Time Synchronizatio

Unit Coordinator
Dr Nguyen Tran
School of Computer Science

Time matters…ex. (1)

- A bank replicates two copies of an account database in New York City (NYC) and San Francisco (SF) so that a query is always forwarded to the nearest city while updates must be carried out at both sites

- In a centralized system,
  - Process A makes a system call to get the time.
  - Later, process B makes a system call for time, which is will be higher or equal to the time returned to process A.

SF NYC

- A client in SF adds $100 to his account which currently contains $1000 while the bank add 1% interest in NYC.
- Updating a replicated database and leaving it in an inconsistent state: $1111 in SF and $1110 in NYC
- The problem is that both sites perceive the updates in different order leading to inconsistencies

Time matters…ex. (2)

- In a distributed system, if there is no global agreement on time, each machine may have slightly different time.

- Compiler will not run as the time of last compiled version is greater than lat

Outline

- Time
- Physical Time
- Logical Time
- Multicast
- Mutual exclusion
- Leader election

Time

- How time is actually measured ?
  - Time has been measured **astronomically**

- **Solar day:** Interval between two consecutive events of the sun's reaching its highest apparent point in the sky = 24 hours = 3600 seconds.
- Basis for the Greenwich Mean Time (GMT) standard

Time

- Earth is slowing down. Period of earth's rotation is not constant
  - Due to tidal friction, atmospheric drag.
- Astronomers introduced **Mean solar second** (averaged over large # of days)
- Physicists took o ion of the **atomic clock.**
  - By counting the transitions of the ce

First atomic clock [http://www.npl.co.uk/people/louis-essen]

Time

- **Second** = Time it takes the cesium 133 atom to make exactly 9,192,631,770 transitions.

- Major laboratories in the world periodically update the Bureau of International d how many times its clock has ticked. duce International Atomic Time (TAI).

- 9,192,631,770 number of transitions was selected to make the atomic second was equal to mean solar second.
  - Can you think of any potential problem ?

Time

- Atomic seconds are of constant length, unlike solar seconds.
- **Leap seconds** are introduced when necessary to keep in phase with the sun.

- If the discrepancy between atomic time and solar time grows to 800msec, BIH introduces a leap second.
- This corrected time system is called **Universal Coordinated Time (UTC).**

Leap Seconds

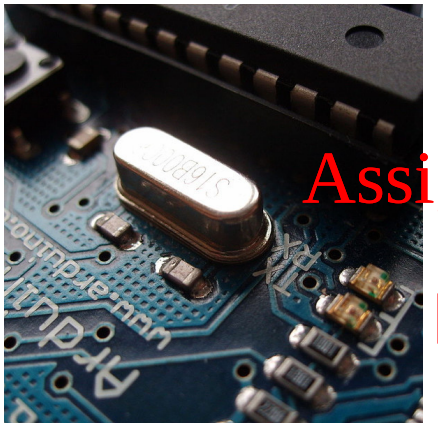
- Computerized systems could malfunction if leap seconds are not handled properly.
  - RedHat crash in 2012. THE INSIDE STORY OF THE EXTRA SECOND THAT CRASHED THE WEB, [https://www.wired.com/2012/07/leap-second-glitch-explained/]
- Operating systems are to account for leap seconds as
- Electric power companies raise their frequency to 61Hz to synchronize the timing of their 60Hz clocks to UTC.
- 37 leap seconds were added so far.
  - The most recent one was on 31/12/2016

Accurate time broadcasters

- Atomic clocks
  - Expensive to have one in every computer
- GPS
  - 29 satellites circ
  - Originally deve
  - Each GPS satellite has up to four at
  - Satellites continuously broadcasts it time
- Short radio wave stations transmit a short pulse at the start of every UTC second
  - NIST WWV station from Fort Collins, Colorado

Computer Clocks (Timers)

- Precisely machined quartz crystal.



