TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Synchronization in Distributed Systems

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Time in distributed systems

- In a DS, processes are executed on network-connected nodes that cooperate to complete a computation and communicate via message exchange
- Observations:
  - Algorithms require synchronization
  - Processes running on different nodes in the DS must have a common notion of time to perform synchronized actions based on time
  - Algorithms require event ordering, meaning determining which event occurred before another
  - In DS, time is important but problematic

Time in distributed systems

- In a centralized system, it is possible to determine the order in which events occurred
  - Shared memory and single clock
- In a DS, it is impossible to have a single physical clock common to all processes
- Yet, the global computation can be viewed as a total order of events, by considering the time at which events were generated
- For many distributed algorithms, it is crucial to determine this time or, at the very least, establish the ordering of events: **how?**

Time in distributed systems: solutions

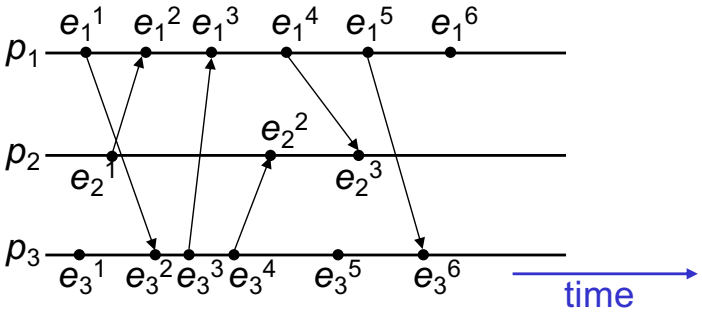
- Solution 1: synchronizing **physical clocks**
  - A physical clock counts the number of seconds
  - Each node in the DS adjusts its physical clock to be consistent with the clocks of other nodes or with a reference clock
- Solution 2: synchronizing **logical clocks**
  - A logical clock counts the number of events
  - In a DS, physical clock synchronization is not necessary; only the ordering of events is required (proposed by Leslie Lamport)

Model of distributed computation

- Components of DS: *N* processes and communication channels
- Each process p_i ($1 \le i \le N$) generates a sequence of events
  - e_i^k: *k*-th event generated by p_i
  - Internal events: state changes in the process
  - External events: send/receive messages
- Evolution of computation represented with space-time diagram

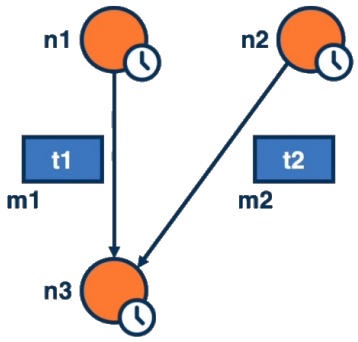
  $\rightarrow_i$: ordering relation between two events in p_i

  $e \rightarrow_i e'$ if and only if *e occurred before e'* in p_i



Timestamping

- Each process labels events with a timestamp
- Simple solution: each process labels events according to its own physical clock
- Does this work?
  - We can reconstruct the ordering of events on the same node
  - But what about the ordering of events on different nodes?
- In a distributed system, it is impossible to have a single physical clock shared by all processes



Synchronous vs. asynchronous DS

- Properties of a **synchronous DS**
  1. There are constraints on the execution speed of each process
     - The execution time of each step is limited, with both lower and upper bounds
  2. Each message transmitted on a communication channel is received within a limited time
  3. Each process has a physical clock with a known and limited clock drift rate from the real clock
     - Clock drift: deviation of clock from the true time over a period

- In an **asynchronous DS**
  - *There are no constraints* on the execution speed of processes, the message transmission delay, or the clock drift rate

Solutions to synchronize clocks

- First solution
  - Synchronize physical clocks of processes with some approximation using synchronization algorithms
  - Each process labels events with the value of its physical clock (which is synchronized with the other clocks with a certain degree of approximation)
  - Timestamping based on physical time (*physical clock*)

- Is it always possible to keep the approximation of physical clocks limited?
  - No, in an asynchronous DS
  - In an asynchronous DS, timestamping cannot be based on physical time; instead, it must be based on logical time (*logical clock*)

Physical clock

- At real-time instant t, the OS reads the time from the hardware clock $H_i(t)$ of the computer and generates the software clock $C_i(t) = aH_i(t) + b$ which approximately measures the physical time instant t for p_i
  - E.g., $C_i(t)$ is a 64-bit number that represents the number of nanoseconds elapsed from a reference instant until time t
  - In general, the clock is not perfectly accurate and may differ from t
  - If C_i behaves well enough, it can be used for event timestamping in p_i
- What should the clock resolution be in order to distinguish two events?
  - Smaller than the time difference ΔT between two relevant events

Physical clock in a DS

- In a SD physical clocks are different and can have different values
- *Skew*: instantaneous difference between the values of two clocks
- *Drift*: clocks measure time at different rates (due to physical variations), so over time they diverge from real time
- *Drift rate*: difference per unit of time between a physical clock and an ideal clock
  - E.g., drift rate of 2 μsec/sec means that the clock increases by 1 sec + 2 msec every sec
  - Drift rate of standard quartz clocks: 10^{-6} s/s (~1 s in 11-12 days)
  - Drift rate of high-precision quartz clocks: 10^{-7} or 10^{-8} s/s

Universal Coordinated Time (UTC)

- International reference time standard
- Based on atomic time, occasionally corrected using astronomical time
  - 1 second is defined as the time is takes for a cesium-133 atom to complete 9,192,631,770 transitions
- Physical clocks using atomic oscillators are extremely accurate, with a drift rate as low as 10^{-13} s/s
- Atomic clock outputs are broadcast by terrestrial radio stations and satellites (e.g., GPS)
  - In Italy, Istituto Galileo Ferraris
- Nodes equipped with receivers can synchronize their clocks with these signals
  - Accuracy of signals from terrestrial radio stations: 1 to 10 ms
  - Accuracy of signals from satellites: 0.5 ms down to 50 ns

Synchronizing physical clocks

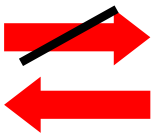
- How to synchronize physical clocks with the atomic clock or with each other?
- **External** synchronization: the clocks C_i (i = 1, 2, … N) are synchronized with a time source S (UTC), such that, for a given real-time interval I

  $|S(t) - C_i(t)| \leq \alpha$ for $1 \leq i \leq N$ in the interval I
  - The clocks C_i have an **accuracy** of α, where $\alpha>0$
- **Internal** synchronization: two clocks C_i and C_j are synchronized with each other, so that

