks

Ordering events

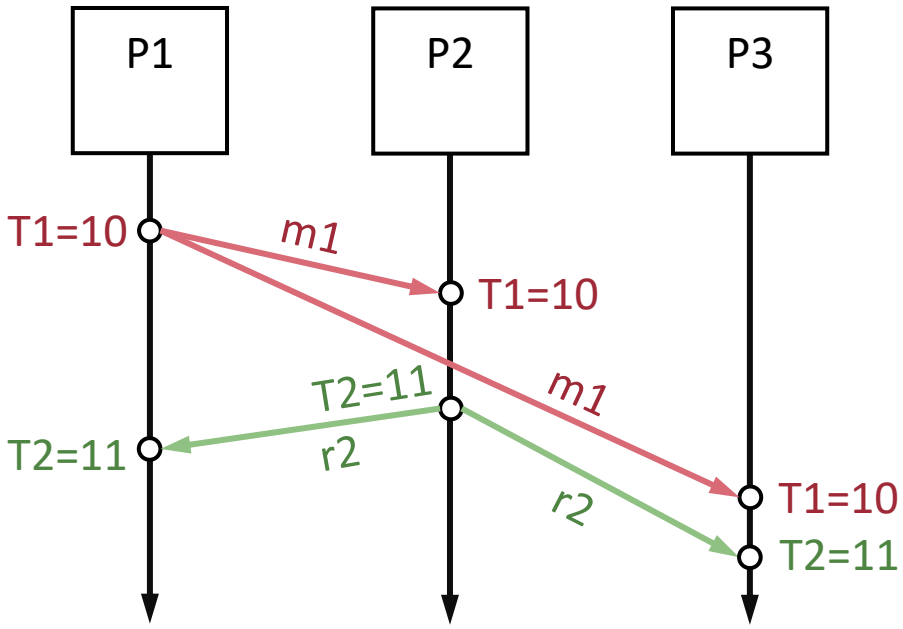
- There are 3 processes: P1, P2 and P3
  - P1 posts message m1 to P2 and P3
  - P2 sends reply r2 of message m1 to P1 and P3
  - P3 receives reply r2 before message m1



How can we fix the ordering problem?

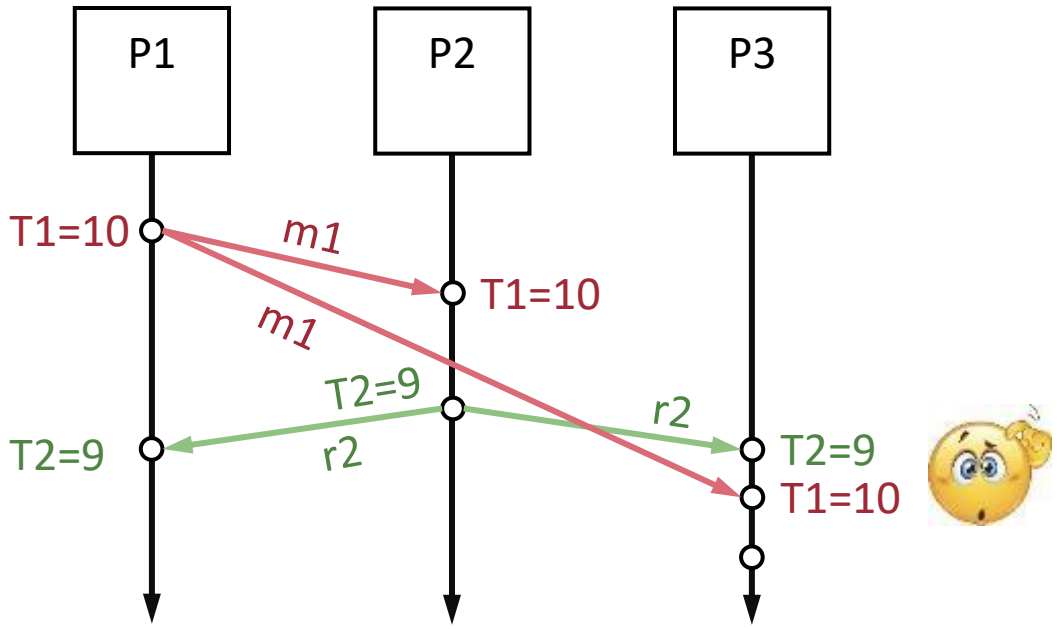
Ordering events

- Can we timestamp the message m1 and reply r2 and deliver the messages in timestamp order on P3?
  - P1 timestamps m1 with 10
  - P2 timestamps m2 with 11

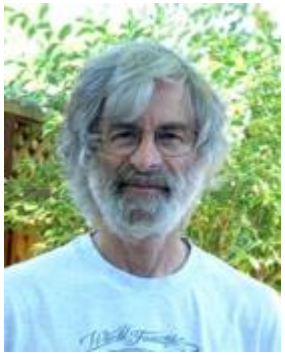


Ordering events

- But what if P2's clock is running slower than P1's clock?
- P3 may still deliver reply r2 before message m1