- When kept under tension, quartz crystals oscillate at the defined frequency depending on;
  - The kid of crystal
  - How it is cut
  - The amount of tension

Computer Clocks (Timers)

- Hardware clock
  - At each clock tick, one is added to the time stored in a battery-backed up CMOS RAM
  - When the CPU is turned off, the time in the CMOS RAM keep being incremented
- Software clock
  - It starts running the hardware clock
  - Used by the operating system to in approximation of time
- Multiple CPUs means multiple clo
- The frequency of two crystals cannot be exactly the same
- Software clocks progressively get out of synch
- The difference returned by two clock values is called **clock skew**

Computer Clocks (Timers)

- Real timers cause an interrupt (ticks) H times every second.
  - e.g. H=60 should generate 216,000 ticks per hour. In practice, it will be in the range of 215,998 – 216,002.
  - Maximum drift rate is specified by the manufacturer
- The relation betw and UTC when clocks tick at different rates (Ideally, C(t) = t)
- The *skew* of the clock is dC/dt-1

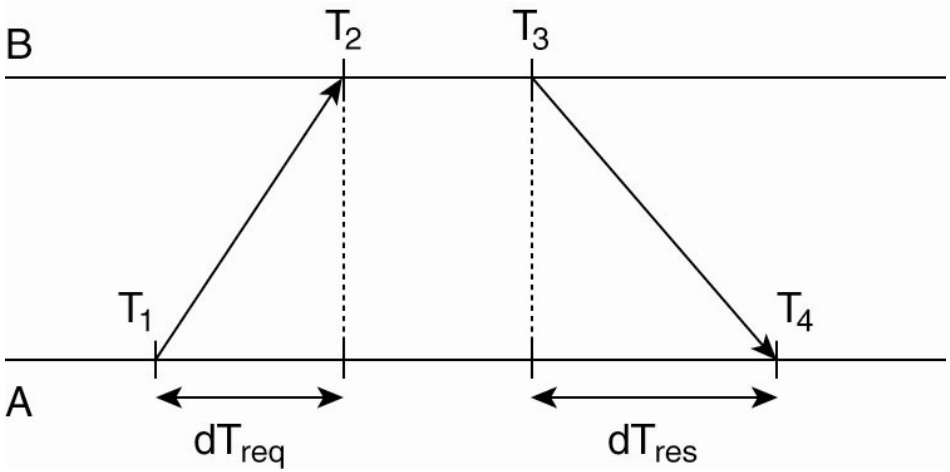
- Two problems
  1. **How do we synchronize them with real-world clocks ?**
  2. **How do we synchronize the clocks with each other ?**

Algorithm 1 : Network Time Protocol

- Clients synchronize its time with a time server which have an accurate time, e.g. an atomic clock.
  - How do we account for propagation delays ?

- Cristian's algorit

1. A sends a message t ocal clock value
2. Upon reception, B re l clock
3. B returns a response message to A with T2 time T3 of its clock
4. Upon reception, A records the current time ock



Algorithm 1 : Network Time Protocol

Cristian's algorithm [1989]

- If clock rate increases monotonically (without even decreasing) and
- If both message transmissions take roughly the same time
- *Round-trip time* = $(T_2 - T_1) + (T_4 - T_3)$
- *Offset* of A relative to B = $T_2 - T_1 - RTT/2$
- A can re-adjust its ti
- This result is not acc pproximation

Algorithm 1 : Network Time Protocol

- NTP is set up pair-wise between servers
  - i.e. B will also probe A for its current time

1. Run Cristian's algorithm 8 times and records the 8 output pairs $<Offset, RTT/2>_i$ $(0 \leq i < 8)$
2. The pair with the minimum RTT/2 is chosen as the most precise offset

- To avoid servers wi
  adjusting their clocks, Servers are
  divided into **strata**
- A server having an atomic clock
  operates at stratum 0
- A server synchronized with a server at
  stratum j operates at stratum j+1

- Accuracy is in the range of 1-50msec

https://www.ntp-zeit.de/index-en.htm

Algorithm 2 : The Berkeley Algorithm

- What if there is no server of stratum 0? Can we still relatively synchronize?
- Server is active, it starts the algorithm periodically to resynchronize nodes

1. Time server (a.k.a., daemon) sends a "what time is it?" request to all nodes

2. The nodes answ lue
3. Once the server has received v e nodes

   - It computes an average value
   - It sends it to the nodes for them to adjust

- The time can be far from the UTC, hence the time is not absolutely synchronized
- This relative synchrony is sufficient in many cases

