CIS 505
Software Systems

Matt Blaze

Spring 2012

2012-02-14

Some slides are courtesy of Boon Loo, Insup Lee., Silberschatz et.al., and Colouris et. al.

Next few lectures

- Naming in distributed systems (Chapter 5)
- Synchronization
  - Physical and Logical clocks (Chapter 6.1-6.2)
  - Mutual exclusion and elections (Chapter 6.3, 6.5)

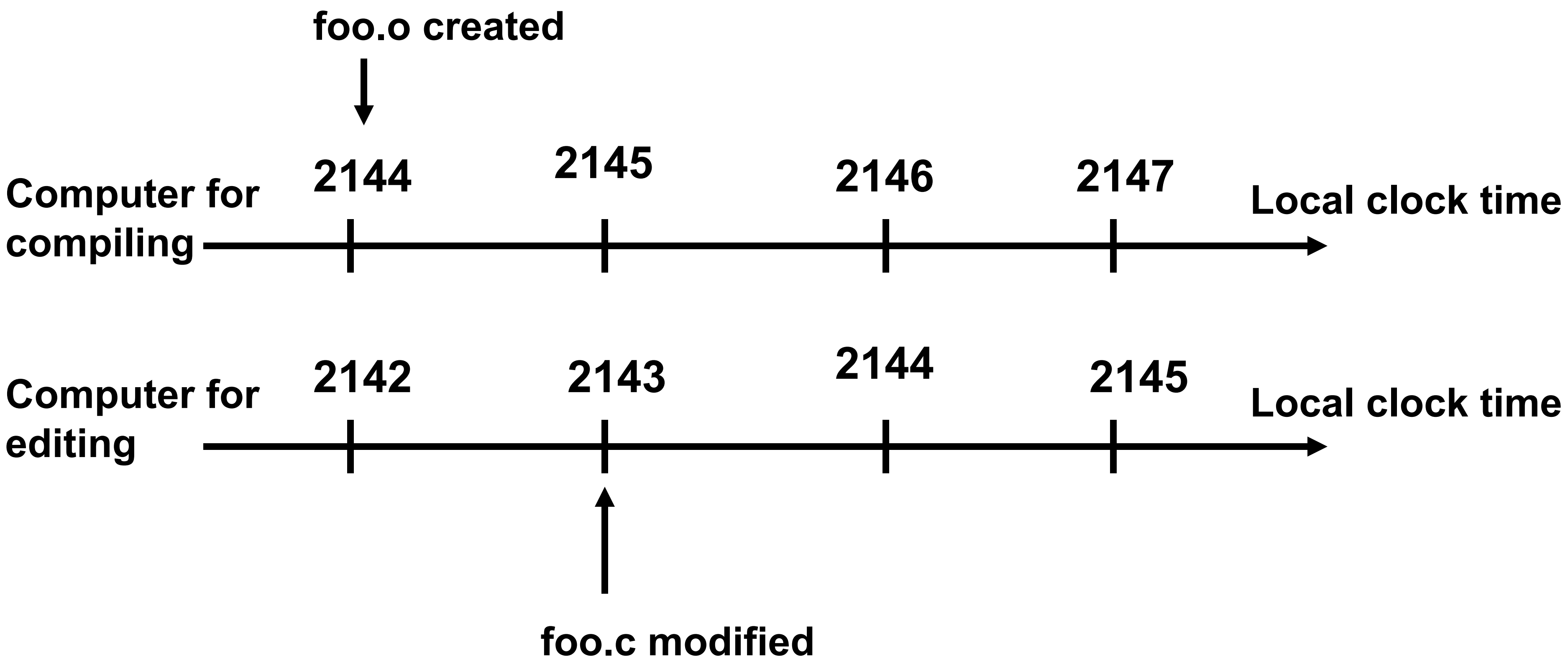
Distributed Synchronization

- So far, we have studied at synchronization on a single machine among different processes/threads
- Synchronization in distributed systems is harder than in centralized systems because the need for distributed algorithms.
- Properties of distributed algorithms:
  1. The relevant information is scattered among multiple machines.
  2. Processes make decisions based only on locally available information.
  3. A single point of failure in the system should be avoided.
  3. Handle reorder and loss of messages
  4. No common clock or other precise global time source exists.
- Why is time synchronization important?
  - Measure delays between distributed components, synchronize streams (audio and video), detect event ordering for causal analysis, utilities use modification timestamps, e.g. Make, archive

Two Problems….

- *Event ordering*
  - what’s the sequence of events across machines in a distributed environment?
- *Absolute time*
  - what time (on the clock) did an event occur?
  - generally, this also provides event ordering
    - but only within the limits of clock error
    - can create ambiguity / “simultaneous” events

Why synchronize clocks?



- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Clock Skew Problem

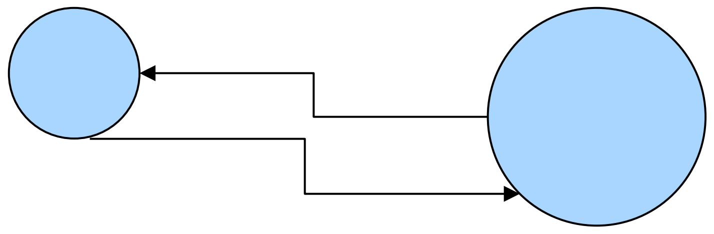
- *Clock skew(offset)*: the difference between the times on two clocks
- *Clock drift* : count time at different rates
  - A clock's frequency varies with temperature. Clocks on different computers drift due to differing oscillation period
  - Ordinary quartz clocks drift by ~ 1sec in 11-12 days. (10^{-6} secs/sec).
  - High precision quartz clocks drift rate is ~ 10^{-7} or 10^{-8} secs/sec