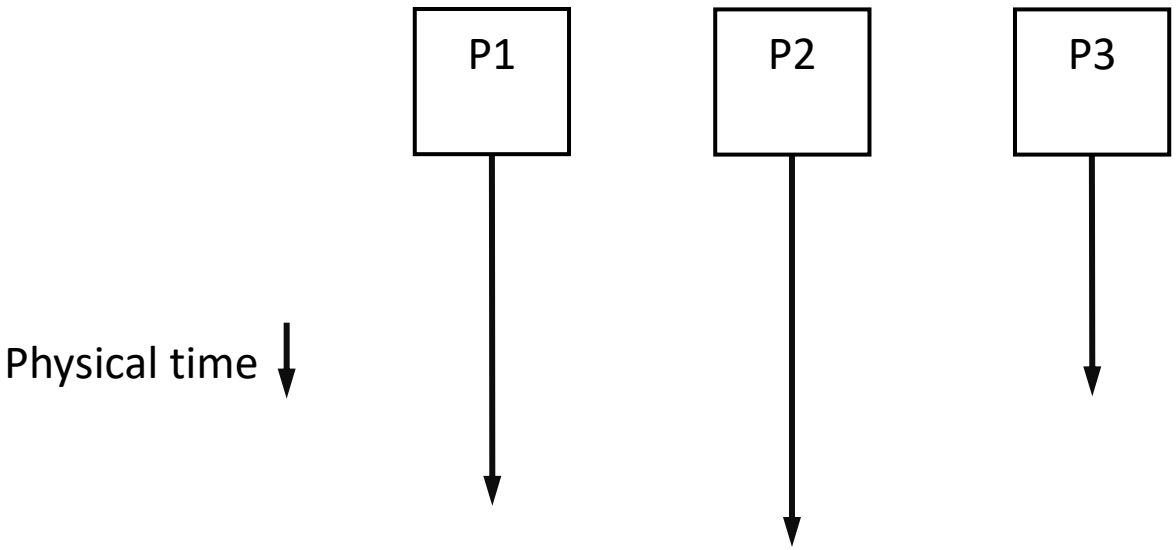
Logical clocks



- Key challenge: need to order events,
  but physical clocks are hard to synchronize
- Landmark 1978 paper by Leslie Lamport
  - Insight: processes do not need to agree on precise clock time,
    what matters is the order in which events occur
  - Idea: capture happens before relationship between every pair of events to define logical clock time
    - Physical clock counts number of elapsed seconds
    - Logical clock will count the number of events

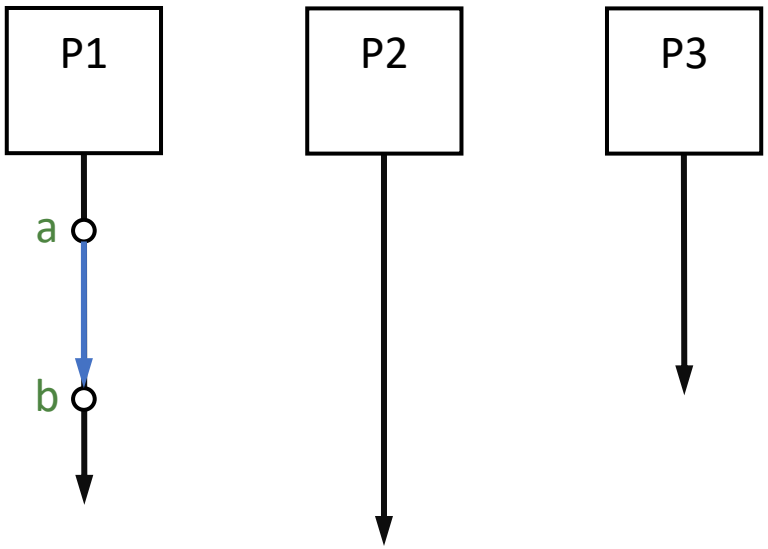
Defining happens-before

- Event a happens before event b is denoted as a → b
  - Captures notion of causality: a may have influenced b
- Consider three processes: P1, P2, and P3



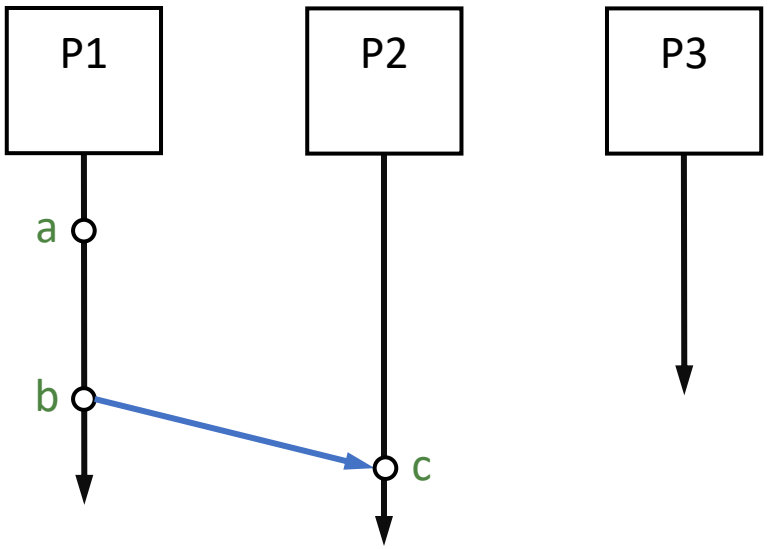
Defining happens-before

- We can observe event order within a process
- If a occurs before b within a process, then a → b