Algorithm 2 : The Berkeley Algorithm

- (a) the time daemon sends a request to all nodes
- (b) the nodes answer the time daemon about their local clock
- (c) the server computes the average and tells nodes to adjust their clock

Pros and Cons of Physical Time

- Pros: very powerful
  - Universal representation of the occurrence time of all distributed events
  - All events are made comparable (total order)

- Cons: hard to achieve
  - This physical time instruments are difficult to synchronize
    - They require t upper-bounded duration
    - They sometim ilar (also lower bounded)
  - In large-scale system, the message delay
    - High (in seconds) and the synchroniza d by message delays
    - Heterogeneous (LAN: milliseconds, W rders of magnitude far apart
    - Unpredictable, unbounded message delays and message can be delivered in disorder

- Let's look at logical time instead, where events may not always be comparable (partial order)

Logical time

- It is not always necessary to synchronize all nodes the same as physical (real) time
  - E.g. it is adequate that two nodes agree that input.o is outdated by a new version of input.c to properly compile.
- Keeping **track of the order of events** is what matters !
- Lamport (in 1978) showed that;
  - If two processes do not interact, it i that their clocks be synchronized
  - What usually matters is that they agree on the order in which events occur, not what time it is.

Logical time

- Definition of the relation called "happens-before"

- "*a happens before b*", denoted by a → b, is true if:
  - **a** and **b** are events from the same process such that **a** occurs before **b**, **or**
  - if **a** is the event of the same message and **b** is the event process, **or**
  - it exists some event **c** such that **a** → i.e., transitive closure)

- If two events, x and y, happen in different processes that do not exchange messages (not even indirectly via third parties);
  - then x → y is not true, but neither is y → x.
  - In this case, x and y are said to be **Concurrent**

Lamport's logical clocks

- Goal: to make the "happens-before" relation visible to processes

- Key idea:
  - each process should keep track of the order in which events appear to take place locally
  - let's introduce a function logical clock C: events → integers

- **Lamport's logical**

  1. Each process p
  2. Before executing an event (i.e. sen to another process or an internal event), p_a increments C_a

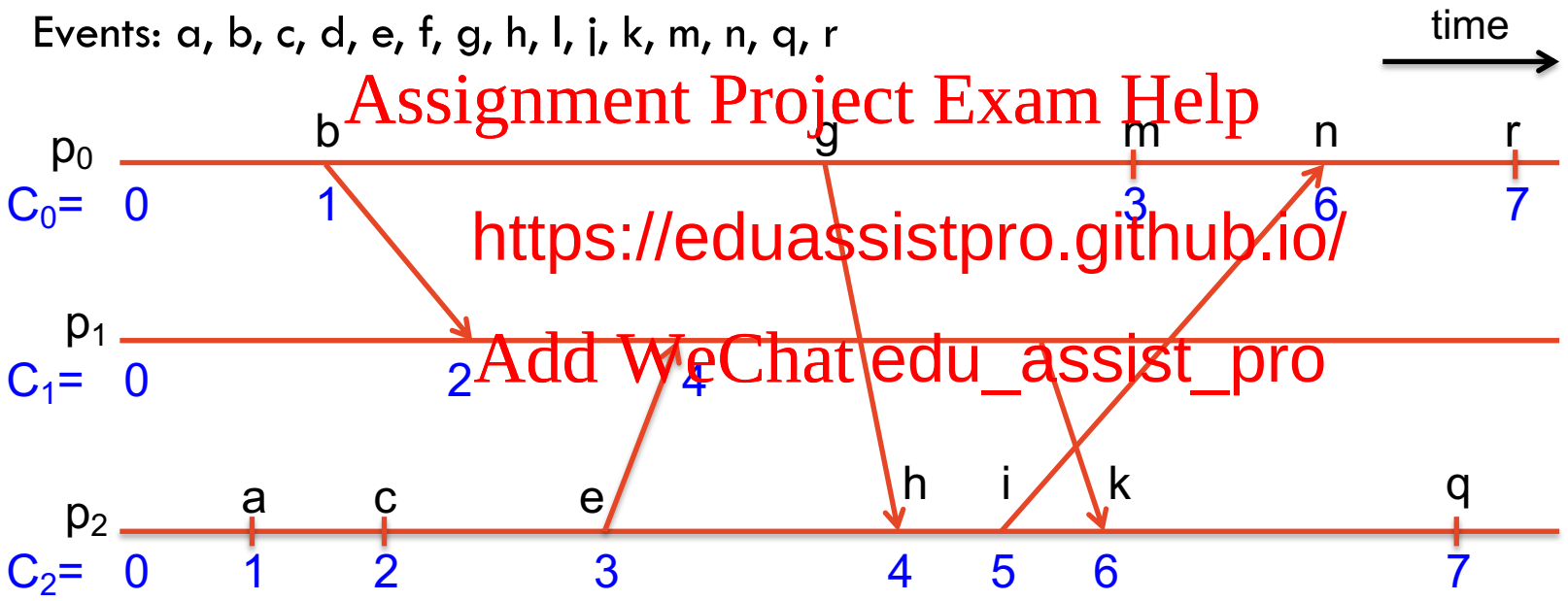
  3. When process p_a sends a message m to process p_b, it sends C_a as the time stamp of the message.
  4. Upon the receipt of a message, process p_b adjust its own local counter as $C_b \leftarrow max\{C_a, C_b\}$
  5. p_b executes step 2 and delivers the message to the application