  $|C_i(t) - C_j(t)| \leq \pi$ for $1 \leq i,j \leq N$ in the interval I
  - The clocks C_i *and* C_j have a **precision** of π, where $\pi>0$

Synchronizing physical clocks

Internal synchronization



External synchronization

- Internally synchronized clocks are not necessarily synchronized externally
  - All clocks can collectively drift from an external source while still remaining synchronized with each other within a bound *D*
- If the set of processes is externally synchronized with accuracy of α, then it is also internally synchronized with a precision of 2α

Correctness of physical clocks

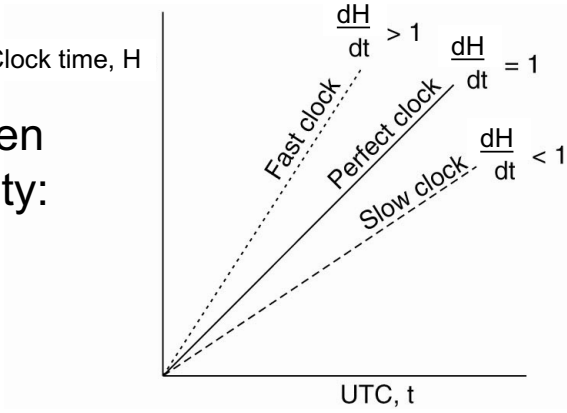
- A hardware clock *H is* correct if its drift rate is between $-\rho$ and $+\rho$ with $\rho > 0$
- If clock H is correct, the error made when measuring a real-time interval $[t, t']$ (with $t'>t$) is bounded:

$$(1 - \rho)\,(t' - t) \leq H(t') - H(t) \leq (1 + \rho)\,(t' - t)$$

  - This avoids "jumps" in the clock value

- For a software clock C, it is often sufficient to require monotonicity:

$$t' > t \Rightarrow C(t') > C(t)$$



When to synchronize physical clocks?

- Due to the drift rate, after a certain time, the clocks in a DS will become misaligned again → periodic synchronization is required to realign them
- Let's consider two clocks with the same maximum drift rate ρ from UTC
- Assume that after synchronization, the two clocks drift in opposite directions from UTC
  - After Δt from synchronization, they will have deviated by at most $2\rho\Delta t$
- To ensure that the difference between the two clocks remains within a desired tolerance δ, they must be resynchronized at least every $\delta/2\rho$

Internal synchronization in synchronous DS

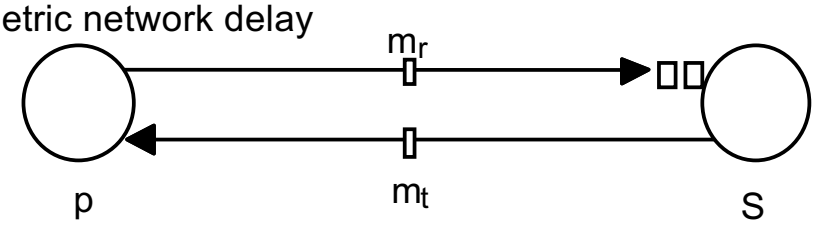
- Internal synchronization algorithm between 2 processes in a synchronous DS
  - p_1 sends its local clock value t to p_2 via a message m, with transmission time T_{trasm}
  - p_2 receives m and sets its clock to $t + T_{trasm}$
    - T_{trasm} is unknown, but since the system is synchronous $T_{min} \leq T_{trasm} \leq T_{max}$
    - Let $u = (T_{max} - T_{min})$ be the transmission time uncertainty
  - Clock setting rule: if p_2 sets its clock to $t + (T_{max} + T_{min})/2$, the optimal lower bound on the clock skew is $u/2$
- Generalization: the algorithm can be extended to N processes
  - The optimal lower bound on the clock skew is $u(1 - 1/N)$
- Asynchronous DS
  - Message delays are unknown and unbounded: $T_{trasm} = T_{min} + x$, with $x \geq 0$ and unknown
  - Different physical clock synchronization algorithms are required

Physical synchronization via a time service

- A time service can provide the current time with a given precision
  - Equipped with a UTC receiver or a high-accuracy clock
- The group of processes that needs to synchronize uses a time service
  - Time service types: centralized or distributed
- Centralized time service
  - Request-driven: **Cristian's algorithm** (1989)
    - External synchronization
  - Broadcast-based: **Berkeley Unix algorithm** - Gusella & Zatti (1989)
    - Internal synchronization
- Distributed time service
  - **Network Time Protocol** (NTP)
    - External synchronization
    - Internet-scale

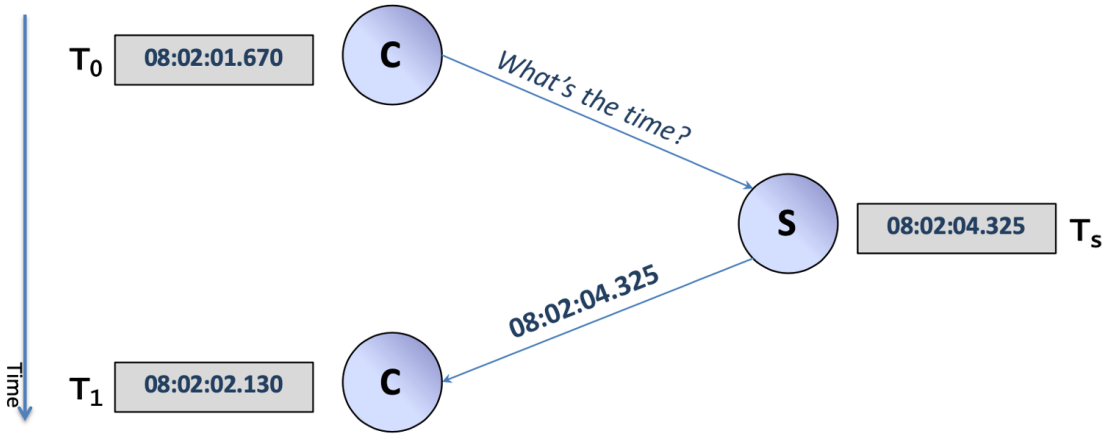
Cristian's algorithm

- A (passive) time server *S* receives the signal from a UTC source → external synchronization
- A process *p* requests the time by sending a request m_r and receives the time *t* in the reply m_t from *S*
- Process *p* sets its clock to $t + T_{round}/2$
  - T_{round} is the round-trip time measured by *p*, assume symmetric network delay