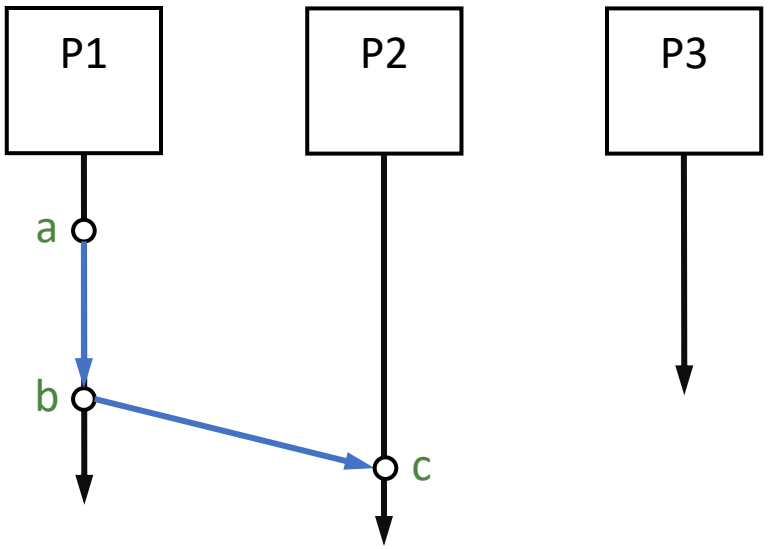
Defining happens-before

- We can observe event order when processes communicate
- If b is a send, c is a receive, then b → c



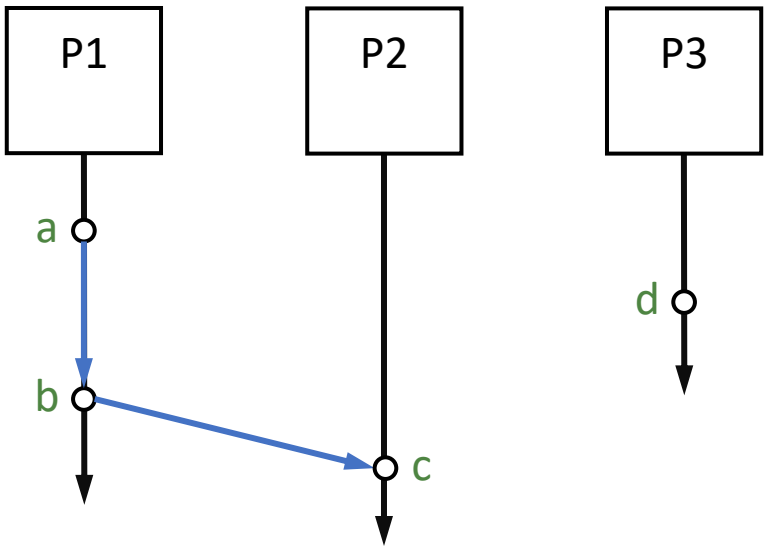
Defining happens-before

- We can observe event order transitively
- 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
- a, d could be ordered in either order, it wouldn't matter

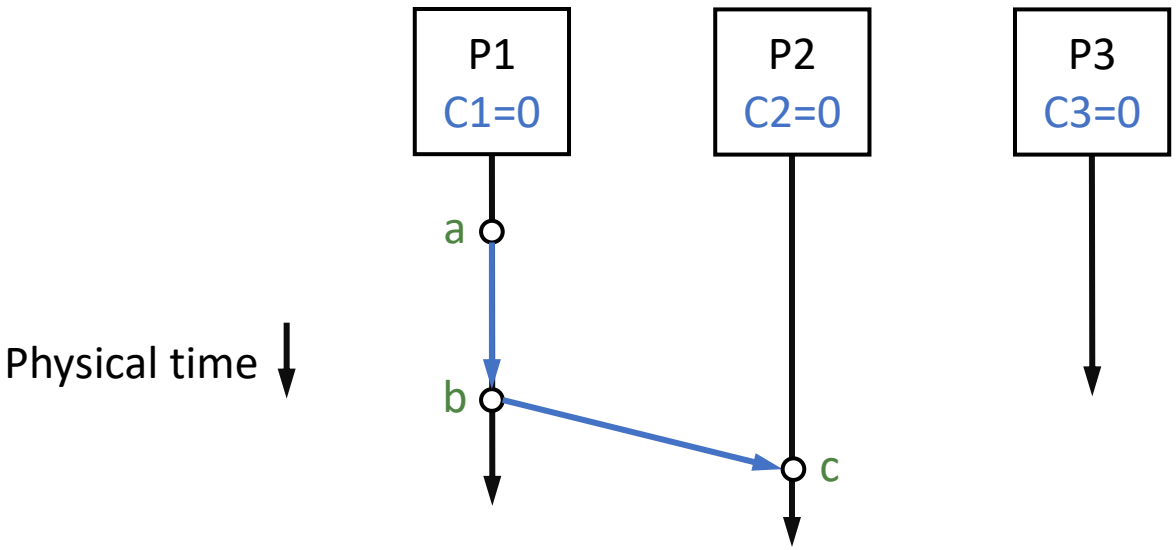


Logical clock time

- We will assign a logical clock C(a) for every event a, so that events can be ordered correctly
- Clock condition:
  If a → b, then we will ensure that C(a) < C(b)

