Time and Ordering

Week 7, Lecture 2

Onur Ascigil

Agenda

- **How do systems keep accurate time?**
- How to order distributed events?

Importance of time

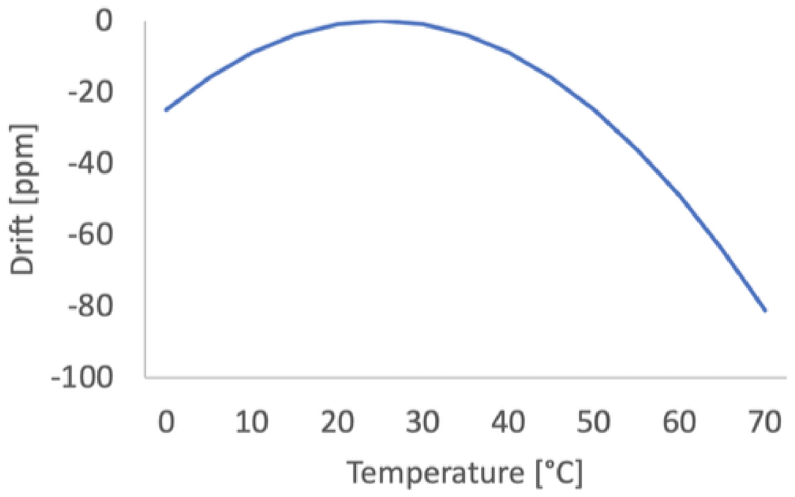
- Computer systems often need to measure time:
  - Schedulers, timeouts, failure detectors, retry timers
  - Logging events along with when each occurred for debugging
  - Data with time-limited validity (e.g., cache entries)
- Two types of clocks:
  - **physical clocks**: count number of seconds elapsed
  - **logical clocks**: count events, e.g. messages sent

Physical clocks (1)

- **1. Quartz clocks:**
  - Found in every computer, mobile phone, wristwatch, microwave owen, etc.
- Uses an electronic oscillator regulated by a **quartz crystal**
  - A precisely-cut piece of synthetic quartz crystal
  - It vibrates at a specific frequency when an electric current is applied to it.
- An oscillator circuit produces signal at a certain frequency
- The oscillations of the quartz crystal are *divided down* to a lower frequency that can be used to drive the clock's timekeeping mechanism.
- **Advantages**: cheap and does not consume much power
- **Disadvantage**: not very accurate – *clock drifts*!
  - Due to manufacturing imperfections
  - Vibration frequency changes with temperature

Quartz clock error: drift

- One clock runs slightly fast, another slightly slow
- Drift measured in parts per million (ppm)
- 1 ppm = 1 microsecond/second = 86 ms/day = ~31 seconds/year



Physical Clocks (2)

- **2. Atomic clock**: a very precise timekeeping device (drifts ~1 second in thousands of years)
  - Based on microwave resonance of certain atoms which are very stable
  - Tunes the frequency of an electronic oscillator to the resonance frequency
  - But very expensive and bulky
  - Network Time Protocol (NTP) servers have atomic clocks
  - GPS Satellites carry atomic clocks

Physical Clocks (3)

