LECTURE 8: SAFE ACCESS TO DISTRIBUTED SHARED RESOURCES: TIME, SYNCHRONIZATION, REPLICATION & CONSISTENCY

Lecture Contents

- Introduction
- Time in Distributed Systems:
  - Physical Clocks; Logical Clocks: Totally Ordered Multicast
  - Lamport's Algorithm: Vector Clocks: Causally Ordered Multicast
- Mutual Exclusion in Distributed Systems:
  - Centralized & Decentralized Solutions
  - Election Algorithms
- Consistency Algorithms:
  - Sequential & Continuous Consistency
  - Causal Consistency
  - Client-Centric Consistency
- Replication & Caching
  - Client- & Server-initiated caching

Introduction

- DS essential in everyday life but has unique challenges, e.g. synchronizing data & resolving conflicts.
- Must replicate content but such replicas must be kept consistent with each other.
- Saw above how processes communicate – related to this is how they cooperate & synchronize with each other.
- Here, mainly look at how processes can synchronize:
  - So, vital that multiple procs don't simultaneously access shared resource, but cooperate to grant each other temporary *exclusive* access.
  - Multiple processes may also need to agree on *event orderings*, e.g. message from process P sent before/ after another from process Q
- Synchronization in DS thus much harder than synchronization in uniprocessor or multiprocessor systems.
- The problems & solutions are, by their nature, rather general, and occur in many different situations in DS.

SECTION 8.1: TIME IN DISTRIBUTED SYSTEMS

Time/Clocks

- *Physical clocks*:
  - *Problem*: Often simply need exact time, not just an ordering.
    - Previously solved by time in terms of *Sun Transits**
  - *Solution*: Universal Coordinated Time (UTC):
    - Based on number of transitions per second of caesium 133 atom**.
    - At present, real time is taken as average of ~50 caesium-clocks worldwide.
    - Introduces a *leap second* from time to time to account for fact that days are getting longer (e.g. due to tidal drag, orbital wobbles etc).
- Note: UTC is broadcast through SW radio & satellite. Satellites can give an accuracy of about ±0.5 ms.

*Time to reach highest point in sky
**Quite accurate

Time/Clocks (/2)

- *Physical clocks:*
  - *Problem*
    - Suppose have distributed system with a UTC-receiver in it ⇒ we still have to distribute its time to each machine.
  - *Basic principle*
    - Each machine has a timer generating interrupt H times per second.
    - There is a clock in machine p that ticks* on each timer interrupt.
    - Denote the value of that clock by $C_p(t)$, where t is UTC time.
    - Ideally, we have that for each machine p, $C_p(t) = t$, or, $\frac{dC}{dt} = 1$

*incs s/w clock counting no. of ticks since some (agreed on) time in the past

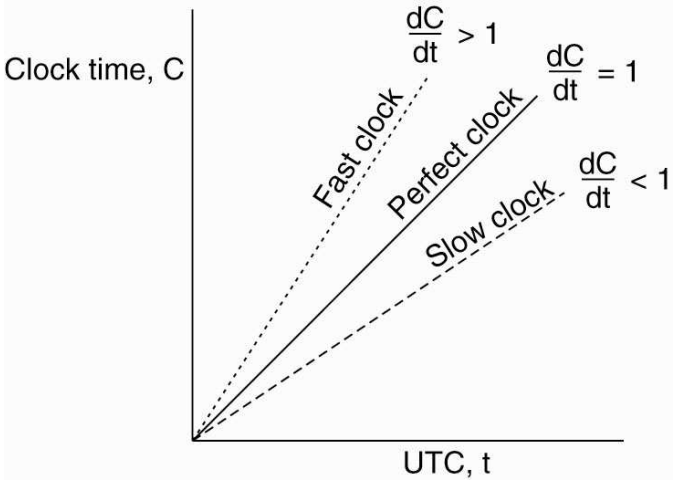
Time/Clocks (/3)