- Result: If $P_a \rightarrow P_b$ then $C_a < C_b$

Lamport's logical clocks

Example:

- Processes: p_0, p_1, p_2
- Events: a, b, c, d, e, f, g, h, I, j, k, m, n, q, r

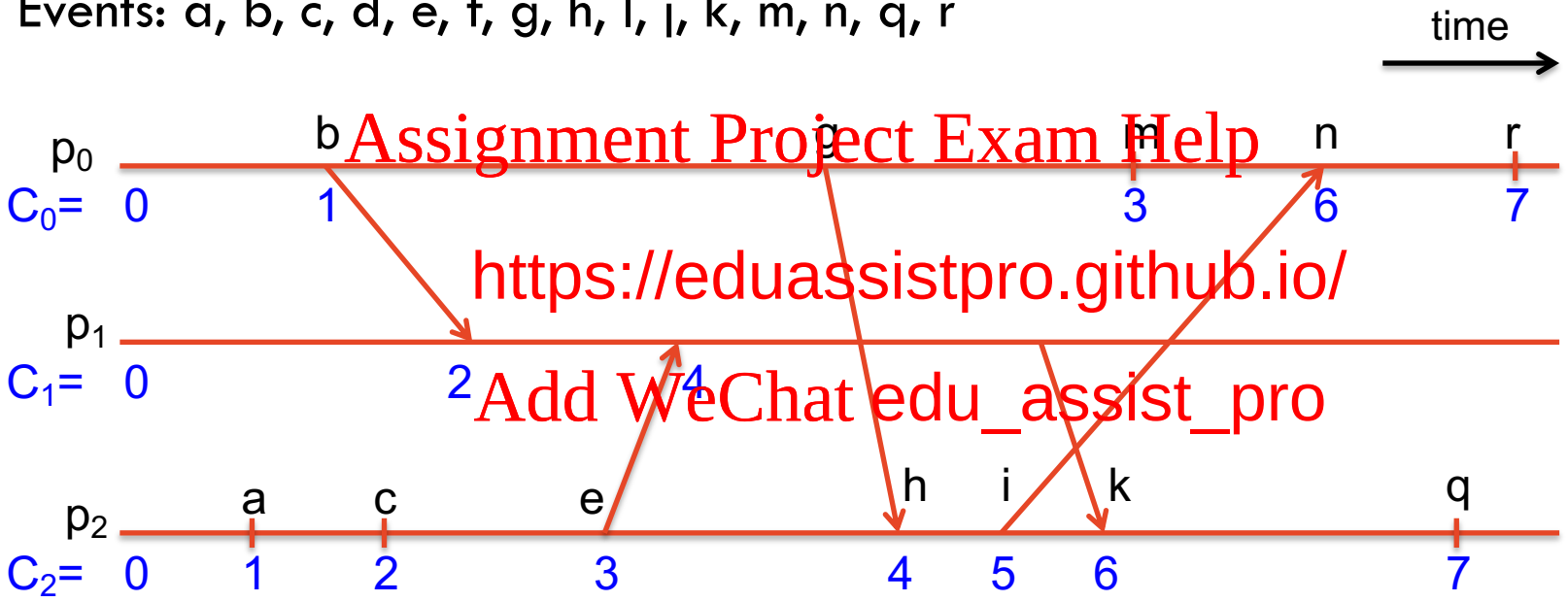


- c → n, j → k, e → h, etc.
- **a** and **b** are concurrent and **a** and **d** are concurrent but b → d