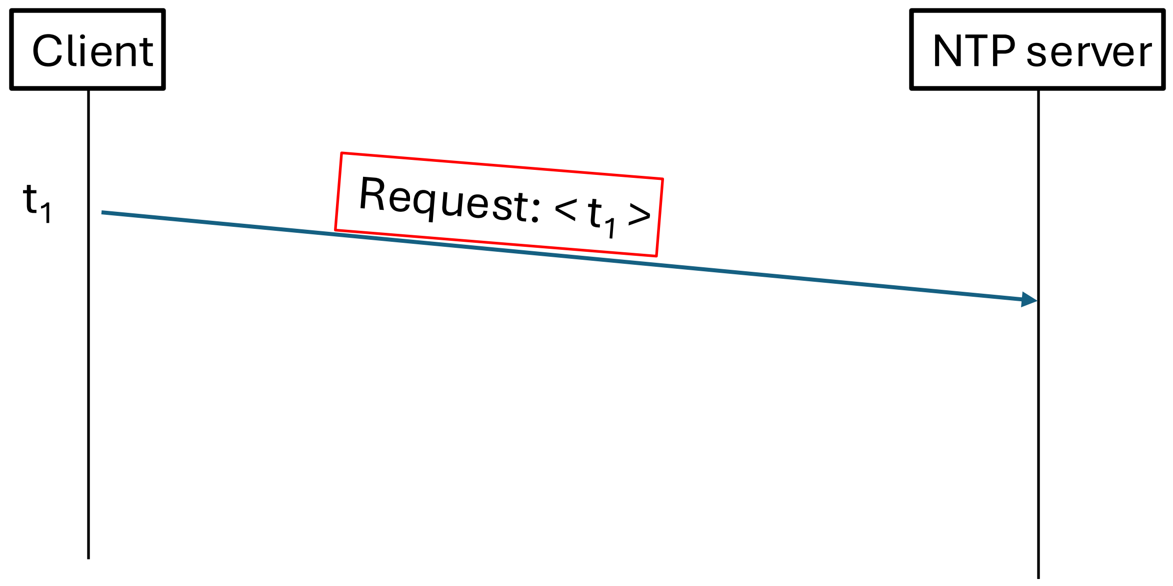
- It is not cost-effective or feasible to have an atomic clock on every device
- **Key Idea:** Use devices that maintain accurate time (using their expensive clocks) as a "time source"
  - Other devices can periodically ask them for the current time
- Time sources:
  - **Network Time Protocol (NTP) Service:** many ISPs operate NTP servers carrying atomic clocks
  - **GPS:** 31 satellites each carrying an atomic clock

Clock Synchronisation (1)

- Computers track physical time with a quartz clock (with battery, continues running when power is off)
- Due to clock drift, clock error gradually increases
  - **Clock skew**: difference between two clocks at a point in time
    - **Clock Skew = Local Clock Time − Reference Clock Time**
- **Idea**: Periodically get the current time from a server (time source) that keeps accurate time
- How do we calculate the skew given that there are delays in communication?
  - By the time, the timestamp from the server arrives at the client, it no longer reflects the current time – so it is not a valid reference to compare against!
  - We need snapshots of both client's local clock and server's clock taken at the same time!