- *Physical clocks:*

- In practice: $1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$

  ρ is the clock's *skew*

- From the figure:
  - 2 clocks drifting from UTC in opposite directions in time Δt , may be $\leq 2\rho\Delta t$ apart



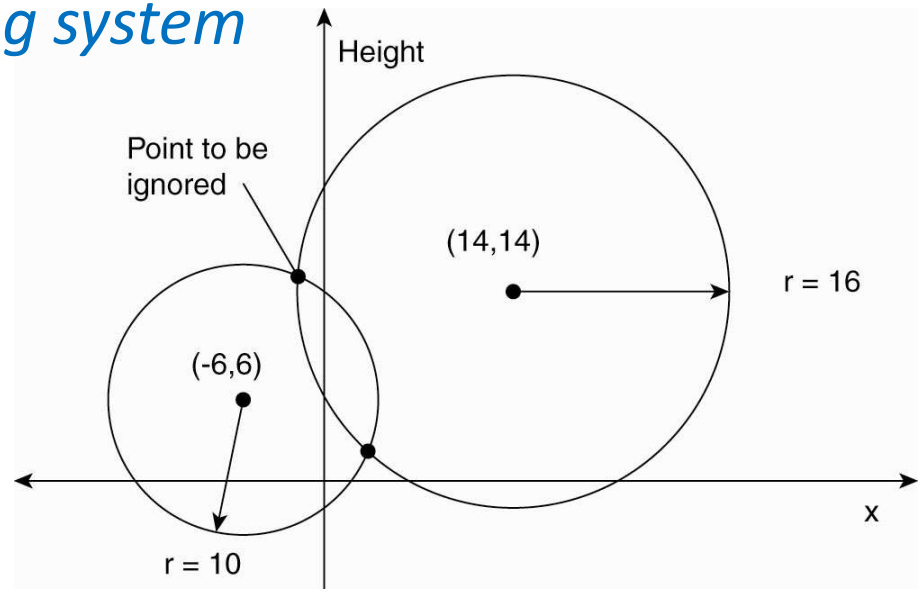
Relation btw clock time & UTC when clocks tick at different rates

- Goal:
  - Don't let 2 clocks differ by more δ than time units

  => synchronise every $\delta/(2\rho)$ secs
  - δ termed the *rate of drift*

Time/Clocks (/4)

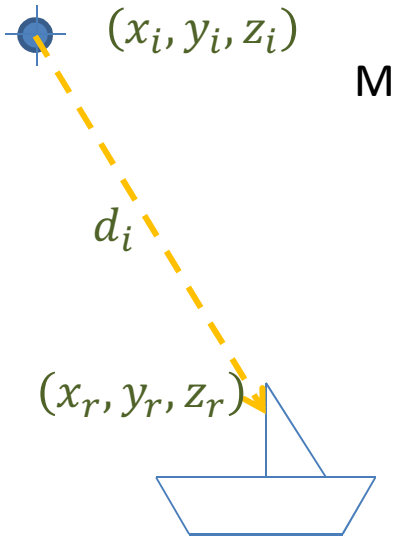
- *Global positioning system*



Computing a position in a 2D space

  - *Basic idea*: Can get accurate account of time as side-effect of GPS
  - *Problem*: Assuming satellite clocks are accurate & synchronized:
    - Takes time before a signal reaches receiver
    - Receiver's clock is definitely out of synch with satellite

Time/Clocks (/5)



Measured distance, $d_i = c$(time for light to go from satellite to ship)

So $d_i = c\Delta_i$

But $\Delta_i = (T_{now} - T_i) + \Delta_r$ (T_i is a satellite's timestamp)

$\Rightarrow d_i = c\Delta_i - c\Delta_r$

(Δ_i is measured time diff,
Δ_r is correction for clock deviation)