Time Sources

- GMT / UTC (Universal Time Coordinated)
  - Base of any international time measure.
  - Based on standard atomic clocks with occasional leap seconds inserted to keep in phase with astronomical time - earth’s orbit round sun.
- Require (GPS or UTC) receivers on servers to support a clock synchronization service
- Broadcast via shortwave radio (WWV) by the National Institute of Standard Time (NIST)
  - Radio stations broadcast UTC & provide a short pulse every second.
  - Random atmospheric delays can reduce accuracy
- Geostationary Environment Operation Satellite (GEOS) or Global Positioning Systems (GPS) provide UTC to better than ±0.5msec

Network Time Protocol (NTP)

- Request time from server
  - Note round trip time
    - Estimate lopsidedness of the latency
  - Add ½ RTT to the server's response
  - Adjust clock



- Average multiple requests
- Tiered system
  - Accuracy vs. load

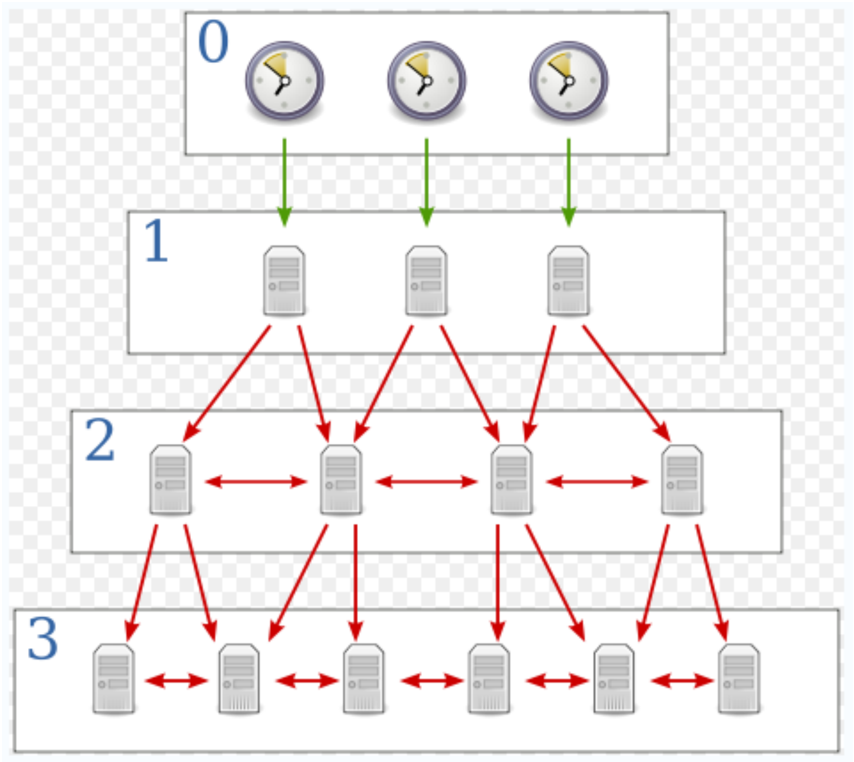
NTP Clock Strata

NTP divides servers into strata

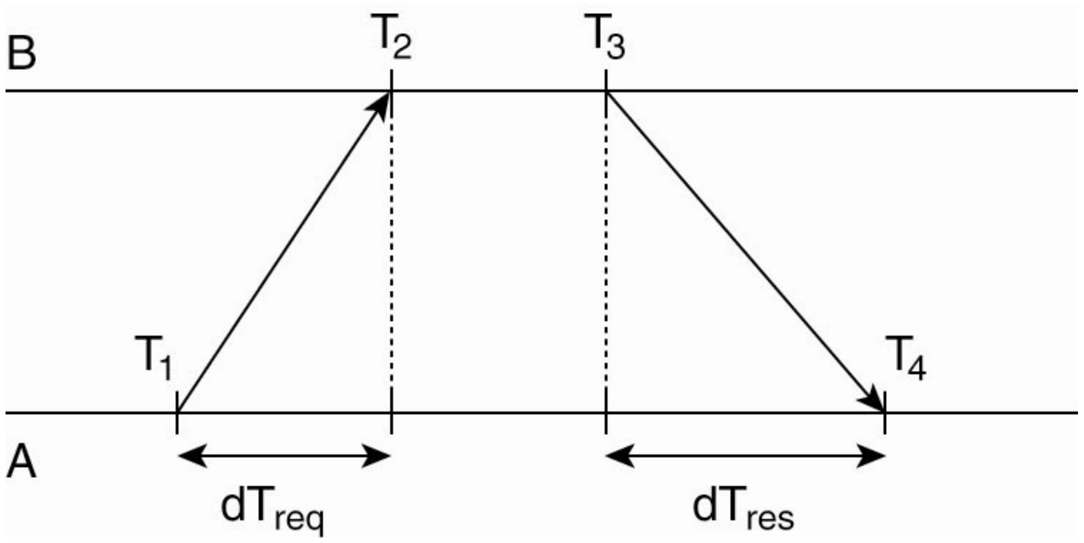
Strata 0: server with reference clocks that receive UTC (WWV receiver or GPS or atomic clock)

Clocks belonging to servers with high stratum numbers are liable to be less accurate than those with low stratum numbers

Three mechanisms:
Multicast: 1 or more server periodically multicast to other servers on a high speed LAN. Set clocks assuming small delays.
Procedure call mode: Client request time from 1 or more servers. Used when there is no multicast.
Symmetric protocol: pair-wise synchronization. Used by layers closer to primary.