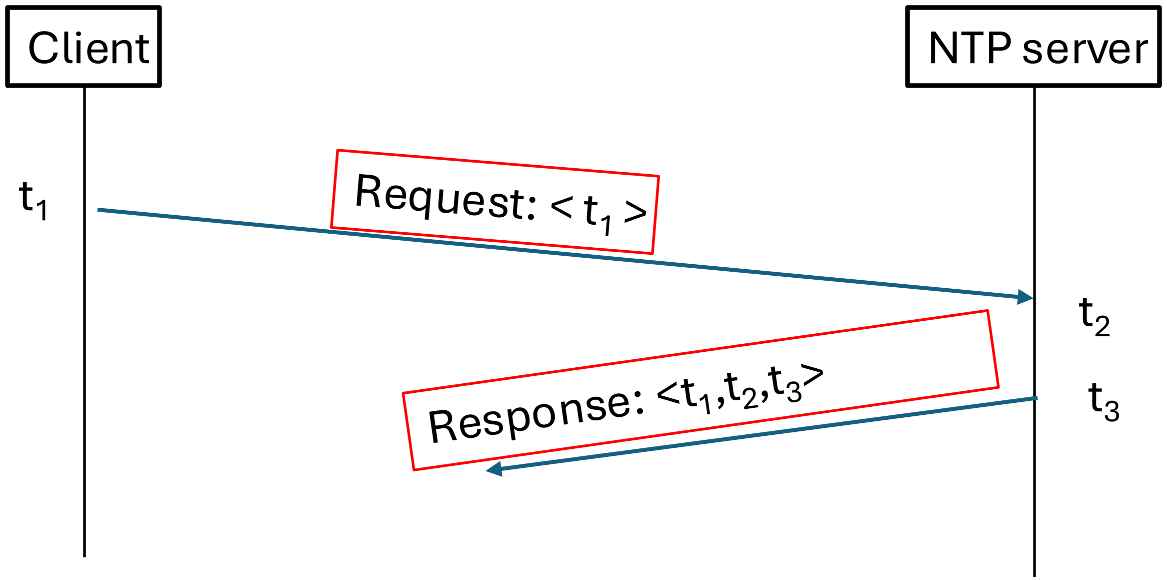
Clock Synchronisation(2) – NTP

- Step 1: The client device sends a request containing the time of sending request (t_1) based on its local (possibly inaccurate) clock



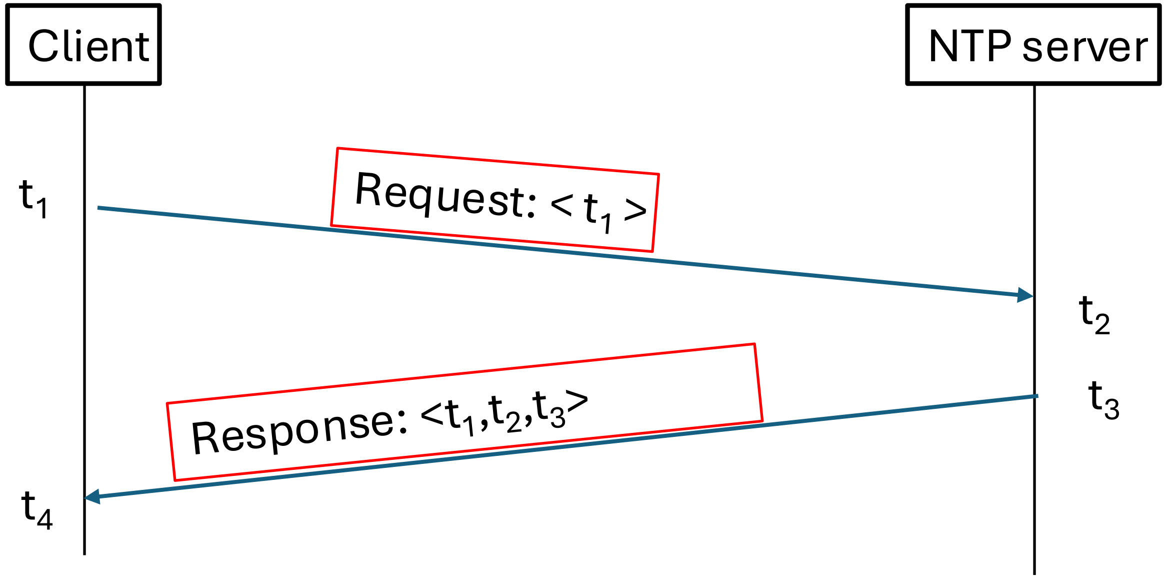
Clock Synchronisation(3) – NTP

- Step 2: NTP server receives the request at time t_2 and sends a response at time t_3 (both timestamps based on NTP server's accurate clock)