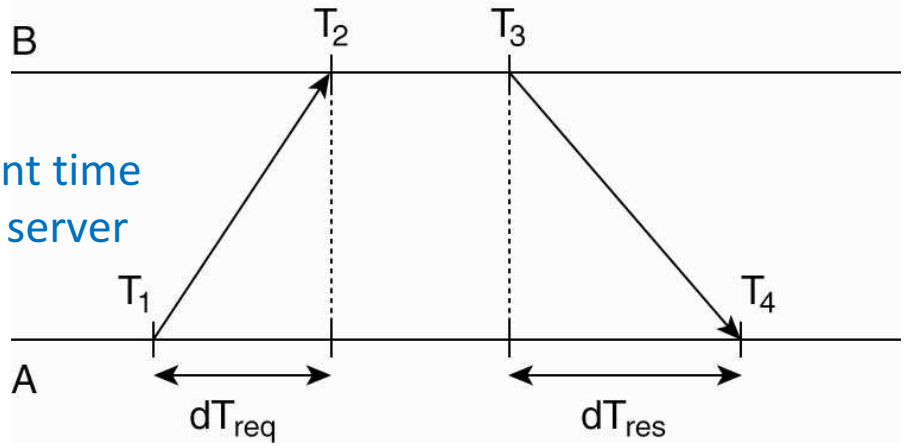
$$\Rightarrow d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

i.e. with 4 satellites, now have 4 equations in 4 unknowns

Time/Clocks (/6)

- *Clock synchronization principles*
  - *Principle I*
    - Every machine asks a time server for accurate time min every $\delta/(2\rho)$ seconds (Network Time Protocol).
    - Ok, but must measure round trip delay, incl interrupts & processing incoming messages.

Getting current time from a time server



  - *Principle II*
    - Time server scans all machines periodically, averages, informs each how to adjust its time wrt. its present time.
    - Ok, probably get every machine in sync. Needn't even propagate UTC time.
  - *Fundamental*: Have to take into account that setting time back never allowed ⇒smooth adjustments.

Time/Clocks (/7)

- *Logical Clocks: The Happened-before* relationship
  - *Problem*: First must introduce notion of ordering before can order anything.
  - The *happened-before* relation
    - If a, b are 2 events in same process, a comes before b, then $a \rightarrow b$*
    - If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$
    - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
  - *Note:* This introduces a *partial ordering* of events in a system with concurrently operating processes
    - For such a system, $x \rightarrow y$ is not true but neither is $y \rightarrow x$

*Read: "a happens before b"

Time/Clocks (/8)

- *Logical Clocks:*
  - *Problem:* How to keep a global view on system behaviour that is consistent with the *happened-before* relation?
  - *Solution:*
  - Attach timestamp $C(e)$ to each event e, with following properties:
    - $P1$ If a and b are two events in the same process, and $a \rightarrow b$, then require $C(a) < C(b)$.
    - $P2$ If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.
    - Everybody agrees on the values of $C(a), C(b)$.

Time/Clocks (/9)

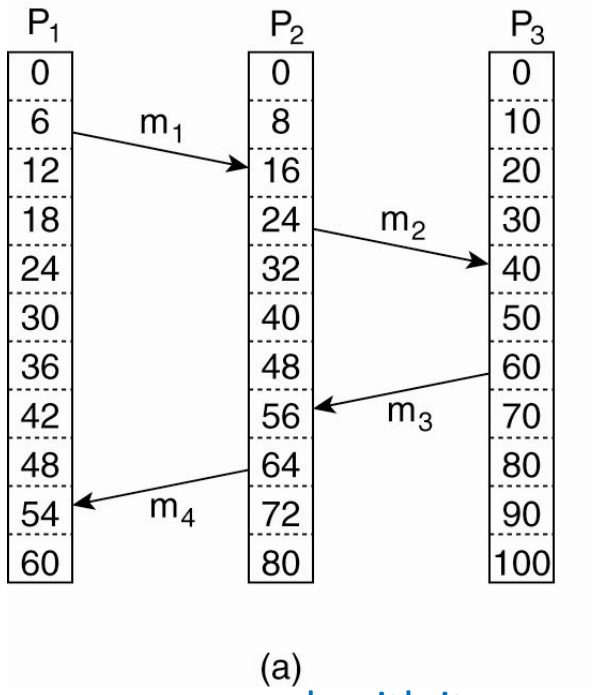
- *Logical Clocks:* Lamport's Algorithm
  - *Problem:*
  - How to attach a timestamp to an event when there's no global clock?