Lamport's logical clocks

Example:

- Processes: p_0, p_1, p_2
- Events: a, b, c, d, e, f, g, h, I, j, k, m, n, q, r



- $c \rightarrow n$, $j \rightarrow k$, $e \rightarrow h$, etc.
- a and b are concurrent and a and d are concurrent but $b \rightarrow d$

Even though $C_a < C_d$, we have that $a \not\rightarrow d$

Vector clocks

- Goal: to make also the "did-not-happen-before" relation visible

- Key idea:
  - each process should also maintain an approximation of the clock of other processes
  - let's introduce a function vector clock VC: events ↦ integers × processes

- **Vector clocks algorith**
  1. Each process p_a keeps a vector V for each process k.
  2. Before executing an event (i.e. se ge to another process or an internal event), p_a increments VC_a, i.e. $VC_a \leftarrow VC_a + 1$.
  3. When process p_a sends a message m to process p_b, it sends VC_a as the time stamp of the message, i.e. $ts(m)=VC_a$
  4. Upon the receipt of a message, process p_b adjust its own local counter as $VC_b[k] \leftarrow max\{VC_b[k], ts(m)[k]\}$ for each process k.
  5. p_b executes step 2 and delivers the message to the application

Vector clocks

Result:

Let VC(a) and VC(b) be the values chosen for events a and b,

We order vector clocks by comparing their coordinates

- One is lower than another, if each of its coordinates is lower than the corresponding c
  VC(a) ≤ VC(b) i VC(b)[i]
- One is strictly lower than another, i d different from the other
  VC(a) < VC(b) if VC(a) ≤ VC(b) and)

This algorithm guarantees that:

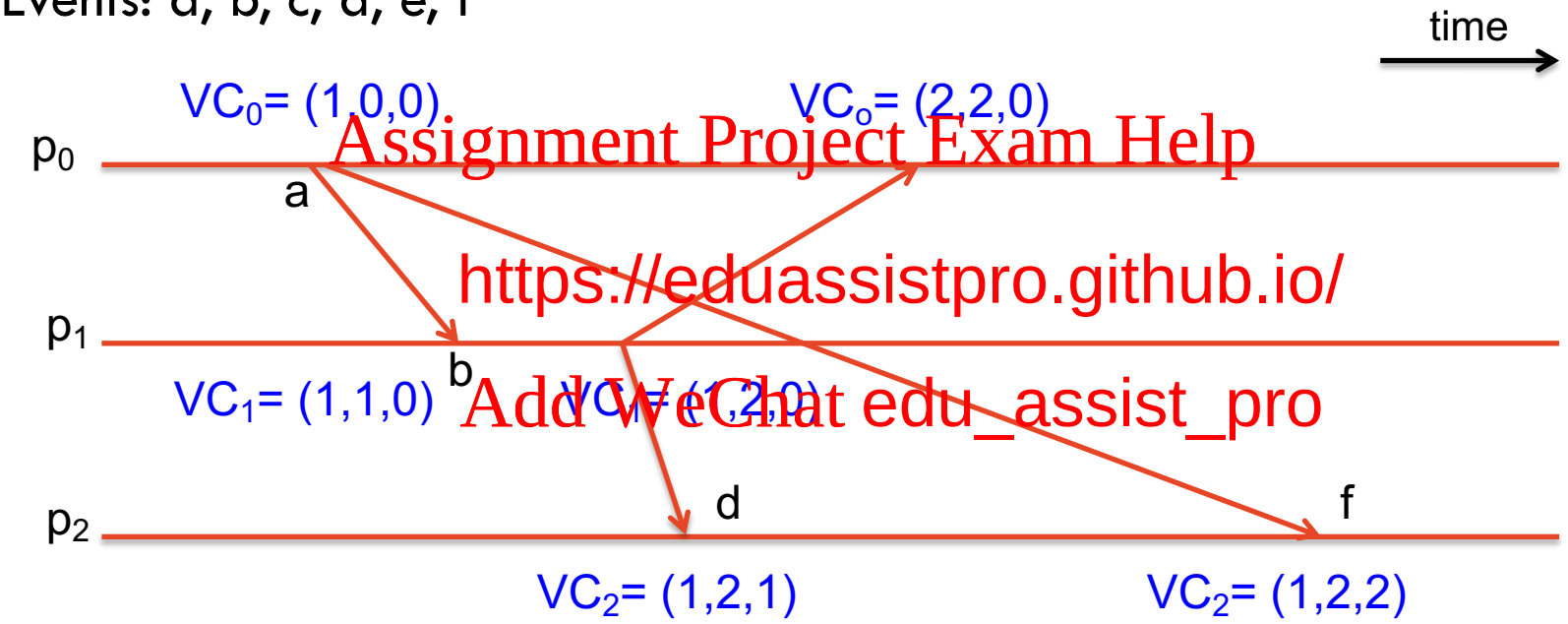
- If VC(a) < VC(b) then a → b
- If a → b then VC(a) < VC(b)