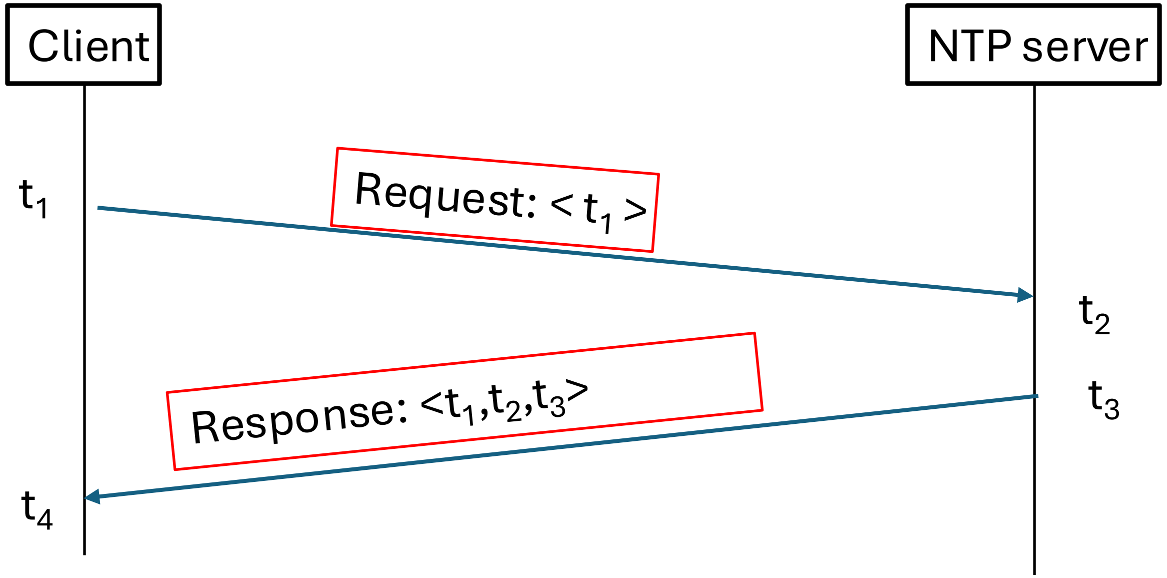
Clock Synchronisation(4) – NTP

- Step 3: The client receives the response at time t_4 based on its own local (possibly inaccurate) clock
- **Question 1:** how can the client estimate the clock skew?