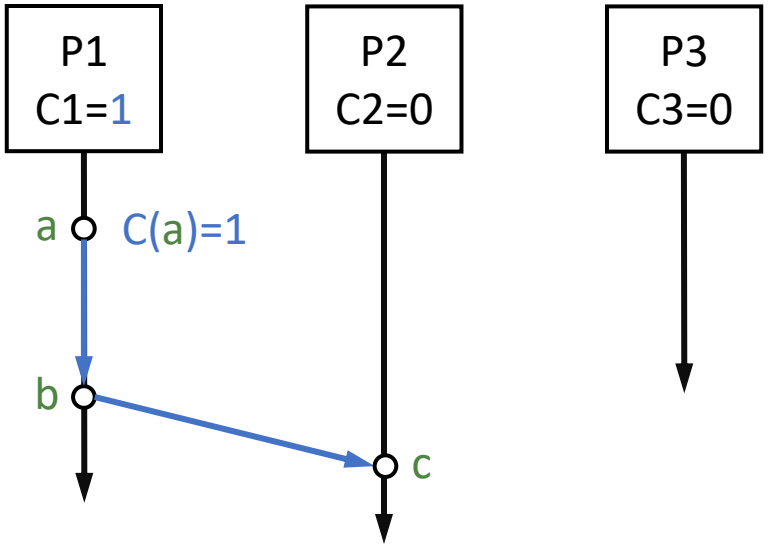
Logical clock algorithm

- Each process P maintains a logical clock C



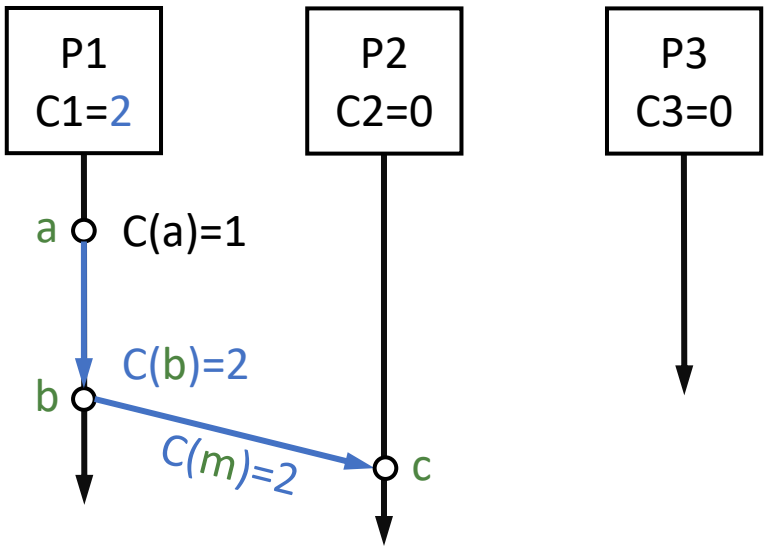
Logical clock algorithm

- Before P1 executes an event a:
  - C1 = C1 + 1 // increment C1 at P1
  - C(a) = C1 = 1 // set C(a) to current clock



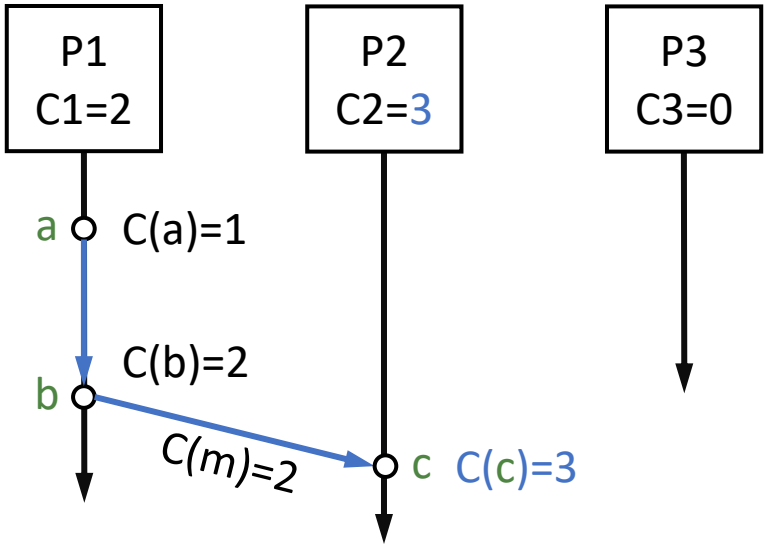
Logical clock algorithm

- When P1 sends message m to P2 at event b:
  - C1 = C1 + 1 // increment C1 at P1
  - C(b) = C1 = 2 // set C(b) to current clock
  - Send C1 in message m, so C(m) = 2