Network Time Protocol

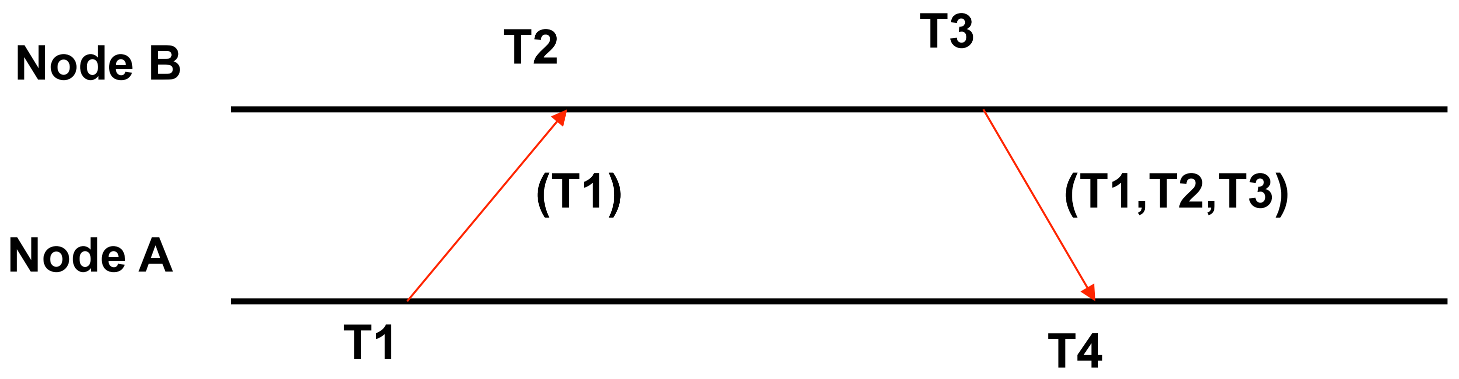


- A sends request to time server B, timestamped with value T1.
- B records the time of receipt T2, and returns a response timestamped T3, piggyback with previously record value T2.
- A records time of response's arrival, T4.
- Assuming symmetrical delays: T2-T1 = T4-T3
- A's delta relative to B = T3 + [(T2-T1) + (T4-T3)] / 2 – T4

  = [(T2-T1) + (T3-T4)] / 2
- If delta < 0, time never runs backwards! Slow clock down (e.g. adding 10 msec per clock tick) so it is corrected over a time period

Synchronization Phase

- Delta = clock drift
- P = propagation delay



T2=T1+P+Delta

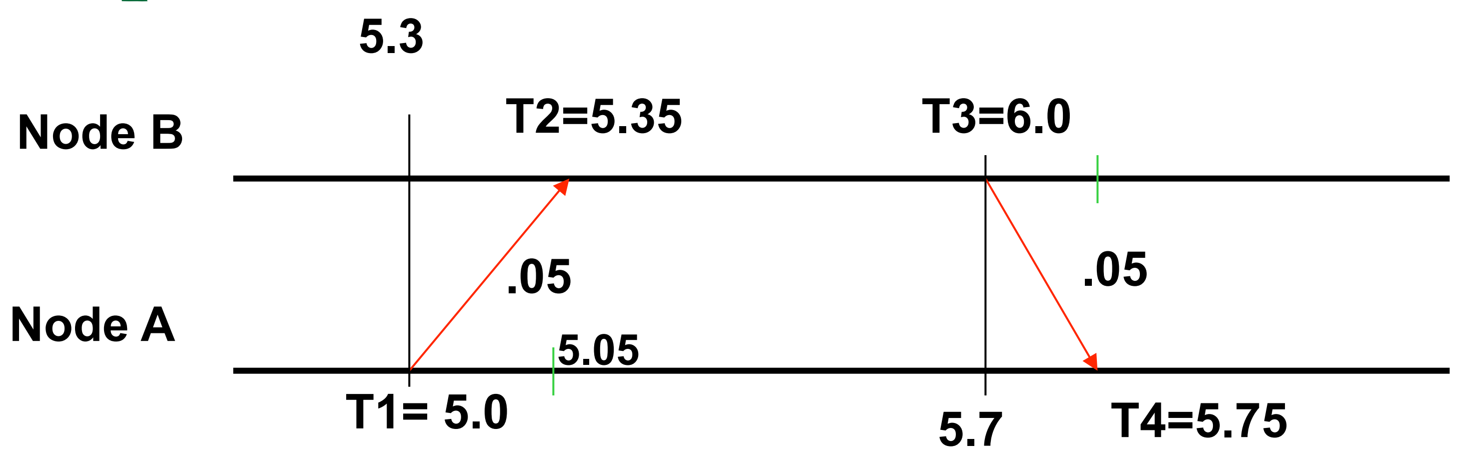
Delta=T2-T1-P

P = ((T2-T1)+(T4-T3))/2

Delta = ((T2-T1)-(T4-T3))/2

Node A corrects its clock by Delta
Note: Sender A corrects to clock of receiver B