- Observations
  - A single time server may fail
    - Solution: use a group of synchronized time servers
  - Malicious time servers are not handled
  - Reasonable accuracy only if T_{round} is small → suitable for low-latency LAN

Cristian's algorithm: example



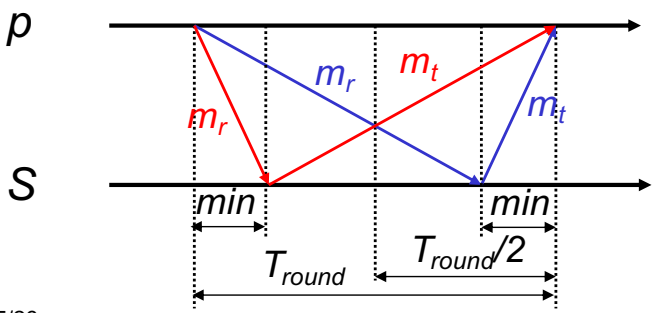
- $T_{round} = T_1 - T_0 = 460$ ms
- C sets its clock to $T_s + T_{round}/2$ = 08:02:04.325 + 230 ms = 08:02:04.555
- C clock increases by 2.425 s

Cristian's algorithm: accuracy

Case 1: *S* cannot insert *t* into m_t before *min* time units have elapsed since *p* sent m_r
  - *min* is the minimum transmission time between *p* and *S*

Case 2: *S* cannot insert *t* into m_t after the instant when m_t arrives at *p* minus *min*

- The time at *S* when m_t arrives at *p* lies in the interval $[t + min, t + T_{round} - min]$
  - Interval width: $T_{round} - 2\ min$
- Accuracy α: $\pm(T_{round}/2 - min)$



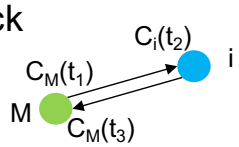
Berkeley algorithm

- Internal synchronization algorithm for a group of nodes
- The *master* node (*active* time server) broadcasts a request to all nodes, including itself, to get the clock values of the other nodes (*workers*)
- The master uses the RTTs to estimate the clocks of the workers
  - δ_i: difference between *M* clock and worker *i* clock (calculated similarly to Cristian's algorithm)

    $\delta_i = (C_M(t_1) + C_M(t_3))/2 - C_i(t_2)$

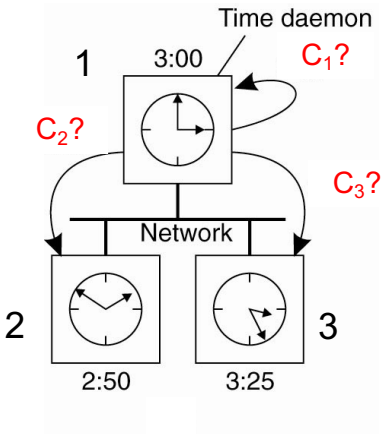
    $C_M(t_1)$ and $C_M(t_3)$: master's clock at sending and receiving

    $C_i(t_2)$: worker *i* clock when reply is sent

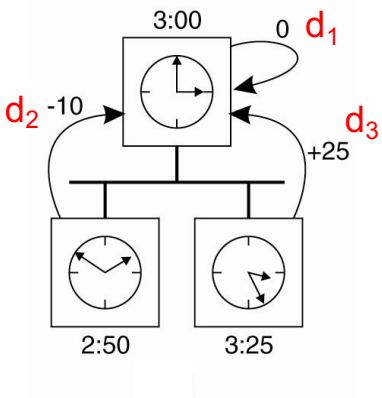


- Master calculates the average of of all δ_i (including its own) and sends correction value to workers
  - If backward adjustment needed: slow down clock

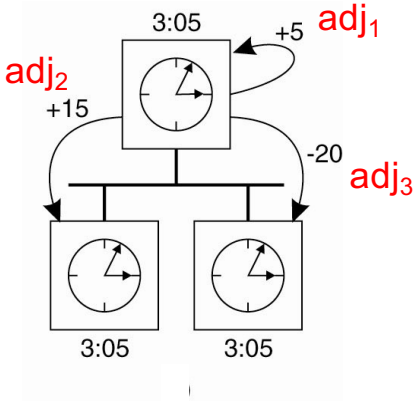
Berkeley algorithm: example



Master broadcasts a request to all nodes



Master calculates the average of clock differences:
avg = (-10+25+0)/3 = 5



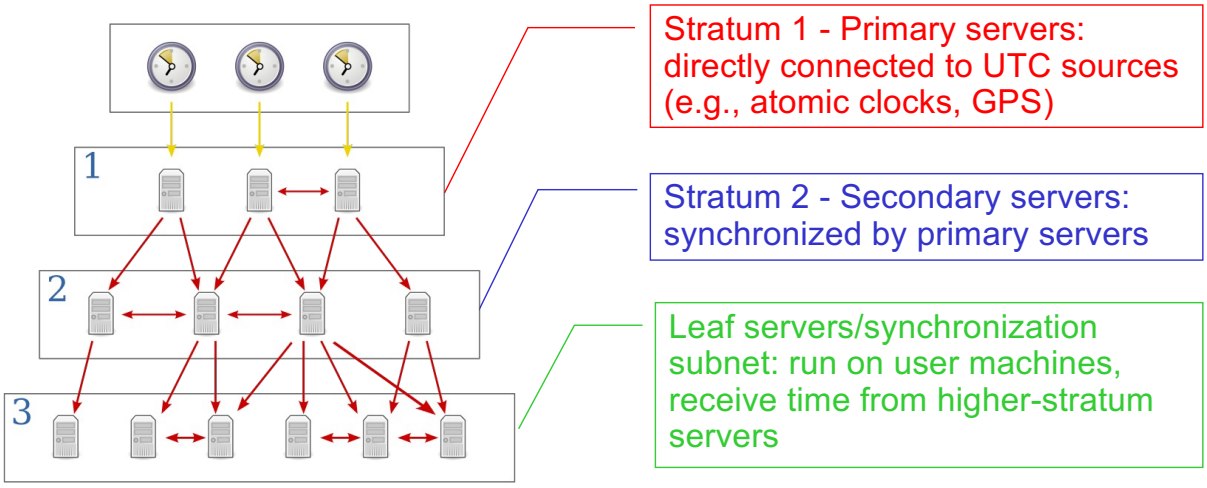
Master sends a correction value to each worker (adj_i = avg - d_i)