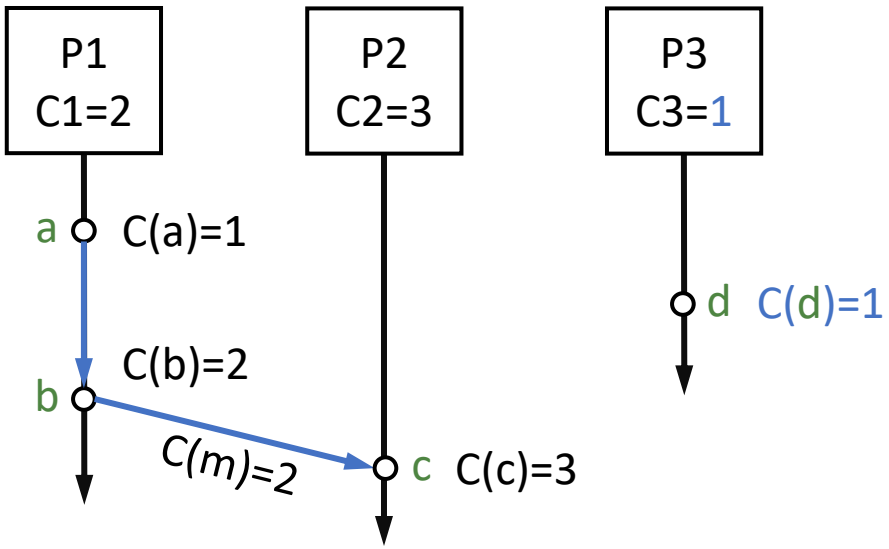
Logical clock algorithm

- When P2 receives message m at event c:
  - C2 = 1+max{C2, C(m)}
    = 1 + max{0, 2} = 3 // update C2 at P2
  - C(c) = C2 = 3 // set C(c) to current clock



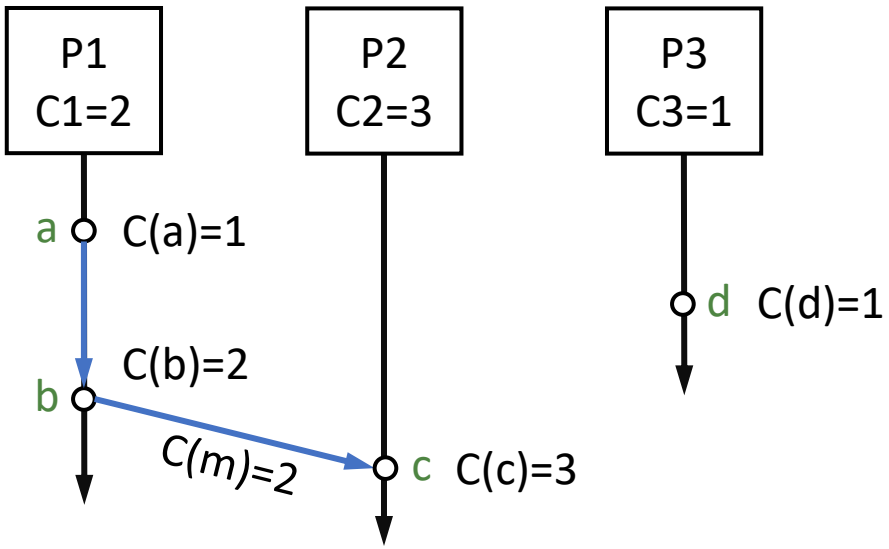
Logical clock algorithm

- Before P3 executes an event d:
  - C3 = C3 + 1 = 1
  - C(d) = C3 = 1



Ordering of events

- The logical clock algorithm, until now, can have ties
  - e.g., C(a) in P1 is the same as C(d) in P3