Example



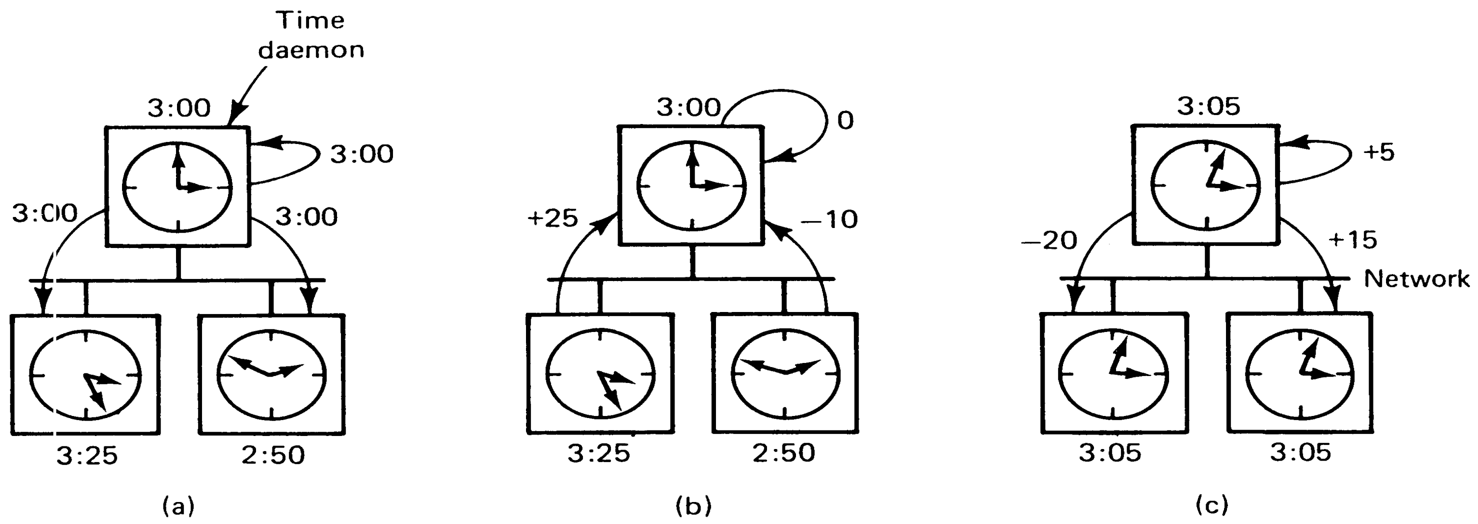
P = ((T2-T1)+(T4-T3))/2
P= ((5.35-5.0)+(5.75-6))/2
P=((.35)+(-.25))/2
P= .1/2=.05

Delta = ((T2-T1)-(T4-T3))/2
Delta= ((.35)-(-.25))/2
Delta= 0.6/2 =.3

So A adds .3 to 5.75 to get 6.05
Only need Delta to adjust clocks

Berkeley Algorithm

- Nominally isolated group of networked clocks
- One local master becomes the reference server. Chosen via election.
  - Master estimates local times with compensation for propagation delay
  - Calculate average time, but ignore occasional readings with propagation delay greater than a cut-off value or whose current clock is badly out of synch.
  - Sends message (delta, why?) to each slave indicating clock adjustment
- Eliminates readings from faulty clocks (takes *a fault tolerant average*).
  - Subset of clocks is chosen that differ from each other by no more than a specific amount, and average taken of readings from these clocks



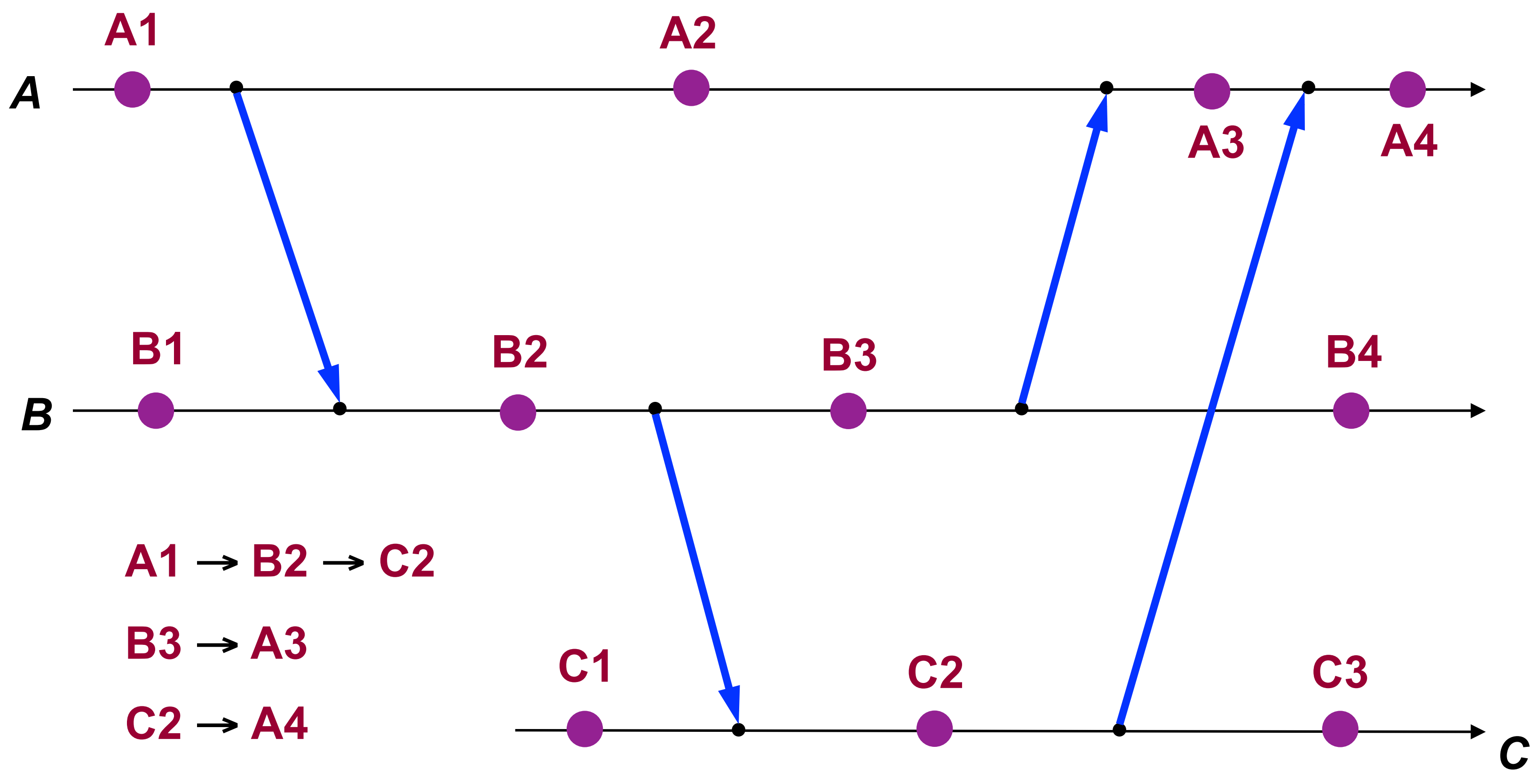
Types of Clocks

- Physical Clocks
  - NTP and Berkeley algorithm to correct drift of computer clocks
- Logical clocks:
  - Who cares about time anyway? Ordering of events is all that matters.
- Vector clocks:
  - Captures causality: When A receives a message from B, it needs to know B’s state that cause the message to be generated.