Berkeley algorithm: features

- Precision: depends on the maximum nominal RTT
  - The master ignores clock values associated with RTTs above this maximum
  - Outliers are discarded to improve accuracy
- Fault tolerance
  - Master failure: another node is elected as master using an election algorithm
  - Tolerance to arbitrary (Byzantine) behavior:
    - Workers sending incorrect clock values
    - The master only considers clock values that differ from each other by at most a specified threshold

Network Time Protocol (NTP)

- Time service for Internet (RFC 5905)
  - Provides external synchronization accurate wrt UTC
  - Configurable on multiple OSs
  - On GNU/Linux: `ntpd` daemon for automatic synchronization, `ntpdate` for manual synchronization
- Hierarchy of servers



NTP

- Architecture
  - Scalable and robust time service
  - Redundant time servers and network paths
  - Authentication of time sources to prevent

- NTP fault tolerance and reconfiguration
  - The synchronization subnet reconfigures automatically in case of failures
  - Primary server loses connection to UTC source: it becomes a secondary server
  - Secondary server loses connection to its primary (e.g., primary crashes): it can switch to another primary server

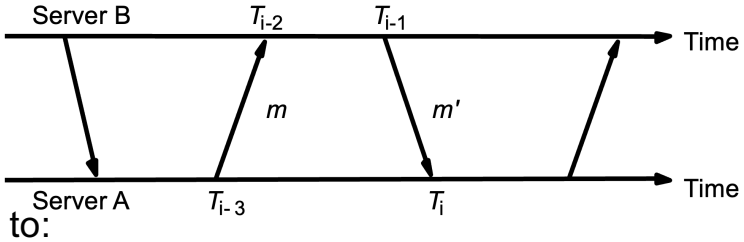
NTP: synchronization modes

1. Multicast mode
   - Server in a high-speed LAN multicasts its clock to others
   - Other servers set their clocks assuming a certain transmission delay
   - Accuracy: relatively low (no way to measure RTT)
2. Procedure call mode
   - Server responds to requests like in Cristian's algorithm
   - Accuracy: higher than multicast
   - Use case: when multicast is not available, makes sense for smaller networks
3. Symmetric mode
   - Server pairs exchange timing information
   - Accuracy: very high
   - Commonly used for higher-level hierarchical servers (Stratum 2 and above)

- All synchronization modes use UDP

NTP: symmetric mode

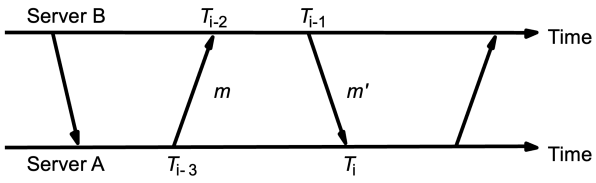
- Servers A and B exchange message pairs (*m*, *m'*) to improve synchronization accuracy
- Servers exchange messages with timestamps
  - A sends to B a message *m, which* contains:
    - Timestamp T_{i-3}: time when A sends *m*
  - B responds to A with message *m*', which contains:
    - Timestamp T_{i-2}: time when B receives *m*
    - Timestamp T_{i-1}: time when B sends *m'*
  - A records T_i when it receives *m'*
  - Time difference T_{i-1}-T_{i-2} may be non-negligible



- Timestamps allow servers to:
  - Estimate round-trip delay
  - Adjust their clocks more precisely

NTP: symmetric mode

- For each pair of messages *m* and *m'* exchanged between two servers, NTP estimates:
  - Offset o_i between the two clocks
  - Round-trip delay d_i, which is the total transmission time of *m* and *m'*



- Let's define:
  - *o*: real offset of clock B relative to clock A ($o = clock_B - clock_A$)
  - *t* and *t'*: actual transmission times of messages m and m', respectively
- For each pair of messages:

$$T_{i-2} = T_{i-3} + t + o \quad \text{and} \quad T_i = T_{i-1} + t' - o$$

$$\text{where } d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

- Subtracting the equation for T_i from the one for T_{i-2}, we solve for *o*:

$$o = [(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)]/2 + (t' - t)/2$$

Difference between the transmission times used to adjust the offset estimate

The estimated offset o_i is given by: $o_i = [(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)]/2$

The real offset *o* is: $o = o_i + (t' - t)/2$

NTP: symmetric mode

- The accuracy of the offset estimation is determined by the round-trip delay d_i
  - Since both t and t' are positive, the real offset o will lie within: $o_i - d_i/2 \leq o \leq o_i + d_i/2$
  - Thus, o_i is the estimated offset and its *accuracy* is $d_i/2$
- NTP servers apply a statistical filtering algorithm on the 8 most recent $<o_i, d_i>$ pairs, choosing as the estimate of o the o_j value corresponding to the minimum d_j
- They then apply a peer selection algorithm to possibly change the peer used for synchronization
  https://www.eecis.udel.edu/~mills/ntp/html/warp.html
- NTP accuracy:
  - 10 ms on the Internet
  - 1 ms on LAN

Perfect synchronization over networks is actually impossible

Google's TrueTime (TT)

- Distributed synchronized clock with bounded non-zero error
  - Designed by Google for Spanner, a global-scale, multi-version, distributed NewSQL database
  - Relies on a well-engineered tight clock synchronization available on all Google servers thanks to GPS and atomic clocks
  - Enables applications to generate monotonically increasing timestamps
  - *Cons*: TT requires special hardware and custom-build tight clock synchronization protocol, which is infeasible for many systems
    - Google also relies on its very high throughput, global fiber optic network linking its data centers

Time in asynchronous DS

- Physical clock synchronization algorithms
  - They (e.g., NTP) estimate transmission times to synchronize clocks between nodes
  - Accuracy depends on knowing the upper and lower bounds of transmission times
- Limitations in asynchronous DS
  - No constraints on transmission times → physical synchronization is used in asynchronous DS but with limited accuracy due to variable transmission times
  - We cannot use physical time to order events occurring on different nodes
- Logical time as a solution
  - What really matters is that processes agree on the order of events rather than the exact physical time at which events occurred
  - We use logical clocks (e.g., Lamport clocks) to maintain event ordering across nodes

Logical time

- Idea: order events based on two intuitive observations:
  1. Two events that occurred on the same process p_i happen exactly in the order observed by p_i
  2. When a message is sent from p_i to p_j, the send event happens before the receive event
- Lamport (1978) and the **happened-before** relation
  - Leslie Lamport introduces the concept of the happened-before relation (also known as *causal ordering* or *precedence relation*)