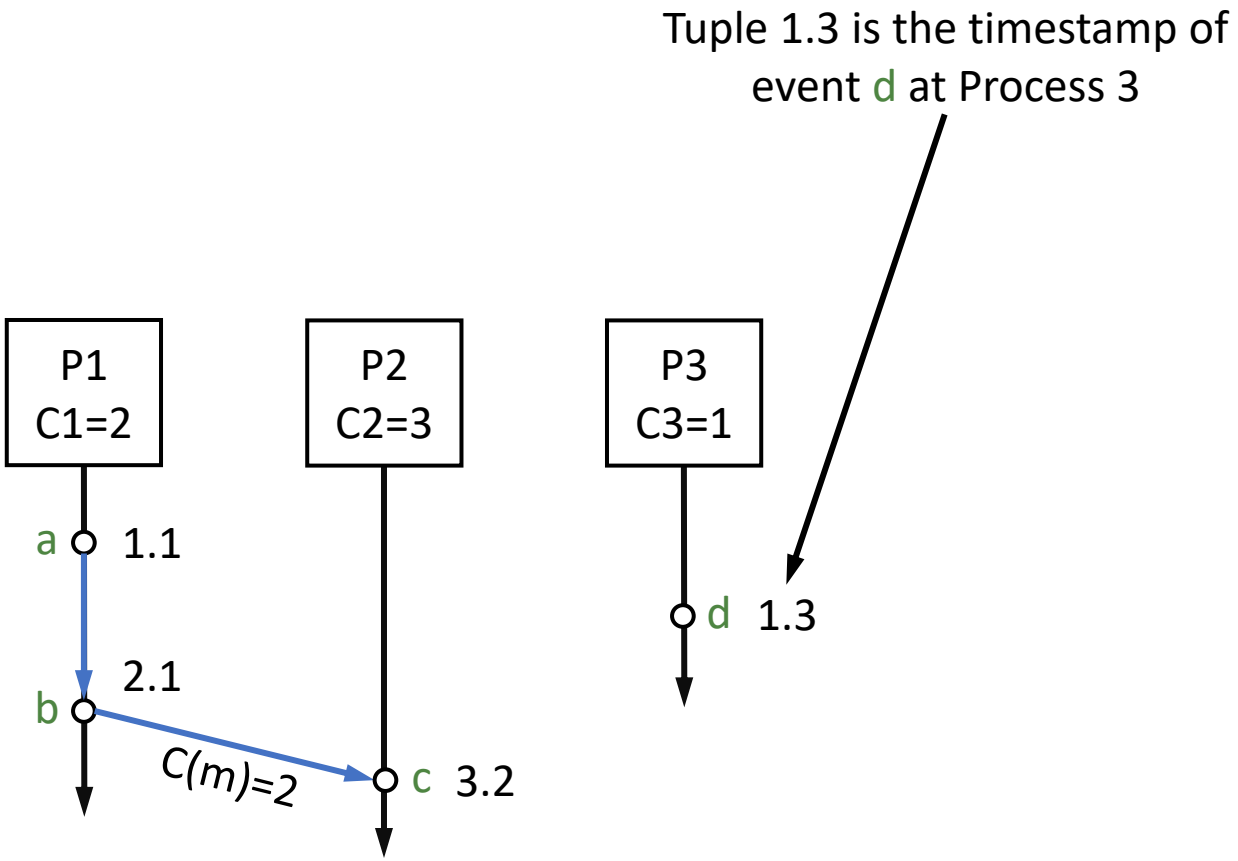


Total order of events

- The logical clock algorithm, until now, can have ties
  - e.g., C(a) in P1 is the same as C(d) in P3
- Use unique process number to break ties
  - Let Process Pi timestamp event a as tuple C(a).i
  - Let Process Pj timestamp event b as tuple C(b).j
- Define order relation ⇒ between a and b events:
  - a ⇒ b iff (C(a) < C(b)) or [(C(a) = C(b)) and (i < j)]
- The ⇒ relation is a total order,
  since for any two events a and b, either a ⇒ b or b ⇒ a

Total ordering of events

- a ⇒ d ⇒ b ⇒ c



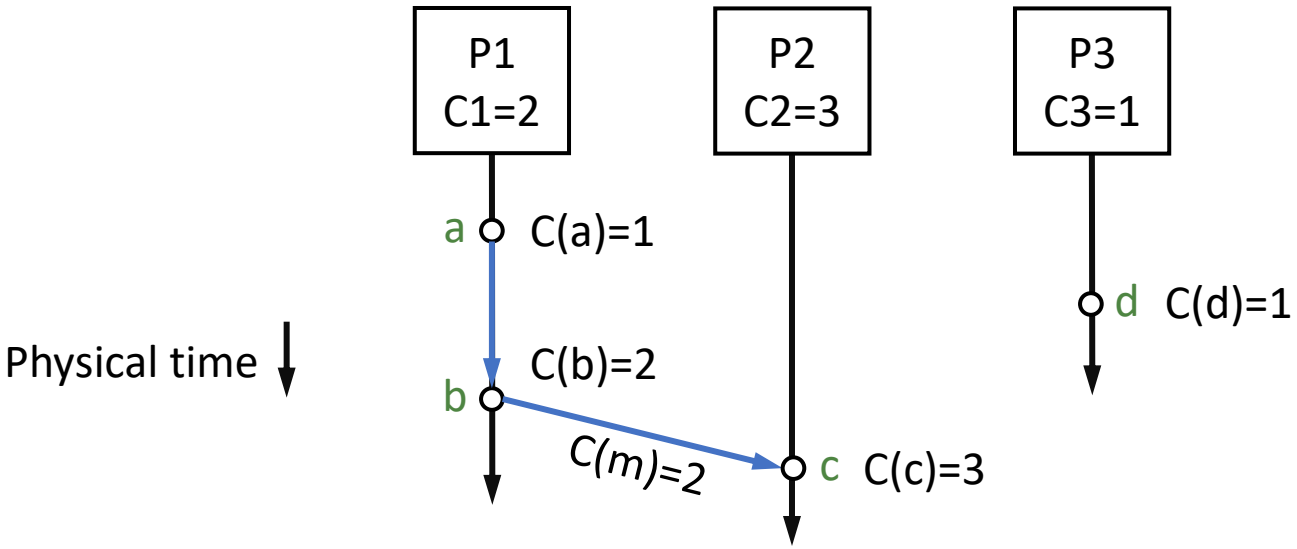
Implications of logical clocks

- Logical clocks allow assigning a total order to events
  - Total order respects causality
  - Ordering does not require physically synchronized clocks!
- Causality assumes all messages are visible to the system
  - P1 issues a job opening request on a job site
  - P1 calls P2 by phone to say that this job is available
  - When P2 looks for the job opening, it is not there! Why?
- We will look at applications of logical clocks later