VC(a) < VC(b) if and only if a → b

Vector clocks

Example:

- Processes: p_0, p_1, p_2
- Events: a, b, c, d, e, f



- $a \rightarrow b$, $a \rightarrow c$, $a \xrightarrow{?} d$, etc.
- $e \not\rightarrow f$ and $f \not\rightarrow e$, $d \xrightarrow{?} e$

VC(i) < VC(j), if and only if $i \rightarrow j$

Logical time

- Clocks give an indication on the ordering of event by using logic instead of physical instrument
- Intuitively, if some event occurred before another then it should be marked by a lower (logical or vector) clock value
- If some process o strictly lower logical clock than another
  - then the process can conclude *a cann* *b*
  - the process cannot distinguish:
    - Whether a happened before b
    - Or a and b are concurrent
- If some process observes that an event a has a strictly lower vector clock than another event b
  - then the process can conclude a cannot happen after b
  - And more particularly, that *a happened before b*

Multicast

Definitions

- Ordering of messages:
  - *Totally ordered*: for all messages m1 and m2 and all process pi and pj, if pi delivers m1 before it delivers m2, then m2 is not delivered at pj before m1 is
  - *Causally ordered*: for all messages m1 and m2 and every process pi, if m1 happens before i before m1 is
- Reliability:
  - *Integrity*: every message received w nt; no message is received "out of thin air"
  - *No duplicates*: no message is received more than once at any single process
  - *Liveness*: all messages broadcast by a process are eventually received by all processes
- In the following we assume reliability and we ensure ordering of messages

Multicast

First-in first-out (FIFO) multicast

- Goal: broadcasting messages that got delivered in the same order at all nodes
- Each process ma ority queue of messages (waiting to be d

1. A process p_i multicasts a message e number corresponding to its current counter value and then i counter by 1
2. The recipient of a message from p_i with sequence number T puts the message in the queue according to number T and waits to perform the FIFO delivery until it has done so for all messages from p_i with sequence numbers less than T