Lamport “Happens Before”

- A → B means A “happens before”
  - A and B are in same process, and B has a later timestamp
  - A is the sending of a message and B is the receipt
- Transitive relationship
  - A → B and B → C implies A → C
- If neither A → B nor B → A are true, then A and B are “concurrent” (not simultaneous)

Causality Example: Event Ordering



A1 → B2 → C2

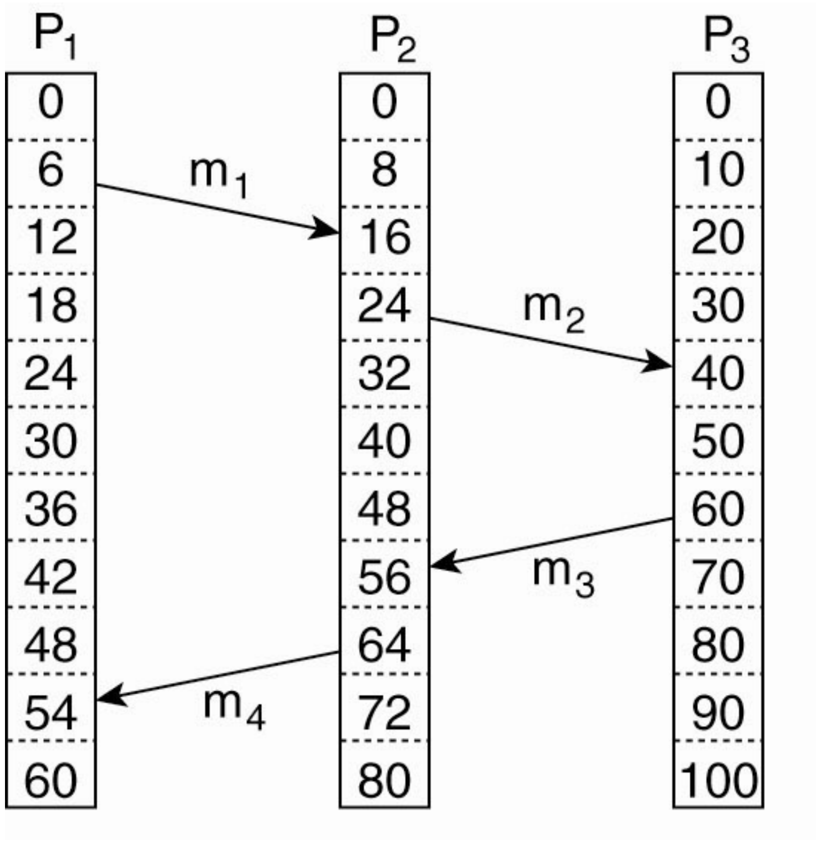
B3 → A3

C2 → A4

Lamport's Logical Clocks

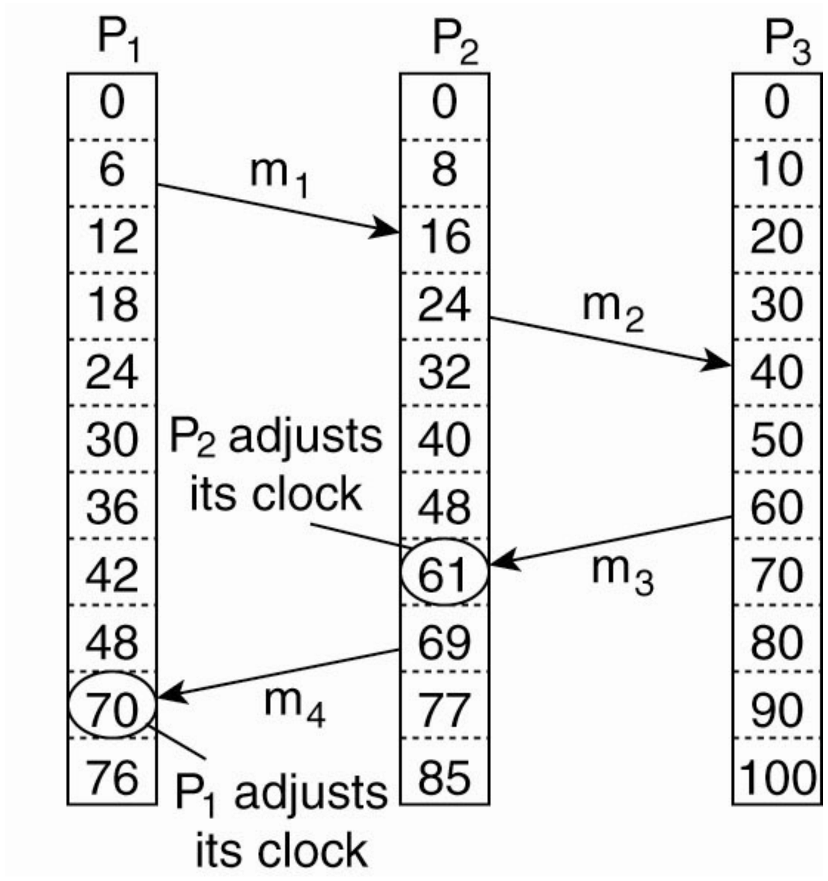
- Basic approach:
  - When message arrives, if process time is less than timestamp s, then jump process time to s+1
  - Clock must tick once between every two events
- Outcome:
  - If $A \rightarrow B$ then must have $LC(A) < LC(B)$
  - If $LC(A) < LC(B)$, it does NOT follow that $A \rightarrow B$
    - Revisit this later for vector clocks