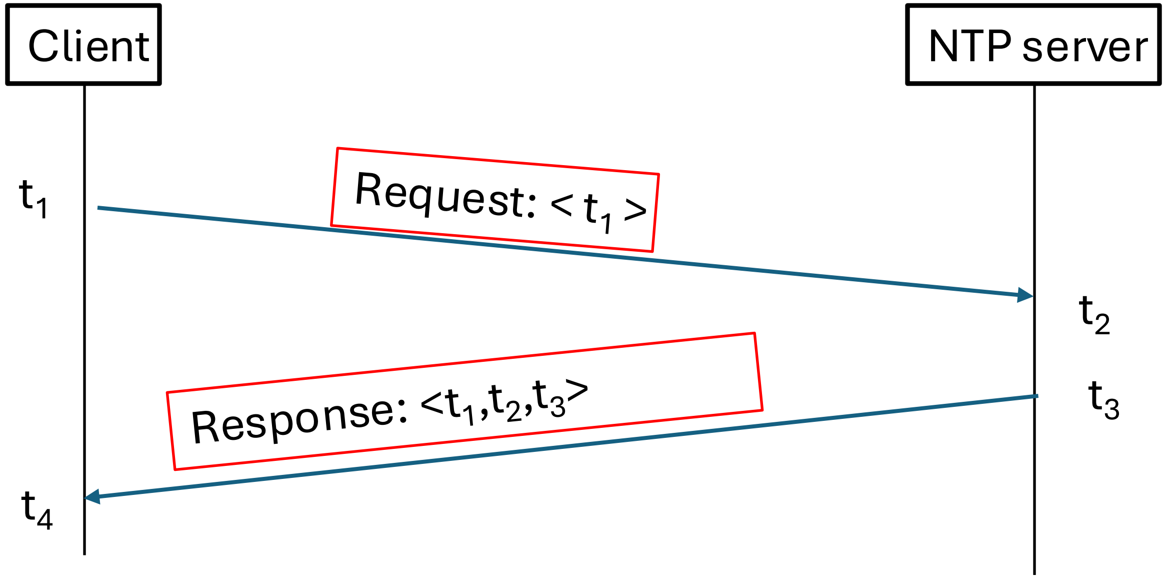
Clock Synchronisation(5) – NTP

- **Question 1:** how can the client estimate the clock skew?
- **Approach:**
  - 1. Calculate the one-way network delay: $t_{1\text{-way}}$, i.e., delay between client and server
  - 2. Then calculate clock skew = $<t_3 - t_4 + t_{1\text{-way-response}}>$ or $<t_2 - t_1 - t_{1\text{-way-request}}>$

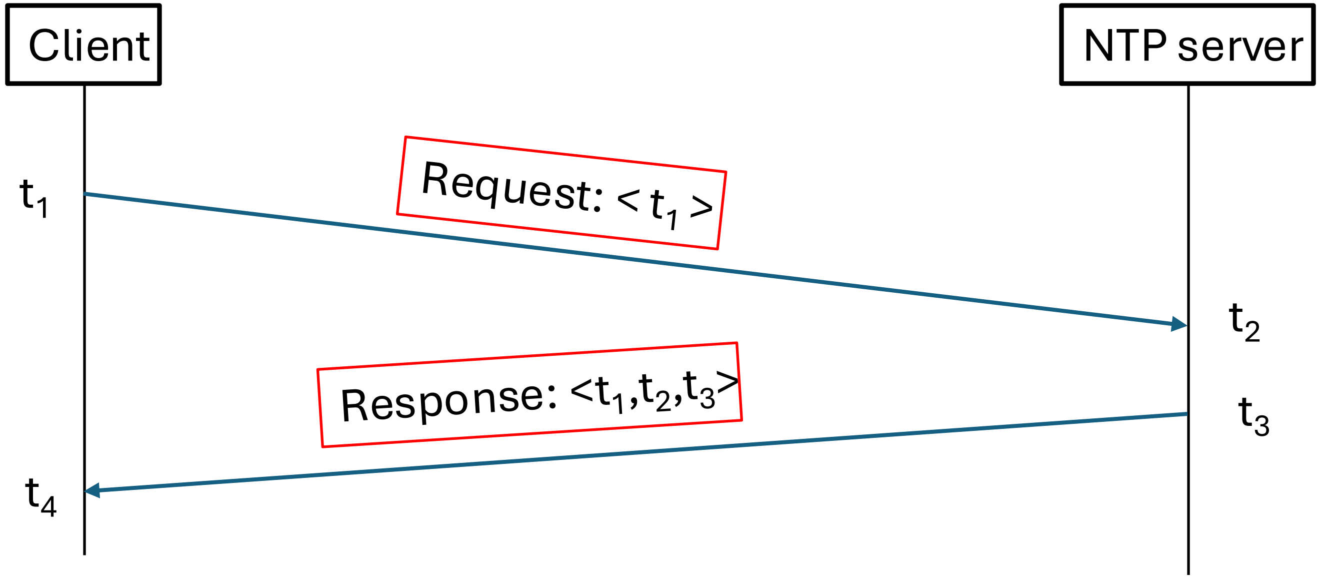


Clock Synchronisation(6) – NTP

- Is it possible for the client to calculate one-way delays: <$t_{\text{1-way-response}}$> and <$t_{\text{1-way-response}}$> ?
  - No, it can only calculate the RTT
- RTT (only communication delays) = *Total RTT – Processing delay*
  - $= (t_4 - t_1) - (t_3 - t_2)$