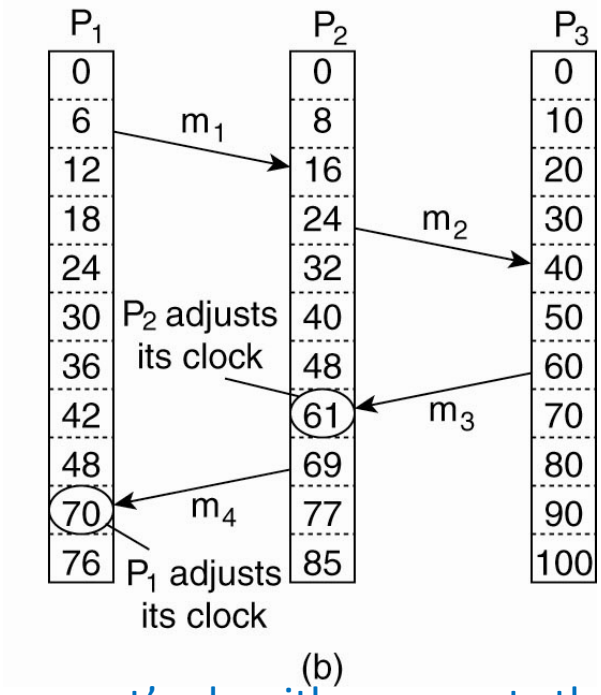
    ⇒ maintain a consistent set of logical clocks, one per process.
  - *Solution:*
  - Each process P_i has local counter C_i, adjusts it as per following rules:
    1. For any 2 successive events taking place within P_i, C_i is incremented by 1.
    2. Each time a message m is sent by process P_i, the message receives a timestamp $ts(m) = C_i$
    3. On receipt of message m by 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.
  - *Notes*
    - Property *P1* is satisfied by (1); Property *P2* by (2) and (3).
    - Can still occur that 2 events happen simultaneously.
    - Avoid this by breaking ties thro process IDs.

Note: C_i is no. of events that have occurred at i

Time/Clocks (/10)

- *Logical Clocks: Example*





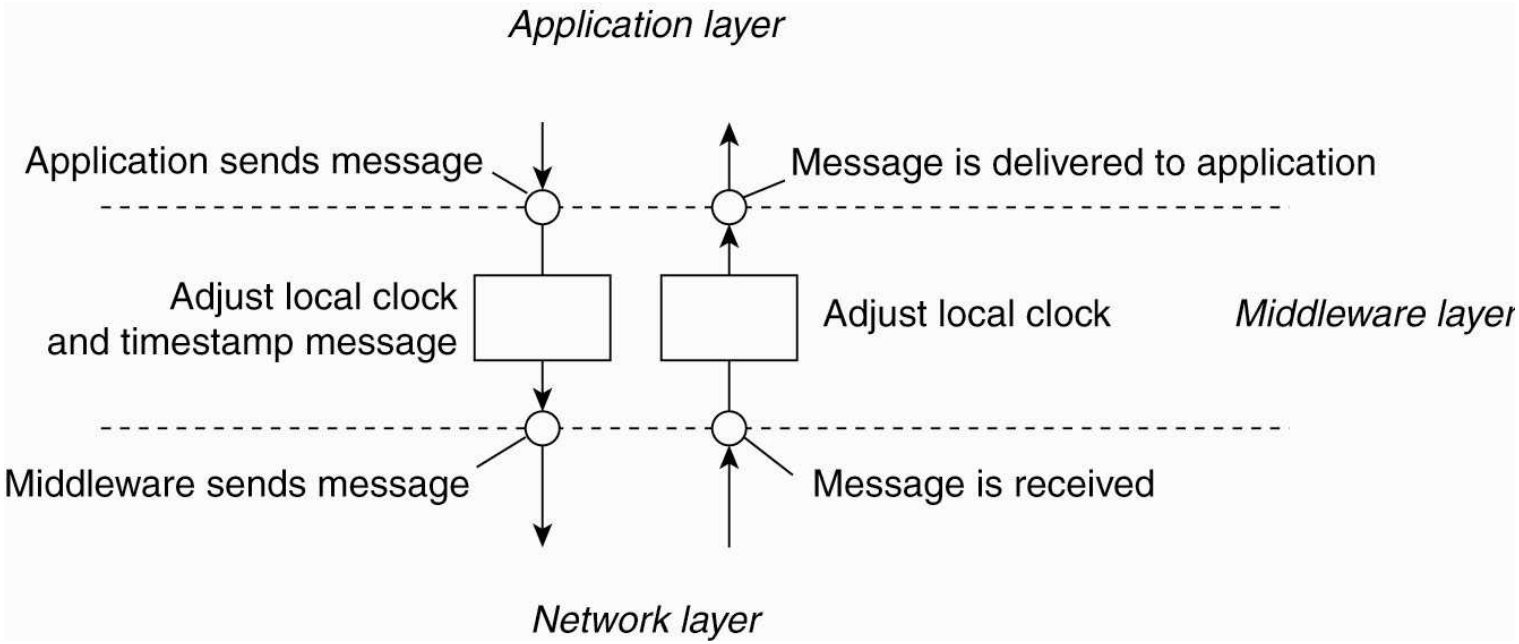
Three processes, each with its own clock. The clocks run at different rates