$\rightarrow_i$: ordering relation between two events that occurred on p_i

$\rightarrow$: happened-before relation between any two events

L. Lamport, Time, Clocks and the Ordering of Events in Distributed Systems, 1978 https://lamport.azurewebsites.net/pubs/time-clocks.pdf

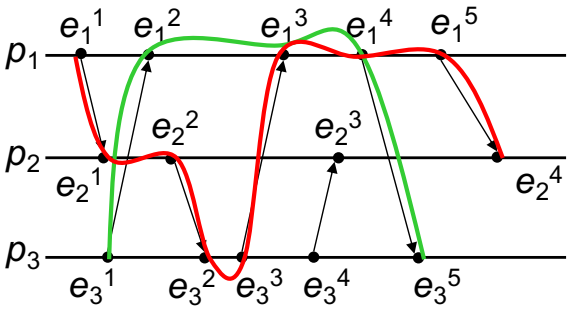
Happened-before relation

- Two events *e* and *e'* are in the happened-before relation (denoted by $e \rightarrow e'$) if one of the following is true:
  1. $\exists\ p_i \mid e \rightarrow_i e'$
  2. $e = send(m) \wedge e' = receive(m)$

     e is the event of sending message *m*, and *e'* is the corresponding receive event
  3. $\exists\ e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$

     The happened-before relation is transitive
- By applying these rules, we can construct a sequence of events $e_1, e_2, \ldots, e_n$ that are causally ordered
- Observations
  - The happened-before relation defines a partial ordering (properties: *non-reflexive, anti-symmetric*, *transitive*)
  - The sequence $e_1, e_2, \ldots, e_n$ is not necessarily unique
  - For a pair of events, they are not always related by a happened-before relation. In such cases, the events are said to be concurrent (denoted by ||)

Happened-before relation: example



- Sequence $s_1 = e_1^1, e_2^1, e_2^2, e_3^2, e_3^3, e_1^3, e_1^4, e_1^5, e_2^4$
- Sequence $s_2 = e_3^1, e_1^2, e_1^3, e_1^4, e_3^5$
- Can you find another sequence?

- Events e_3^1 and e_2^1 are concurrent

  $e_3^1 \not\rightarrow e_2^1$ and $e_2^1 \not\rightarrow e_3^1$

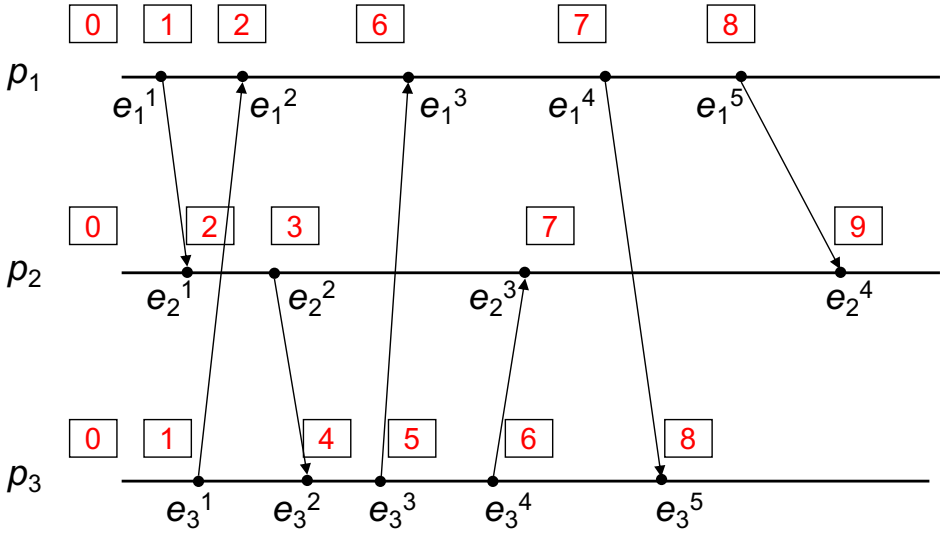
Scalar logical clock

- A monotonically increasing software counter (scalar value), used to order events in DS
  - Scalar logical clock (or Lamport clock): we call it "scalar" to distinguish from vector clock
- Does not rely on physical clock
- Each process p_i has its own scalar clock L_i, and it uses it to apply timestamps to events
- Denote $L_i(e)$ as the timestamp, based on the scalar clock, applied by process p_i to event e
- Key property: **if $e \rightarrow e'$ then $L(e) < L(e')$**
- Key observations:
  - If $L(e) < L(e')$, it does not necessarily mean that $e \rightarrow e'$; however, $L(e) < L(e')$ implies that $e \rightarrow e'$
  - The happened-before relation introduces a partial ordering of events: in the case of concurrent events, it is impossible to determine which event actually happens first

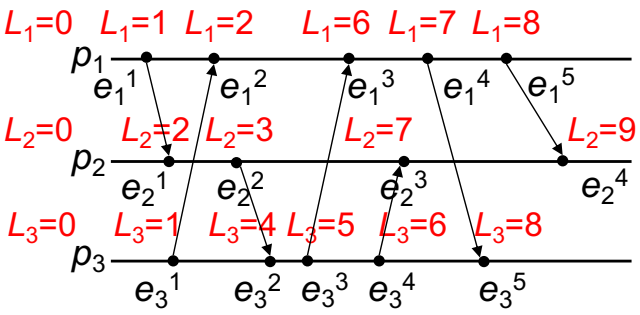
Scalar clock: update

- Scalar clock update algorithm by Lamport
- Initialization: each process p_i initializes its scalar clock L_i to 0 ($\forall\ i = 1, \ldots, N$)
- Internal event: before executing an internal event, p_i increments L_i by 1: $L_i = L_i + 1$
- Sending a message: when p_i sends message m to p_j
  - Increment the value of L_i: $L_i = L_i + 1$
  - Attach the timestamp $t = L_i$ to message m
  - Execute the *send*(m) event
- Receiving a message: when p_j receive message m with timestamp t
  - Update its scalar clock $L_j = max(t, L_j)$
  - Increment the value of L_j: $L_j = L_j + 1$
  - Execute the *receive*(m) event