Example: Lamport's Logical Clocks



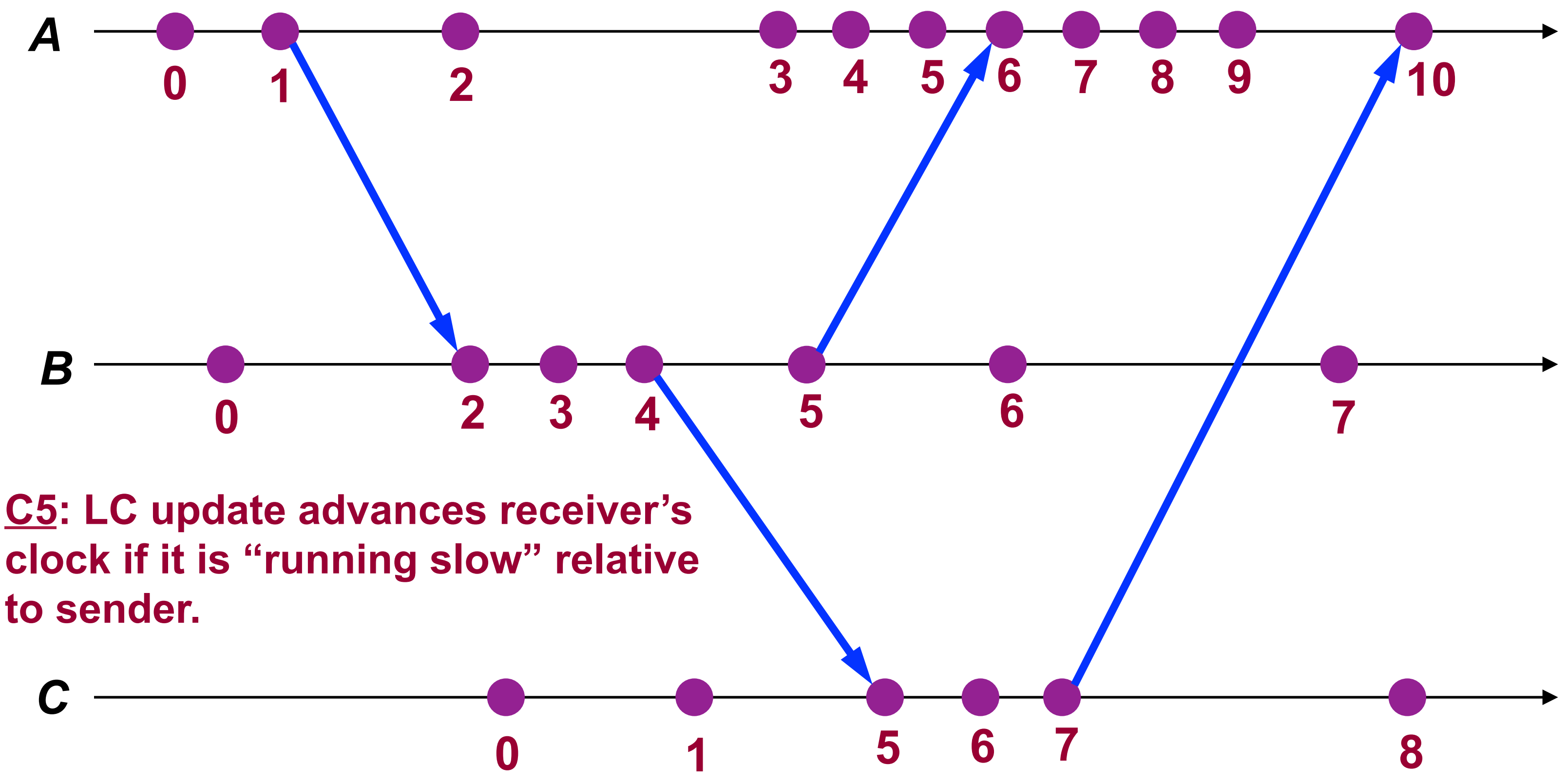
- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's algorithm corrects the clock.

Example: Lamport's Logical Clocks



- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's algorithm corrects the clock.
- **Invariant: If A → B then must have LC(A) < LC(B)**

Logical Clocks: Example



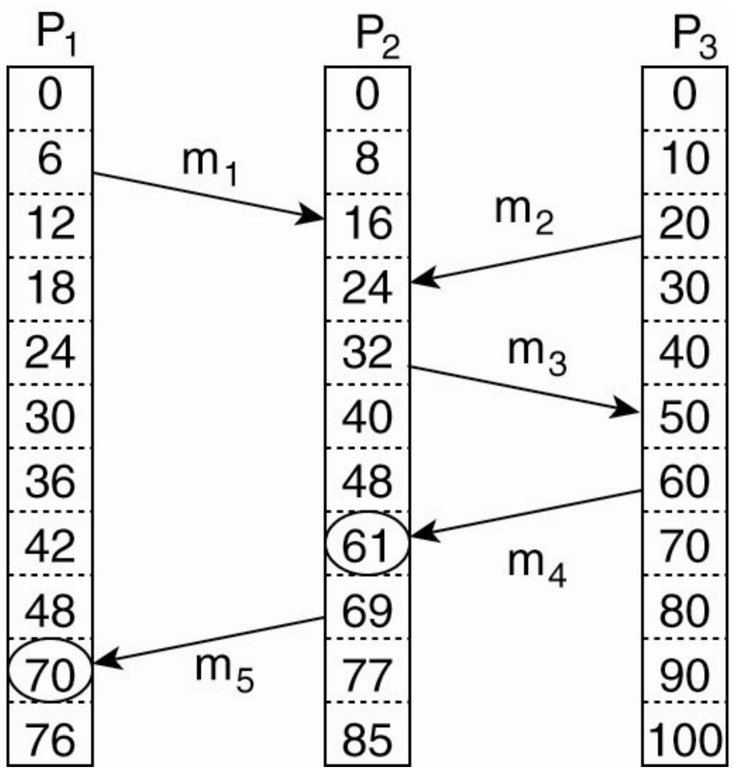
A6-A10: receiver's clock is unaffected because it is "running fast" relative to sender.