Multicast

Totally ordered multicast

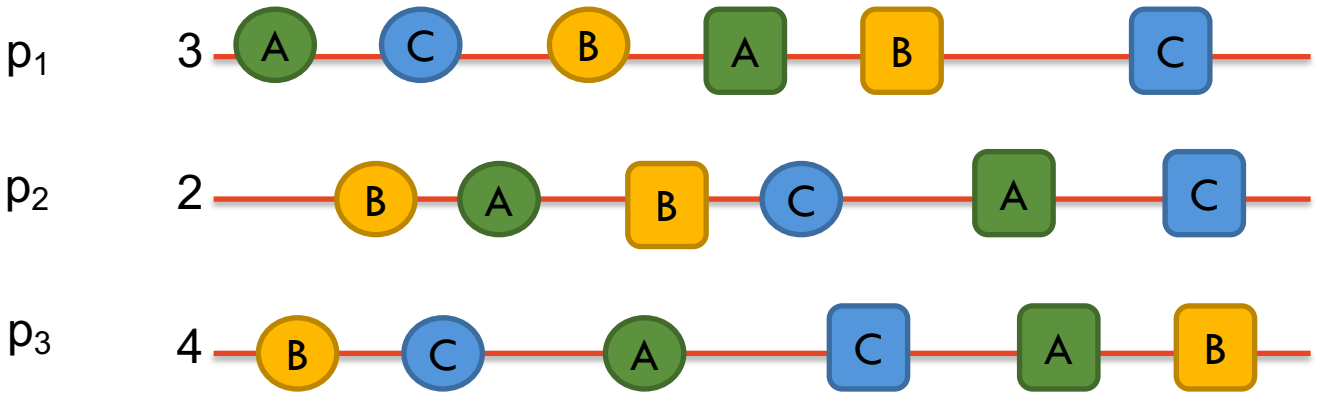
- Goal: broadcasting messages that got delivered in the same order at all processes
- Each process maintains a logical clock and a priority queue of messages (waiting to be delivered to the top layer)

1. A process p_i increments its logical clock asts a message with the current logical clock as the timestamp m
2. Any incoming message is queued in a p ccording to its temporary timestamp, and acknowledged using a FIFO-multicasts message to every other processes (including the sender)
3. Once all processes have acknowledged, the sender sets the definitive clock as the current clock it has for this message
4. A message in the queue is delivered once there is a message, acknowledged by all, from each process with a larger timestamp in the queue

Multicast

Totally ordered broadcast algorithm (con't)

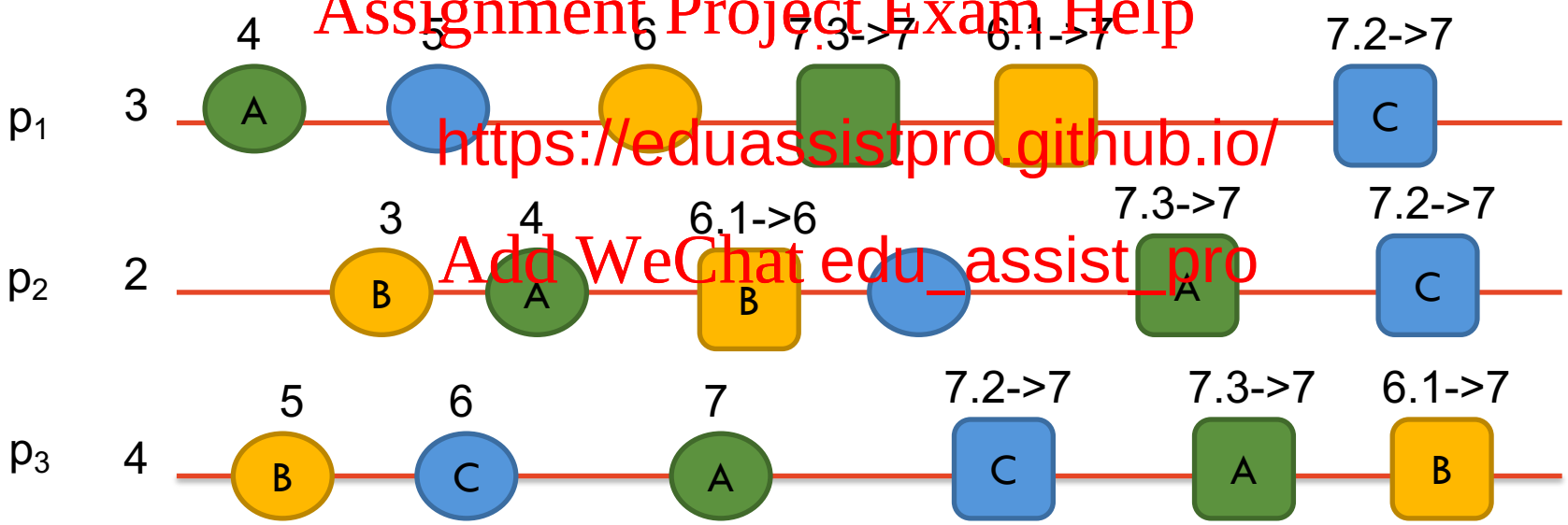
- Initial clocks are 3, 2, 4 at resp. processes 1, 2, 3; messages are A, B, C with timestamp = 1, 1, 1
- Circles represent message (msg) reception
- Squares represent the last ack reception for a particular message
- Upon the receipt of ust its own local clock *Ci* ← max{*Ci*, *ts*(*m*)} and i
- Upon the receipts of an acknowledgem s own local clock Ci ← max{Ci, ts(m)} and does not increment t