Scalar clock: example



Scalar clock: example

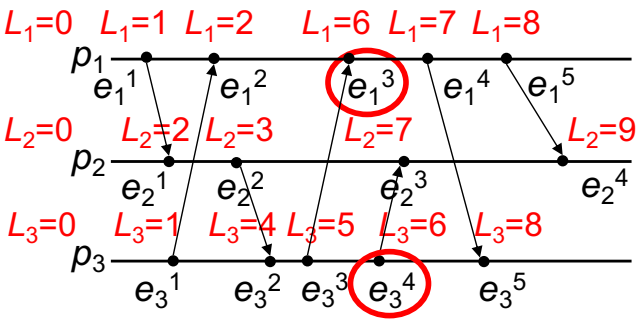


- Observations
  - $e_1^1 \rightarrow e_2^1$ and the respective timestamps reflect this causal relationship (L_1=1 and L_2=2); indeed, *if* $e_1^1 \rightarrow e_2^1$ *then* $L(e_1^1) < L(e_2^1)$
  - $e_1^1 \parallel e_3^1$ and the respective timestamps are equal (L_1=1 and L_3=1); indeed, *if* $L(e_1^1) \geq L(e_3^1)$ then $e_1^1 \not\rightarrow e_3^1$
  - $e_2^1 \parallel e_3^1$ and the respective timestamps differ (L_2=2 and L_3=1); indeed, *if* $L(e_2^1) \geq L(e_3^1)$ then $e_2^1 \not\rightarrow e_3^1$

Total ordering of events

- Problem: using a scalar clock, two or more events can have the same timestamp. How can we establish a total ordering of events to avoid two events occurring at the same logical time?
- Solution: in addition to the scalar clock, we use the process number on which the event occurred
  - We establish a total ordering ($\prec$) between the processes
- Total order relation between events, denoted by $e \Rightarrow e'$: if e is an event on process p_i and e' is an event on process p_j, then $e \Rightarrow e'$ if and only if:
  1. $L_i(e) < L_j(e') \vee$
  2. $L_i(e) = L_j(e') \wedge p_i \prec p_j$
- This relation is used in some distributed mutual exclusion algorithms to ensure a total ordering of events across processes

Total ordering of events: example



- e_1^3 and e_3^4 have the same scalar clock value: how to order them?
- $e_1^3 \Rightarrow e_3^4$ since $L_1(e_1^3) = L_3(e_3^4)$ and $p_1 \prec p_3$

Scalar clock and its limitation

- The scalar clock has the following property
  - If $e \rightarrow e'$, then $L(e) < L(e')$
- However, it is not possible to guarantee that:
  - If $L(e) < L(e')$, then $e \rightarrow e'$
    See slide 36: $L(e_3^1) < L(e_2^1)$ but $e_3^1 \parallel e_2^1$
- Consequence: it is not possible to determine, by just looking at scalar clocks, whether two events are concurrent or not
- How to overcome this limitation?
- To solve this issue, vector clocks were introduced by Mattern (1989) and Fidge (1991)

Vector clock

- A vector clock for a system with N processes is a vector of N integers
- Each process p_i maintains its own vector clock V_i
- For process p_i, $V_i[\, i \,]$ is the local scalar clock
- Each process uses its own vector clock to assign timestamps to events
- Similar to Lamport's clock, the vector clock is attached to the message m, and the timestamp becomes vector-valued
- The vector clock captures the full characteristics of the happened-before relation

$$e \rightarrow e' \text{ if and only if } V(e) < V(e')$$

Vector clock: meaning and comparison

- Given the vector clock V_i
  - $V_i[\,i\,]$ is the number of events generated by p_i
  - $V_i[\,j\,]$ with $i \neq j$ is the number of events that occurred at p_j, which p_i knows about

- Comparison of vector clocks
  - $V = V'$ if and only if $\forall\, j$: $V[\,j\,] = V'[\,j\,]$
  - $V \leq V'$ if and only if $\forall\, j$: $V[\,j\,] \leq V'[\,j\,]$
  - $V < V'$ (and thus the event associated with V *precedes* the event associated with V') if and only if
    - $\forall\, i \in [1, \ldots, N]$: $V[\,i\,] \leq V'[\,i\,]$

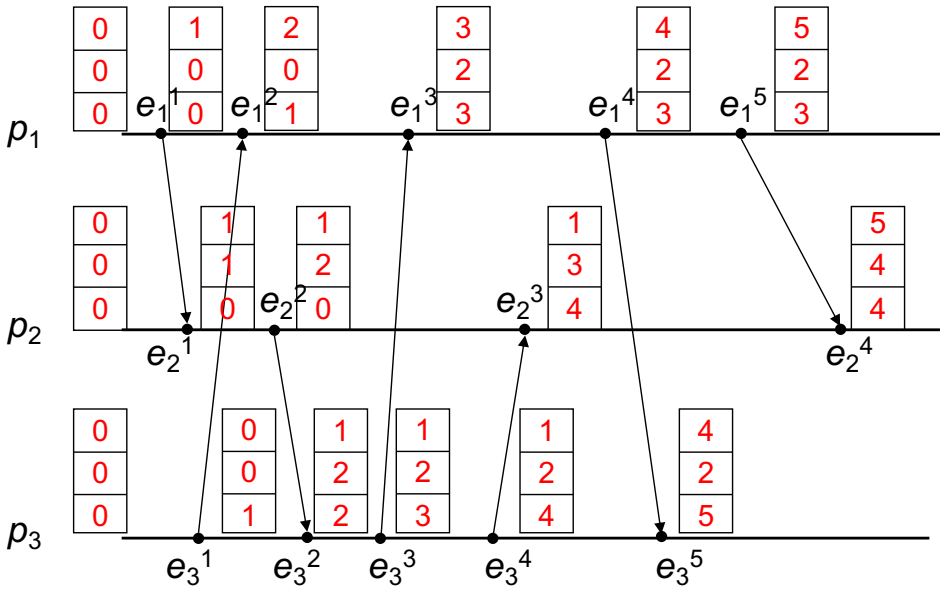
      and
    - $\exists\, j \in$ in $[1, \ldots, N]$: $V[\,j\,] < V'[\,j\,]$
  - $V \,||\, V'$ (and thus the event associated with V *is concurrent* to the event associated with V') se e solo se
    - $\neg(V < V')$ and $\neg(V' < V)$

Vector clock: update

- Initialization: each process p_i initializes its own vector clock V_i: $V_i[k]=0 \;\; \forall\, k = 1, 2, \ldots, N$
- Internal event: before executing an internal event, p_i increments its own vector clock component $V_i[i]$ by 1: $V_i[i] = V_i[i] + 1$
- Sending a message: when p_i sends message m to p_j
  - Increment its own clock component $V_i[i]$ by 1: $V_i[i] = V_i[i] + 1$
  - Attach the vector timestamp $t = V_i$ to message m
  - Execute the *send(m)* event
- Receiving a message: when p_j receives message m with timestamp t:
  - Update its vector clock component for each process K:

    $V_j[k] = max(t[k], V_j[k]) \;\forall\, k = 1, 2, \ldots, N$
  - Increment its own clock component $V_j[j]$ by 1: $V_j[j] = V_j[j] + 1$
  - Execute the *receive*(m) event

Vector clock: example



Using vector clocks to compare events

- Let $V(e)$ and $V(e')$ be the vector timestamps of two events e and e'
- By comparing $V(e)$ and $V(e')$, we can determine if the events are in happened-before relationship or concurrent

V	V'
1	1
2	2
0	2

$V(e) < V'(e)$ and thus $e \rightarrow e'$

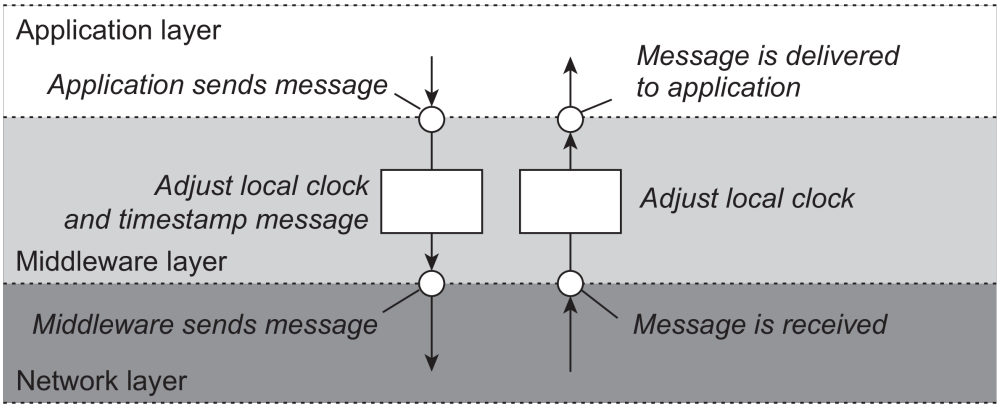
V	V'
1	1
2	0
0	2

$V(e) \neq V'(e)$ and thus $e \parallel e'$

Examples of applications of logical time

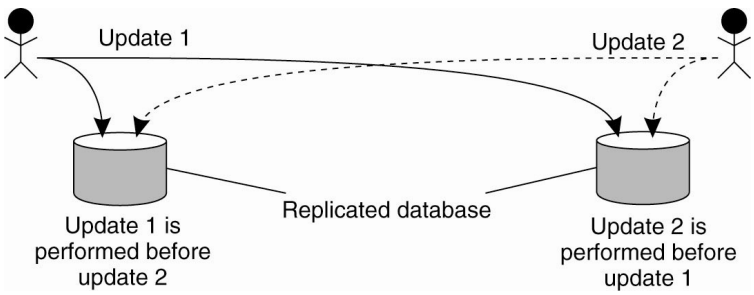
- We will examine two applications of logical time:

1. Scalar clock for totally ordered multicasting
2. Vector clock for causally ordered multicasting



Totally ordered multicasting: problem

- To ensure that concurrent updates on a replicated database are seen in the same order by every replica, you need to synchronize and order the updates across all replicas, ensuring consistency
  - p_1: add \$100 to a bank account (initial value: \$1000)
  - p_2: increment the account by 1%
  - There are two replicas and, without synchronization, each replica may apply the updates in a different order
    - Replica #1 ← 1111 (first p_1 and then p_2)
    - Replica #2 ← 1110 (first p_2 and then p_1)



Totally ordered multicasting

- A totally ordered multicast is a multicast operation in which all messages are delivered in the same order to every recipient
- Assumptions:
  - Reliable communication: no message loss occurs
  - FIFO ordered communication: messages sent from p_i to p_j are received by p_j in the same order in which p_i sent them p_j

- Let's consider two solutions to implement totally ordered multicasting
  - Centralized
  - Distributed, using scalar clocks

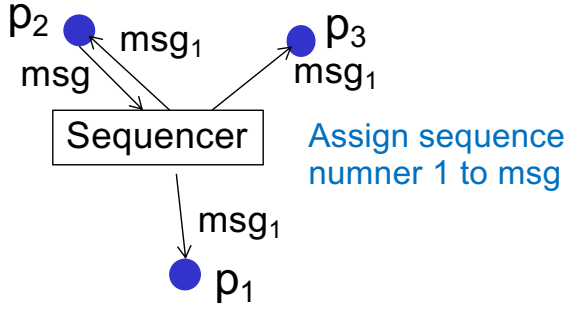
Totally ordered multicasting: solutions

- Centralized solution: a single coordinator (*sequencer*) manages the order of messages)
- The sequencer ensures that messages are delivered to all recipients in the same order
  - Each process p_i sends its update message to the sequencer
  - The sequencer then assigns a unique sequence number to each update message and multicasts the message
  - Each process p_i executes the update in the order of the sequence numbers

✓ Simple to implement

✗ Single point of failure

✗ Potential bottleneck at the sequencer



Totally ordered multicasting: solutions

- Distributed algorithm using scalar clocks
  - Tag message with scalar clock: each update message msg_i sent by process p_i is tagged with p_i's scalar clock at the time of sending
  - Multicast the message: p_i multicasts to all processes (including itself)
  - Queue messages locally: each receiving process p_j places msg_i in a local queue $queue_j$, sorted by timestamp (scalar clock value)
  - Acknowledge receipt: p_j multicasts an ack for msg_i to all processes, indicating it has received the message
  - Deliver message to application: p_j delivers msg_i to the application only if all of the following conditions are met:
    1. msg_i is at the head of $queue_j$ (and all acks for msg_i have been received by p_j)
    2. for every other process p_k, there is a message msg_k in $queue_j$ with a timestamp greater than msg_i

    msg_i is delivered only when p_j knows that no other process can multicast a message with a smaller or equal timestamp than msg_i: this ensures that all processes deliver messages in the same total order

Totally ordered multicasting: distributed

✓ Decentralized: no single point of failure and works better than centralized one with larger systems

✗ Slightly more complex than centralized solution

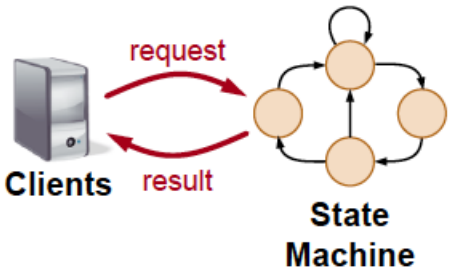
✗ Communication cost:

- Ack overhead: $O(N^2)$ acks for each message
  - Each process sends N-1 acks for each message it receives, and this happens for every process in the system

- Mechanism designed by Lamport for **state-machine replication**

State machine replication

- Software replication technique applied to a service that can be implemented as a deterministic state machine
  - Used to replicate the state of a machine ("state" can be any data or process state) across multiple replicas
  - A state machine consists of a series of states and transitions between those states
  - Each operation in the system causes a transition between states: next state of service = f(current state, op)
  - Deterministic state machine: for every given state and operation, there is exactly one defined next state
- Each replica processes the same sequence of operations in the same order, ensuring that all replicas stay in sync (i.e., end up in the same state)



State machine replication

- To achieve fault tolerance, service is replicated on several nodes, all of them running a state machine replication (SMR) middleware
  - Set of replicas behaves as a "centralized" server
- Service makes progress as long as any majority of replicas are up
- Replicas can recover by reapplying the operations in the same order
- Used in systems that require strong consistency and fault tolerance, such as distributed databases, file systems, and distributed transaction systems

State machine replication

- Lamport's totally ordered multicast ensures consistency, but it may face scalability challenges
  - Particularly in systems with high latency or frequent failures
- Other well-known solutions for ensuring state-machine replication we examine
  - Paxos consensus algorithm
  - Raft consensus algorithm

Causally ordered multicasting

- Ensures that a message is delivered only after all the messages that causally precede it (i.e., events that are causally related) have already been delivered
  - Causal relation between two events: the second event is *potentially* influenced by the first event, meaning there is a cause-effect relationship
  - Goal: deliver the cause before the effect to maintain the causal order of events
  - Weaker than totally ordered multicasting
    - We do not require all messages to be delivered in a global total order. Instead, we only require that causal relationships are respected
  - Example:
    - p_1 sends messages m_A and m_B
    - p_2 sends messages m_C and m_D
    - m_A causes m_C
    - Delivery sequences that are compatible with causal ordering (and FIFO ordering): $m_A\, m_B\, m_C\, m_D$ $m_A\, m_C\, m_B\, m_D$ $m_A\, m_C\, m_D\, m_B$ but $m_C\, m_A\, m_B\, m_D$ is not valid

Causally ordered multicasting

- Assumptions: reliable and FIFO ordered communication
- Use vector clocks to solve the problem of causally ordered multicasting in a decentralized way
- Algorithm
  - Sending the message: process p_i sends the message m to all other processes, attaching a vector timestamp (ts_m). The timestamp is p_i's vector clock (V_i) at the time of sending the message
  - Receiving the message: *w*hen p_j receives m from p_i, p_j does not immediately deliver the message to the application. Instead, p_j places the message in a waiting queue
    - $V_j[i]$ counts the number of messages sent by p_i to p_j
  - Conditions for delivering the message: m will be delivered to the application only if both of the following conditions are met:
    1. $ts_m[i] = V_j[i] + 1$ p_j has received the next message it expects from p_i
    2. $s_m[k] \leq V_j[k]\ \forall\ k \neq i$ p_j has seen at least all the messages that p_i has seen up to that point

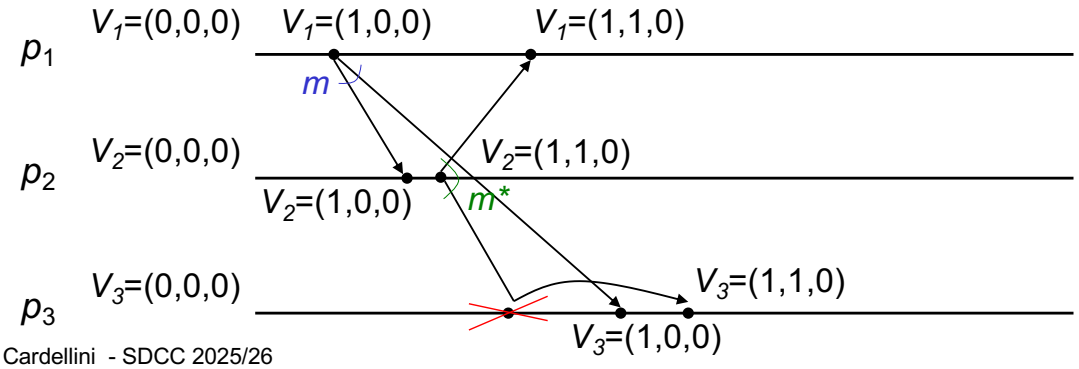
Causally ordered multicasting: example

- p_1 sends m to p_2 and p_3
- p_2 *receives* m and then sends m^* *to* p_1 and p_3
  - Cause-effect relationship between m and m^*
- Suppose that p_3 receives m^* before it receives m
  - The algorithm avoids the violation of causal ordering: p_3 will hold m^* in a waiting queue until m is delivered first

Clock update

p_i sends msg: $V_i[i] = V_i[i] + 1$

p_j receives msg with ts_{msg}: $V_j[k] = \max\{ V_j[k], ts_{msg}[k] \}$



Timestamps in practice

- Timestamps are useful for comparing events in DS, e.g.
  - Reconciling object updates in distributed storage system
  - Restoring system state after crash

    1. Checkpoint the system: a checkpoint is created, capturing the system's state at a specific point in time; 2 Record events: each event is recorded with a timestamp to track the order in which events occurred; 3 After a crash: the checkpoint is restored, and the system replays the events in order using the timestamps
- How to compare timestamps across different processes?

1. Physical timestamps: rely on synchronization of physical clocks across processes
   - Use case: Google Spanner uses TrueTime to maintain consistency
   - Limitation: require all clocks to be synchronized, which can be difficult in geographically distributed systems with high latencies

Timestamps in practice

2. Logical timestamps: use scalar clocks to order events
   - Use case: Oracle DB use systems change numbers based on scalar clocks to track changes to data, ensuring that operations are applied in the correct order
   - Limitation: scalar clocks do not fully distinguish between causally related events and concurrent events
3. Vector timestamps: extend logical timestamps by using vector clocks
   - Use case: Amazon DynamoDB uses vector clocks to determine which object version is the most recent by comparing vector timestamps

References

- Sections 5.1 and 5.2 of van Steen & Tanenbaum book
- Sections 14.1 - 14.4 of Coulouris et al. book
- Lamport, Time, clocks and the ordering of events in a distributed system, Comm. ACM, 1978 https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system
- Raynal and Singhal, Logical time: Capturing causality in distributed systems, IEEE Computer, 1996 https://doi.org/10.1109/2.485846