Lamport's algorithm corrects the clocks.

- *Impossibility:* In (a) m_3 arrives at P_2 before it was sent from P_3
- *Lamport's Algorithm:*
  - P_2 adjusts it's clock to 1 + sending time (=60) on arrival of m_3 from P_3

Time/Clocks (/11)

- *Logical Clocks:*
  - Adjustments take place in the middleware layer:



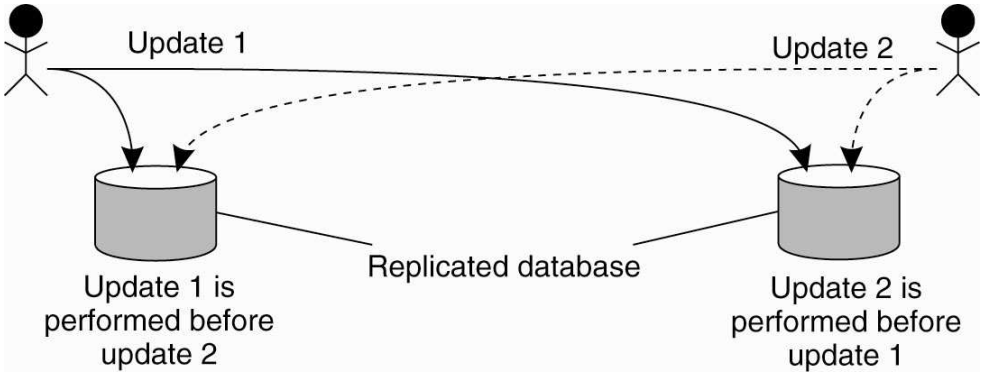
The positioning of Lamport's logical clocks in distributed systems

Time/Clocks (/12)

- *Logical Clocks:*

Example of Totally Ordered Multicast

- *Problem:*
- Sometimes must ensure that concurrent updates on a replicated DB are seen in the same order everywhere:
  - P1 adds $100 to an account (initial value: $1000)
  - P2 increments account by 1% interest in New York
- Two replicas



Updating a replicated database & leaving it in an inconsistent state.

- *Result:* In absence of proper synchronization:
  replica #1 ← $1111, while replica #2 ← $1110.

Time/Clocks (/13)

- Logical Clocks: *Example Totally Ordered Multicast*
  - *Solution:*
    - Process P_i sends timestamped message msg_i to all others.
    - The message itself is put in a local queue $queue_i$.
    - Any incoming message at P_j * is queued in queue j , according to its timestamp, and acknowledged to every other process.