C5: LC update advances receiver's clock if it is "running slow" relative to sender.

Motivation for Vector Clocks

- *Logical* clocks induce an order consistent with causality:
  - If A → B then must have LC(A) < LC(B)
- However, the converse of the *clock condition* does not hold:
  - If LC(A) < LC(B), it does NOT follow that A → B
  - *LC(*e_1*) < LC(*e_2*)* even if e_1 and e_2 are concurrent.
  - Concurrent updates may be ordered unnecessarily.
- We need a clock mechanism that is necessary and sufficient in capturing causality

Vector Clocks

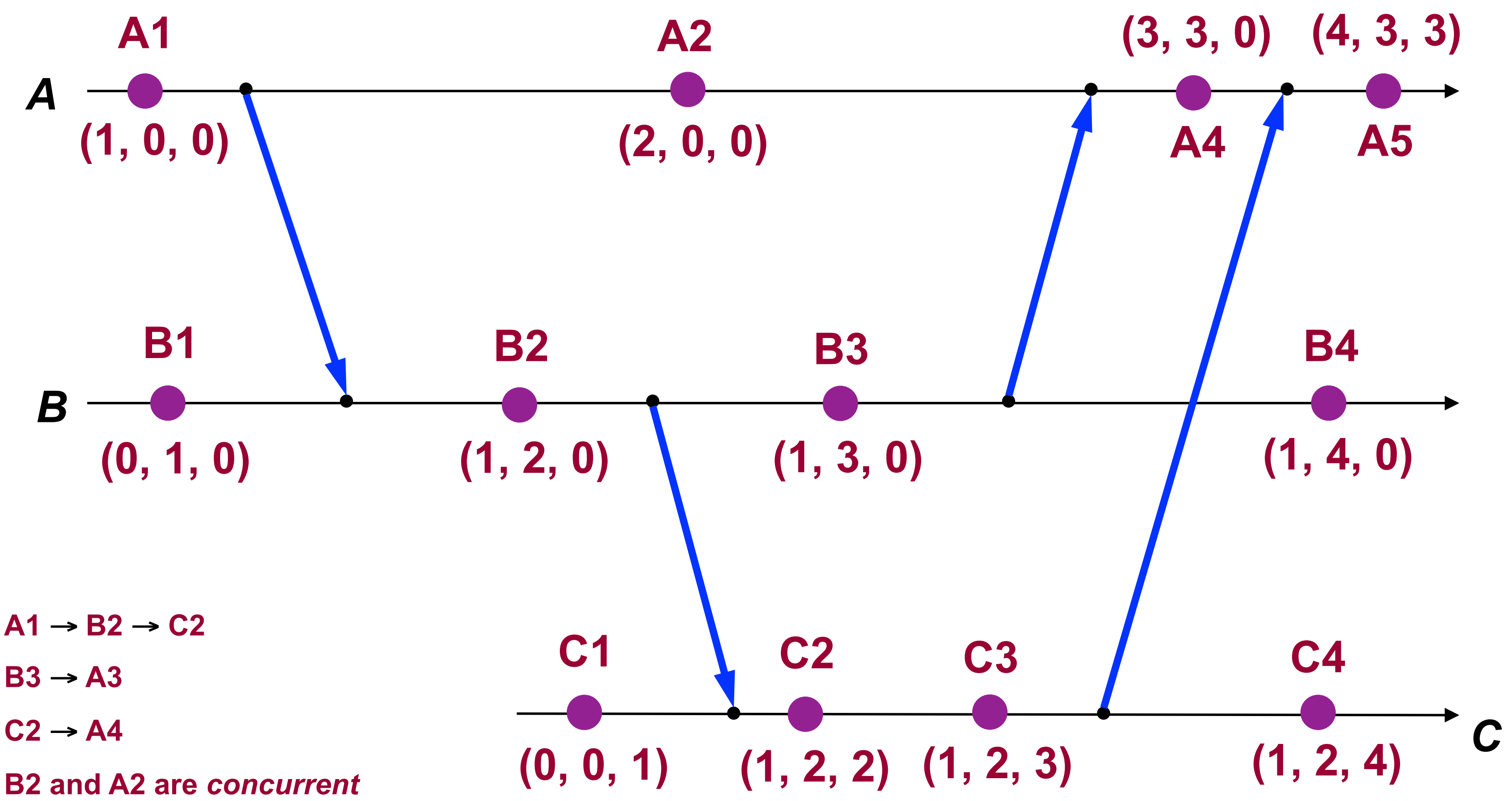


- Event a: P2 receive(m1). LC(a)=16
- Event b: P3 send(m2). LC(b) =20
- Events a and b are concurrent even though 16<20. I.e. we cannot conclude that a causally precedes b,

Vector Timestamps

- When process *I* generates a new event, it increments its logical clock.
- At each process I, a vector V_I is maintained:
  - $V_I[I]$: number of events occurred in process I
  - $V_I[J] = K$: process I knows that K events have occurred at process J
- All messages carry vectors
- When J receives vector v, for each K it sets $V_J[K] = v[K]$ if it is larger than its current value $V_J[K]$