Vector Clocks

Motivation for vector clocks

- Let's look at logical clocks again
  - The clock condition says: a → b implies C(a) < C(b)
  - However, C(a) < C(b) does not imply a → b
  - E.g., C(d) < C(c) below, but d ‖ c



We cannot determine whether two events a and b with logical times C(a), C(b) are causally related

Vector clocks: idea

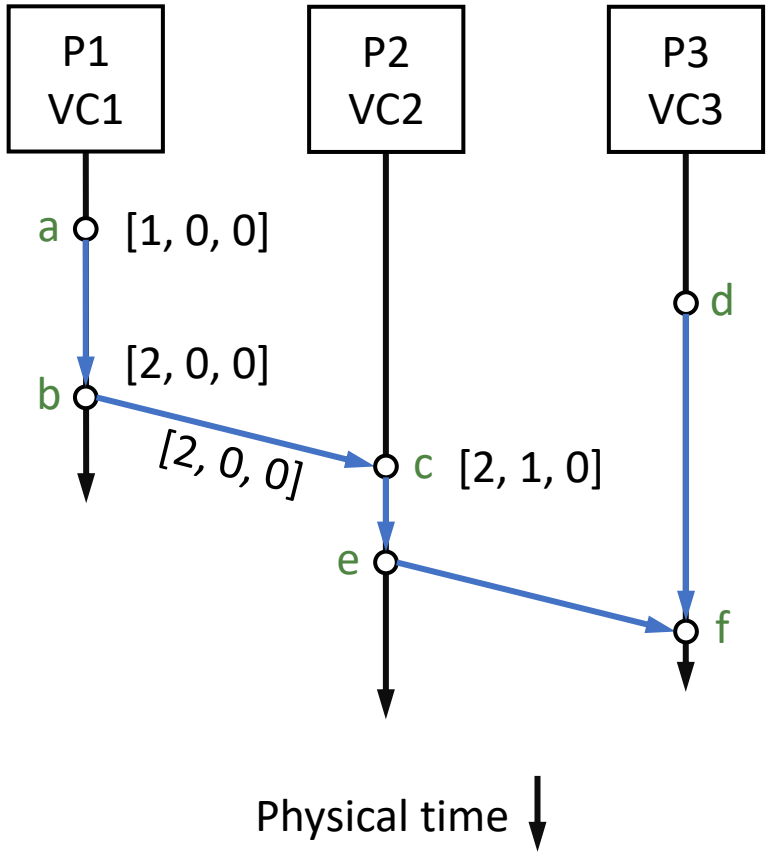
- One integer can't order events in multiple processes
- Idea: label each event with a vector clock (VC)
- A vector clock is a vector of integers, with one entry for each process
- For event e, VC(e) = $[c_1, c_2, \ldots, c_n]$
  - n is number of processes
  - Each entry c_i is a count of events in Process i that happen before event e

Vector clocks: algorithm

- Each process maintains a vector clock
  - Initially, all vector clocks are VC = [0, 0, …, 0]
- Three rules:
  1. For each local event in Process i:
     - VC[i] = VC[i] + 1 // increment count for self
  2. When Process i sends a message m to any process:
     - VC[i] = VC[i] + 1
     - send(VC, m) // send local vector with message m
  3. When Process i receives message with vector clock VC_m:
     - VC[j] = max{VC[j], VC_m[j]}, for each j in {1, …, n}
     - VC[i] = VC[i] + 1
- Rules like logical clock rules, but use vector clocks