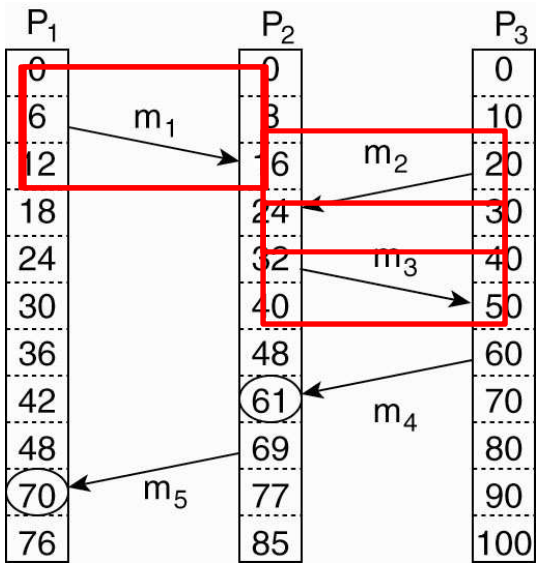
> P_j passes a message msg_i to its application if:
>
> (1) msg_i is at the head of queue j
>
> (2) For each process P_k , there is a message msg_k in queue j with a larger timestamp. This means that msg_i is at the head of j 's queue *and* has been acknowledged by other processes.

  - *Note:* We are assuming that communication is *reliable* & *FIFO ordered*.

* e.g. acknowledgement.

Time/Clocks (/14)

- *Logical Clocks: Example*
  - *Observation:*
    - Lamport's clocks don't guarantee that if $C(a) < C(b)$ that a causally preceded b



Event a : m_1 is received at T = 16;
Event b: m_2 is sent at T = 20

- From diagram, know that for P_2, $T_{rcv}(m_1) < T_{snd}(m_3)$ but what can be concluded in general from this statement?
- Know $T_{rcv}(m_1), T_{snd}(m_3)$ correspond to events that took place at P_2 but also know $T_{rcv}(m_1) < T_{snd}(m_2)$ but no causality there

Time/Clocks (/15)

- *Logical Clocks*:
  - *Problem with Lamport's Clocks:*
    - No guarantee that if $C(a) < C(b)$ that a causally preceded b
  - *Solution: Vector Clocks:*
    - Each process P_i has an array $VC_i[1 \ldots n]$, where $VC_i[j]$ denotes no. of events that process P_i knows have taken place at process P_j .
    - When P_i sends message m, it adds 1 to $VC_i[i]$, & sends VC_i along with m as *vector* timestamp $ts(m)$.
      - Result: on arrival, recipient knows P_i 's timestamp (i.e. the number of events at P_i that causally precede i)
    - When a process P_j delivers a message m that it received from P_i with vector timestamp $ts(m)$, it

      (1) updates each $VC_j[k]$ to $\max\{VC_j[k], ts(m)[k]\}$

      (2) increments $VC_j[j]$ by 1.

      i.e. max of current value of Vector Clock at P_j for k & no. of events at P_k before m sent by i
    - Put another way, $ts(m)[k]$ is a tuple consisting of a process's logical time & its last *known* time of process k in terms of no. of events that occurred at k
    - So with Vector Clocks know that if $VC(a) < VC(b)$ ie a causally preceded b

Time/Clocks (/16)

- *Vector Clocks:*

A Digression on Message Timestamps

  - If event a has timestamp $ts(a)$ then $ts(a)[i] - 1$ denotes number of events processed at P_i that causally precede a
  - Hence, when P_j gets a message from P_i timestamped $ts(m)$, it knows how many events have occurred at P_i that causally preceded the sending of m
  - This way, it knows how many events have occurred at other processes prior to the sending of m by P_i

Time/Clocks (/17)

- *Vector Clocks*: Causally Ordered Multicasting*
  - *Observation:*
    - Can now ensure that a message is delivered only if all causally preceding messages have already been delivered.
    - Note, in terms of messages sent and received $VC_i[j] = k$ means that P_i knows that k events have occurred at P_j
  - *Adjustment:*
    - P_i increments $VC_i[i]$ only on sending a message, & P_j "adjusts" $VC_j[k]$ (to $max\{VC_j[k], ts(m)[k]\}$ on receiving a message (i.e., effectively doesn't change $VC_j[j]$).