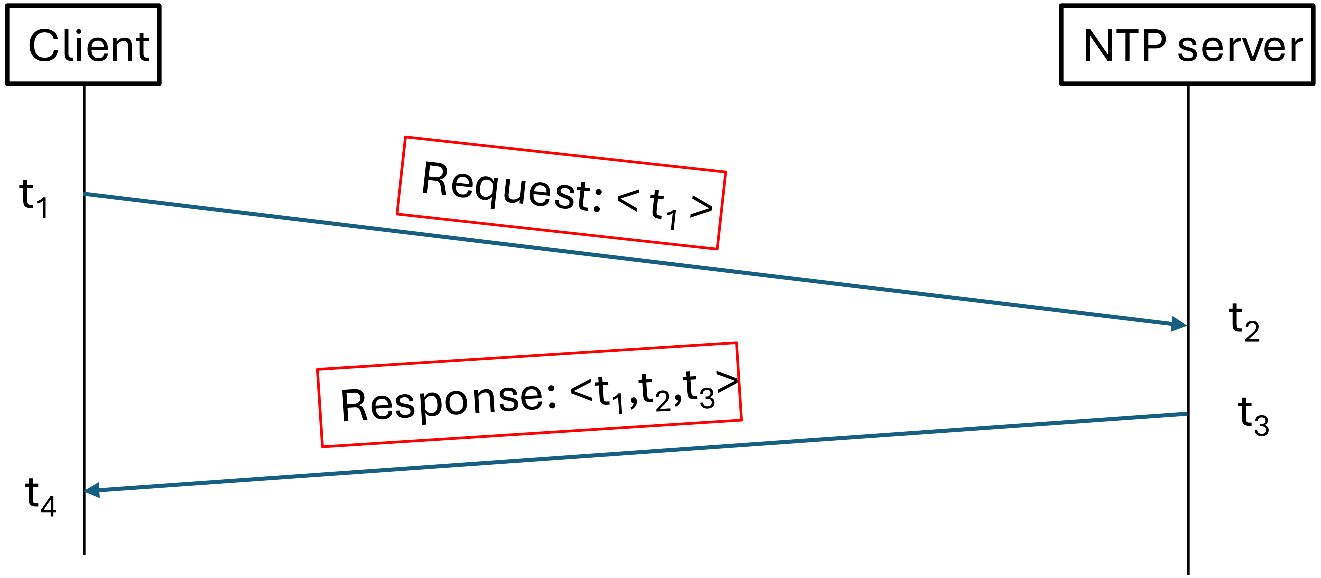


Clock Synchronisation(7)



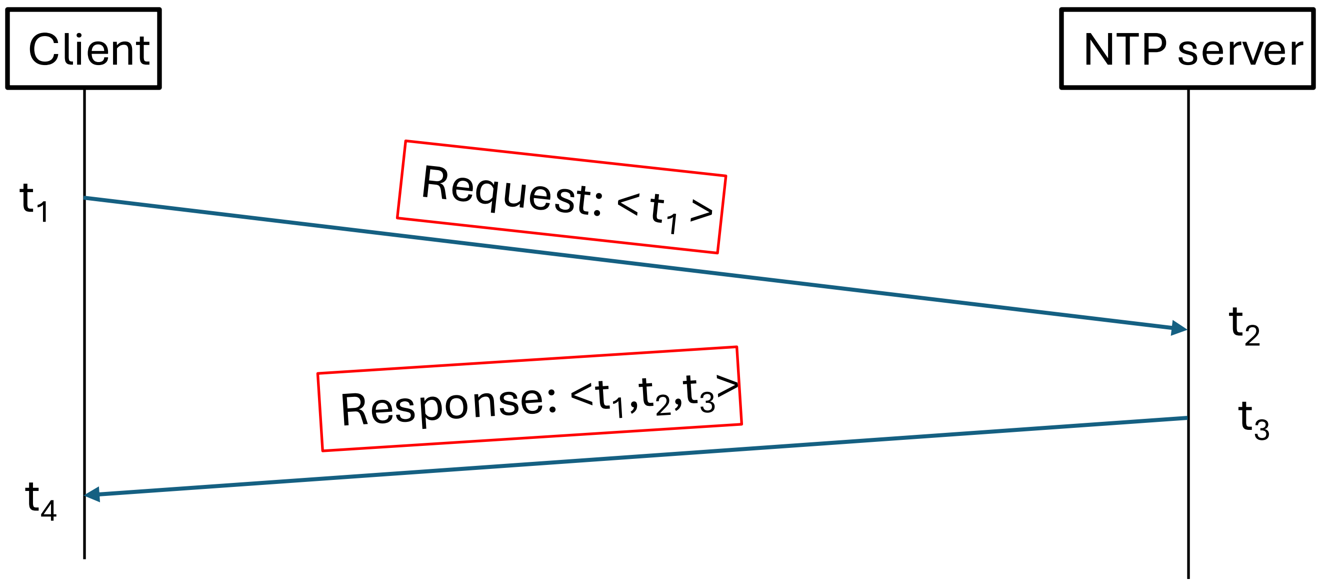
- RTT = $(t_4 - t_1) - (t_3 - t_2)$
- Client estimates the one-way delay on both request and response, $t_{1\text{-way}}$, as: RTT/2
  - This assumes that the network delays are symmetric both ways, which does not necessarily hold on the Internet!