Multicast

Totally ordered broadcast algorithm (con't)

- Use the notation <received ts(m).received process id> → < new local clock> to illustrate the clock value after the reception of an ack.
- 7.3 represents the message was received at process 3 at clock 7. Then, max{local, received} is taken to reach the new local clock.



- Messages are delivered in the order of their clocks, B, C, A, everywhere (lower timestamps and IDs order)

Multicast

Causal ordered multicast

- Goal: broadcasting messages that got delivered only if all causally preceding messages have already been delivered
- Each process maintains a vector clock VC_i (similar as before) and a priority queue o delivered to the top layer)
- Process p_i increments $VC_i[i]$ only a message (not when receiving) and p_i adjusts VC_i whe a message.
- p_i postpones delivery of m from p_i until both conditions are met:
  1. $ts(m)[i] = VCj[i] + 1$
  2. $ts(m)[k] \leq VCj[k]$ for $k \neq i$

Multicast

Causal ordered multicast (con't)

- Take $VC_2 = [0,2,2]$, $ts(m) = [1,3,0]$ from p_0. What information does p_2 have, and what will it do when receiving m (from p_0)?

Mutual exclusion

- Concurrent access to the same resource may corrupt the resource, or make it inconsistent.

- Problem: Multiple system want exclusive access to some resource

Mutual exclusion

Centralized algorithm

- Fair and easy to implement
- Requires only three messages per use of resource (request, grant, release)
- No process ever waits forever → no starvation
- But, single point of failure

Mutual exclusion

Decentralized algorithm

- Principle. Assume every resource is replicated n times, with each replica having its own coordinator ⇒ access requires a majority vote from m > n/2 coordinators. A coordinator always responds immediately to

- Assumption: When a coordinator ill recover quickly, but will have forgotten about permissions it had granted.

- Problem: A coordinator may grant permission to some node whereas it already did it to another node

Mutual exclusion

Distributed algorithm - Ricart and Agrawala algorithm

- Principle. The same as before except that acknowledgments are not sent. Instead, replies (i.e., grants) are sent only when
  - The receiver has no interest in the shared resource or
  - The receiver is waiting for the resource, but has lower priority (known through comparison of ti

  - Two processes want to access a shared resource at the same moment
  - Process 0 has the lowest timestamp, so it wins.
  - When process 0 is done, it sends an OK also, so 2 can now go ahead.

Mutual exclusion

- **Distributed algorithm - Ricart and Agrawala algorithm**

- Single point of failure is replaced by n failures
- More message exchanges
- Getting permission from everyone is really an over-kill

Mutual exclusion

Token ring algorithm

- Principle. Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to accessing the resource (if it wants to).

Mutual exclusion

Comparison of the last four solutions

- Messages and delay are necessary to get granted the permission to access the resource (this means entering the critical section) or to release the permission (exiting it)

Conclusion

- Time matters in distributed systems
- Physical time
  - It is expensive to have accurate physical time on every computer
  - Synchronization with someone who knows the accurate time
    - Network Time Protocol and Berkeley Algorithm
- Logical time
  - Keeping **track of the order of events** is
  - To make the "happens-before" relation sses → Logical clocks
  - To make also the "did-not-happen-bef le → Vector clocks
- Ordered multicasting deliver messages in the same order at all nodes
- Providing exclusive access and leader election are practical challenges for many distributed systems

What's Next ?

- Tutorial 5 on Wednesday.

- Assignment 1 is available – Good Luck !

- Next week: **Con** ems

- See you all next week !