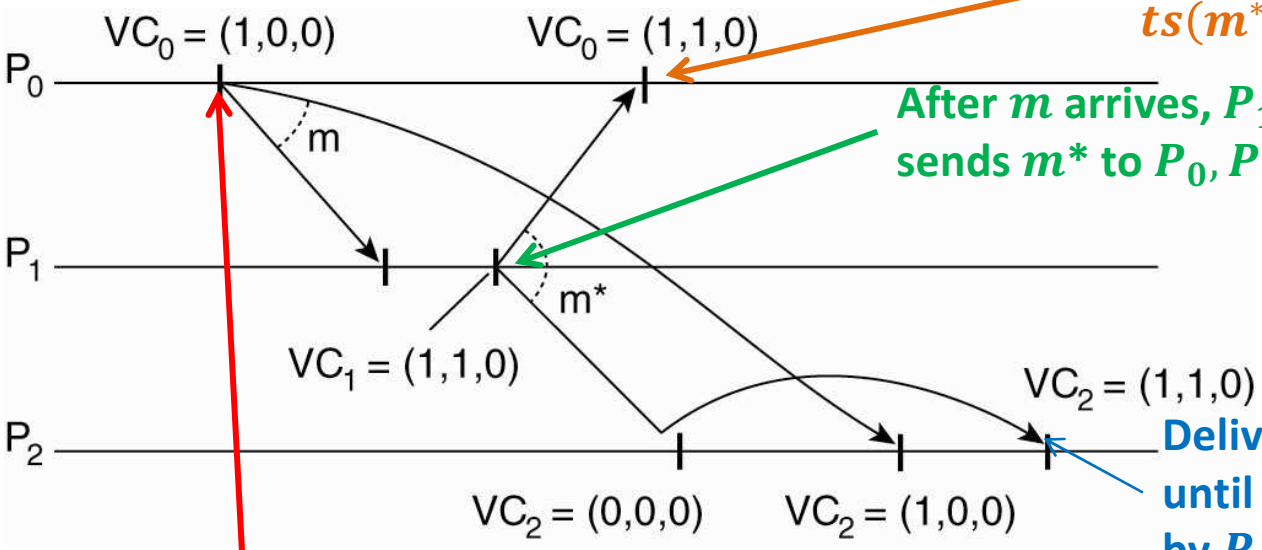
P_j postpones delivery of m until:

- $ts(m)[i] = VC_j[i] + 1$ (i.e. m is next message P_j expects from P_i)
- $ts(m)[k] \leq VC_j[k]$ for $k \neq i$. (i.e. P_j has seen all messages seen by P_i when P_i sent m)

* Not as strong as *Totally Ordered Multicasting*.

Time/Clocks (/18)

- *Vector Clocks*: Example 1
  - Recall each time message m is sent by process P_i, the message receives a timestamp $ts(m) = C_i$ (C_i denotes no. of events at occurred at P_i)
  - Thus when P_j receives m from P_i it knows about the number of events that have occurred at P_i before the sending of m.



P_0 delivers m^* cos $ts(m^*) = VC_0[1] + 1$

After m arrives, P_1 sends m^* to P_0, P_2

Delivery of m^* delayed by P_2 until m is received & delivered by P_2 's application layer

At $(1, 0, 0)$ local time P_0 sends message m to P_1, P_2

$$ts(m) = (1,0,0) \Rightarrow VC_1(1,1,0)$$
$$ts(m^*) = (1,1,0) \Rightarrow VC_0(1,1,0)$$