Clock Synchronisation(7)



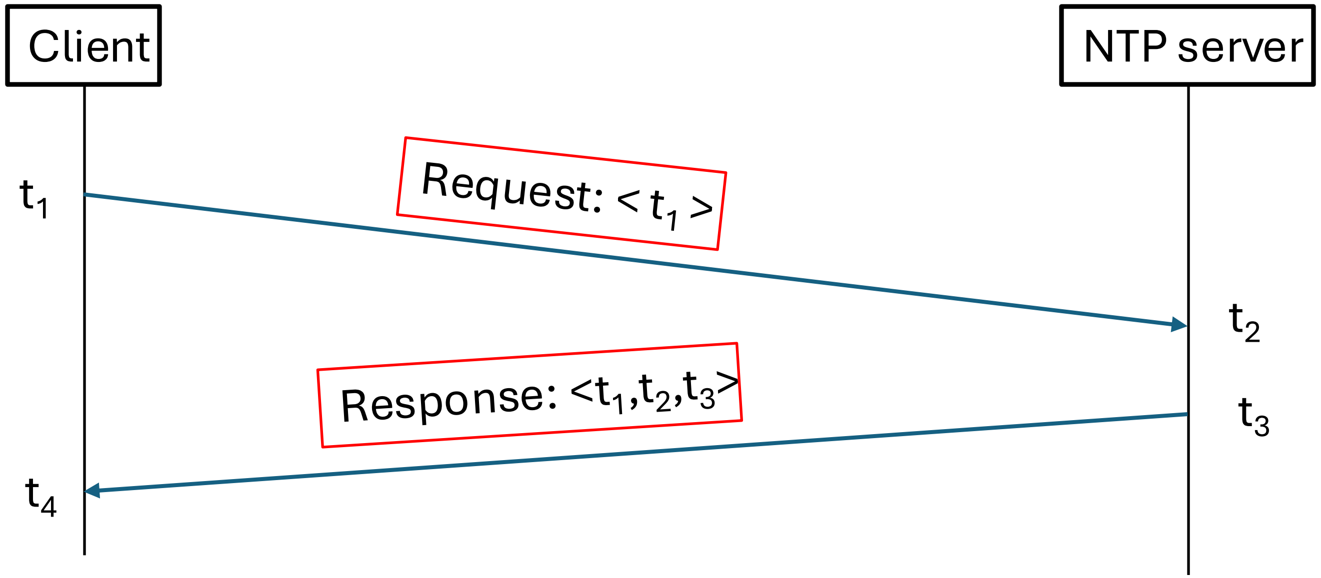
- Finally, how do we calculate the skew?
- Both are valid:
  - Skew_using the request: $t_2 – t_1 – t_{1\text{-way}}$
  - Skew using the response: $t_3 – t_4 + t_{1\text{-way}}$