Vector Clocks: Example



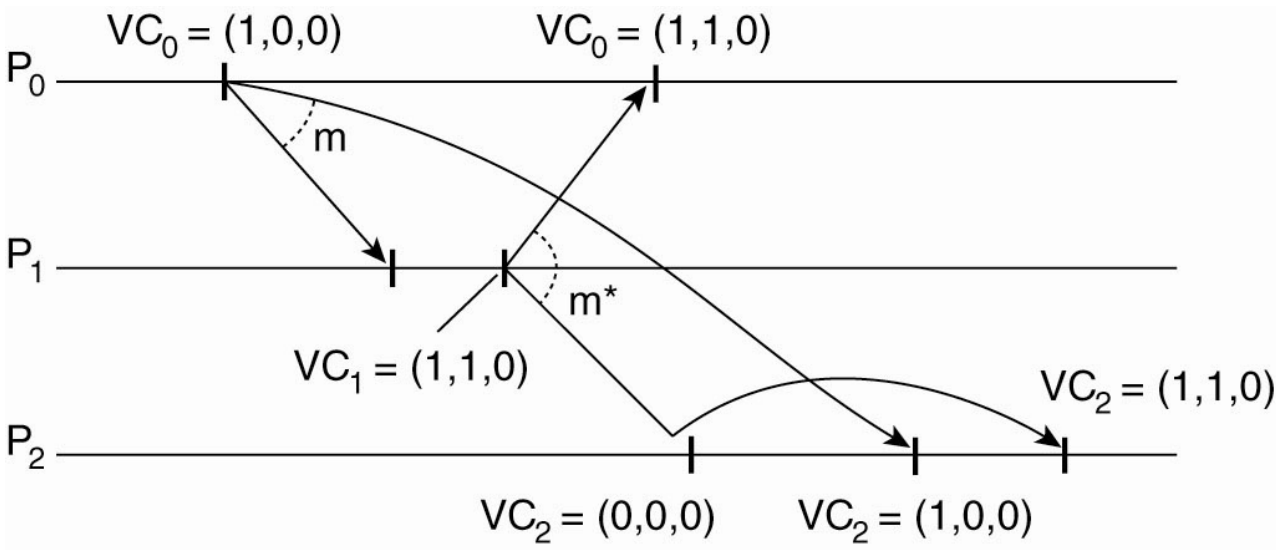
A1 → B2 → C2

B3 → A3

C2 → A4

B2 and A2 are *concurrent*

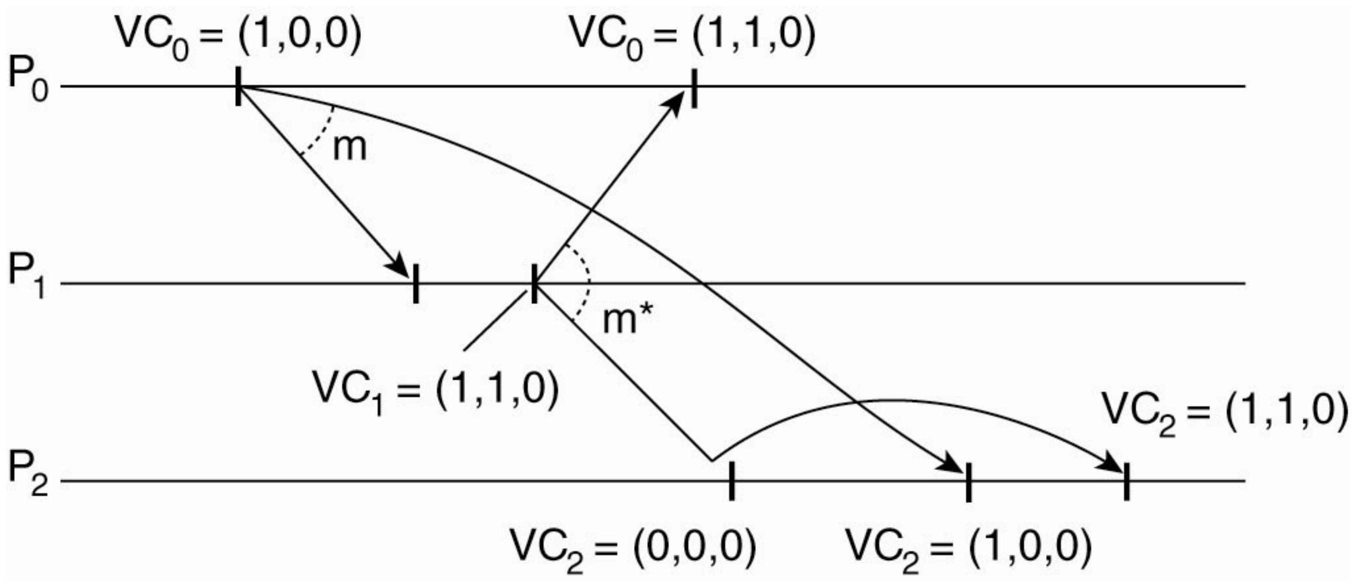
Enforcing Causal Communication



A message is delivered only when all messages that casually precede it have also been received as well.

- P0 sends message m (1,0,0) to P1 and P2. Message to P2 is delayed
- After P1 receives m, it generates m* to P0 and P1.
- Message m* (1,1,0) arrives earlier at P2 compared to message m (1,0,0).
- m* gets delayed behind m at P2 to ensure casually ordered communication

Enforcing Causal Communication



P_i sends message m to P_j with vector timestamp ts(m).
Message delivered if following two conditions are met:

1. ts(m)[i] = VC_j[i] + 1 [m is the next message P_j expects from P_i]
2. ts(m)[k] <= VC_j[k] for all k!=i [P_j has seen all message seen by P_i when it sends message m]