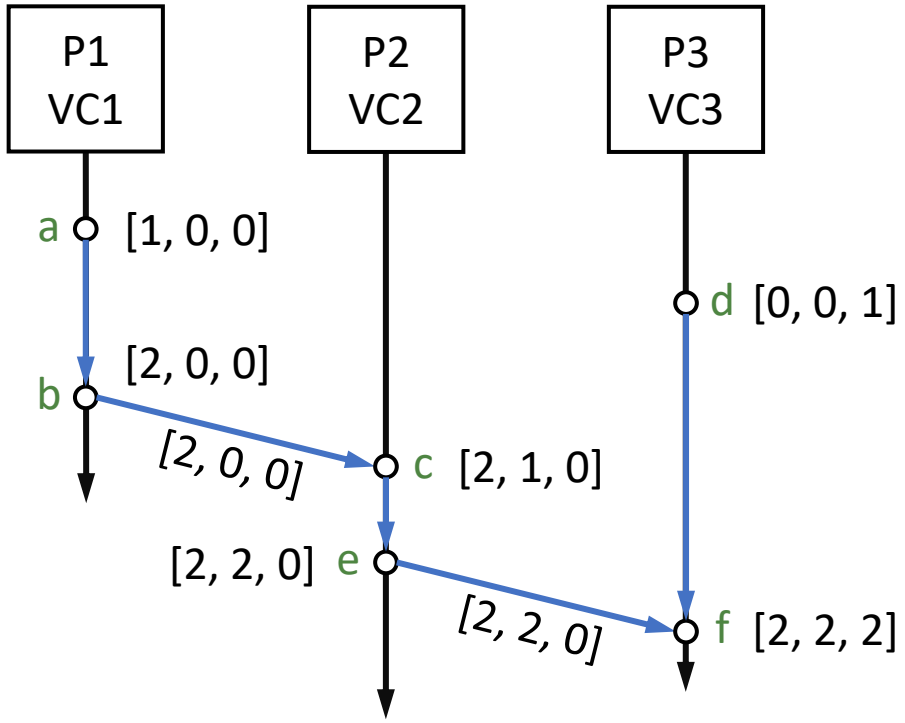
Vector clocks: example

- All processes' VCs start at [0, 0, 0]
- Applying local update rule
- Applying send rule
  - Piggyback local vector clock on inter-process message
- Applying receive rule



Vector clocks: example

- All processes' VCs start at [0, 0, 0]
- Applying local update rule
- Applying send rule
  - Piggyback local vector clock on inter-process message
- Applying receive rule

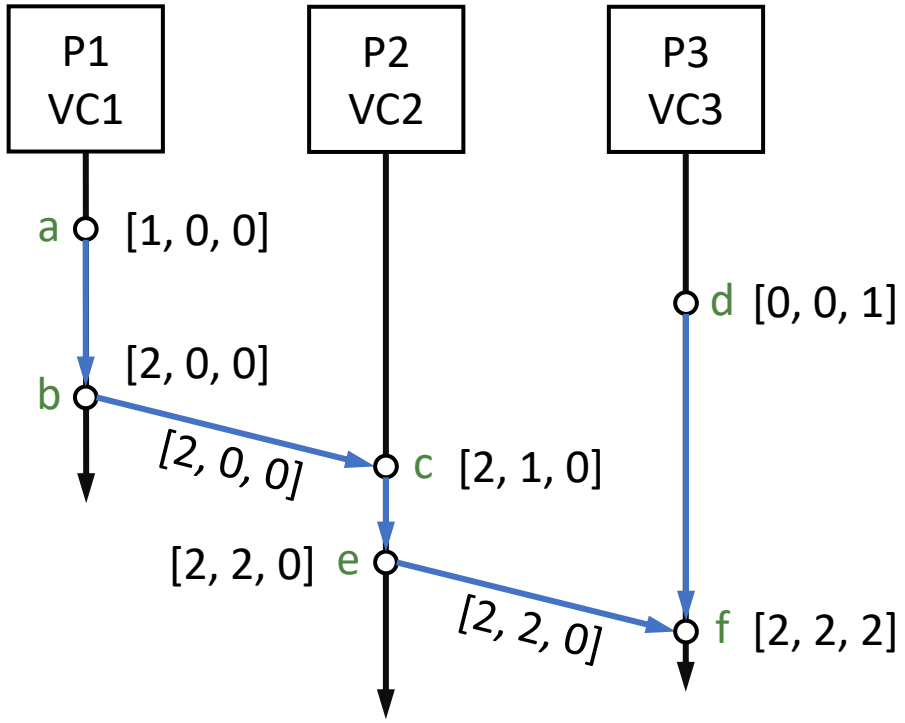


Comparing vector clocks

- For events a and b,
  the rules for comparing vector clocks are:
  - $V(a) = V(b)$ when $a_k = b_k$ for all k
  - $V(a) < V(b)$ when $a_k \leq b_k$ for all k and $V(a) \neq V(b)$
  - $V(a) \parallel V(b)$ when $a_i < b_i$ and $a_j > b_j$, for some i and j

Vector clocks and causality

- Vector clocks capture causality:
  - V(a) < V(b) ⇔ a → b
  - V(a) = V(b):
    a and b are the same event
  - V(a) ‖ V(b):
    a and b are concurrent
- We will look at applications of vector clocks later



Conclusions: physical clocks

- Each node runs its own physical clock
- Distributed applications running on different nodes need to order events, expect clocks to be synchronized
- Clock synchronization protocols allow synchronizing a physical clock to a reference clock
- But exact synchronization is not possible

Conclusions: logical, vector clocks

- Happens-before → relationship allows tracking potential causality between events
- Logical clocks use → to provide a total order of events:
  - a → b ⇒ C(a) < C(b), without requiring physical clocks
  - Use a single counter per event, but cannot track causality
- Vector clocks capture causal order between events:
  - a → b ⇔ V(a) < V(b)
  - Require a vector counter per event