Clock Synchronisation(8)



- Another approach: calculate average skew of request and response:
  - $= \dfrac{(t_2 - t_1 - t_{1-way}) + (t_3 - t_4 + t_{1-way})}{2}$
- Finally, we have: Clock Skew $= \dfrac{(t_2 - t_1) + (t_3 - t_4)}{2}$

Clock Synchronisation(9)



- How to adjust the local clock, given the skew?
  - Better to adjust slowly
  - Suddenly moving the clock forwards or backwards can have implications for applications (e.g., ones that measure elapsed time based on physical clock)

Clock Synchronisation(10)

- The clock synchronisation performed by NTP and similar protocols is not perfect
  - Even after syncronised with NTP, the clocks of two nodes can have significant skew
    - Sub-millisecond accuracy is possible only under ideal conditions
  - The accuracy can be low: 1-50 milliseconds, if
    - the communication delay between the nodes is large, and
    - the network delay in the two directions is asymmetric
- GPS is another time source
  - 31 orbiting satellites each carrying an atomic clock
  - Each broadcasts the current time at nanosecond-level granularity (at most 20-30 ns error)
  - With a GPS antenna, one can obtain the time with **micro-second level accuracy**
  - However, it requires minimum electromagnetic interference and clear line of sight
    - Not suitable for indoor computers particularly in a datacenter

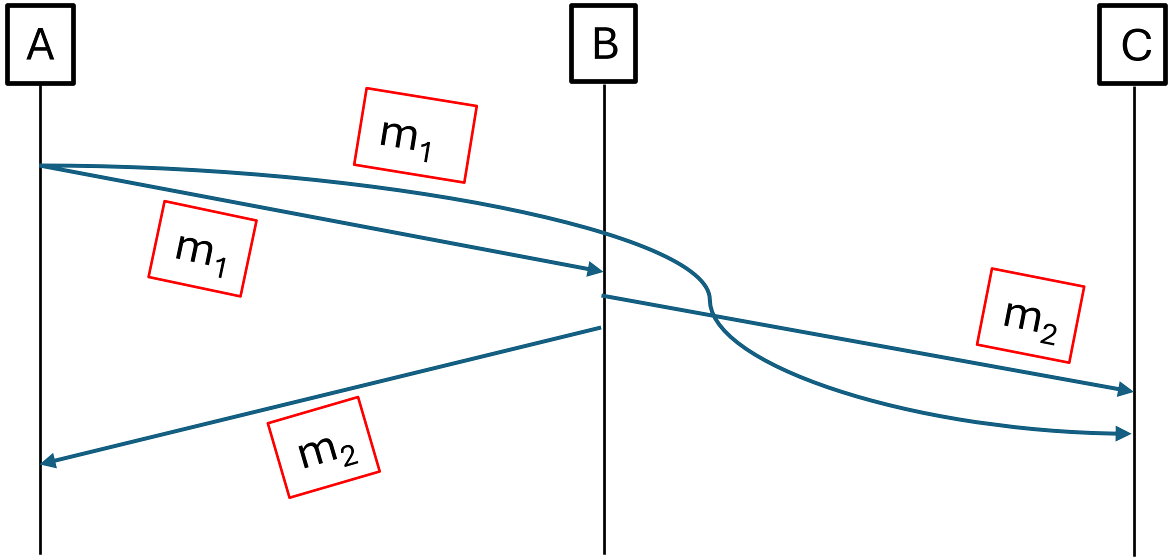
Representing time in Computers

- Timestamp: represents a specific point in time
- Two most common representations:
  - Unix time: number of seconds since 1 January 1970 00:00:00 UTC (the "epoch")
  - ISO 8601: year, month, day, hour, minute, second, and timezone offset relative to UTC. example: 2020-11-09T09:50:17+00:00
- gettimeofday() system call in Linux

Agenda

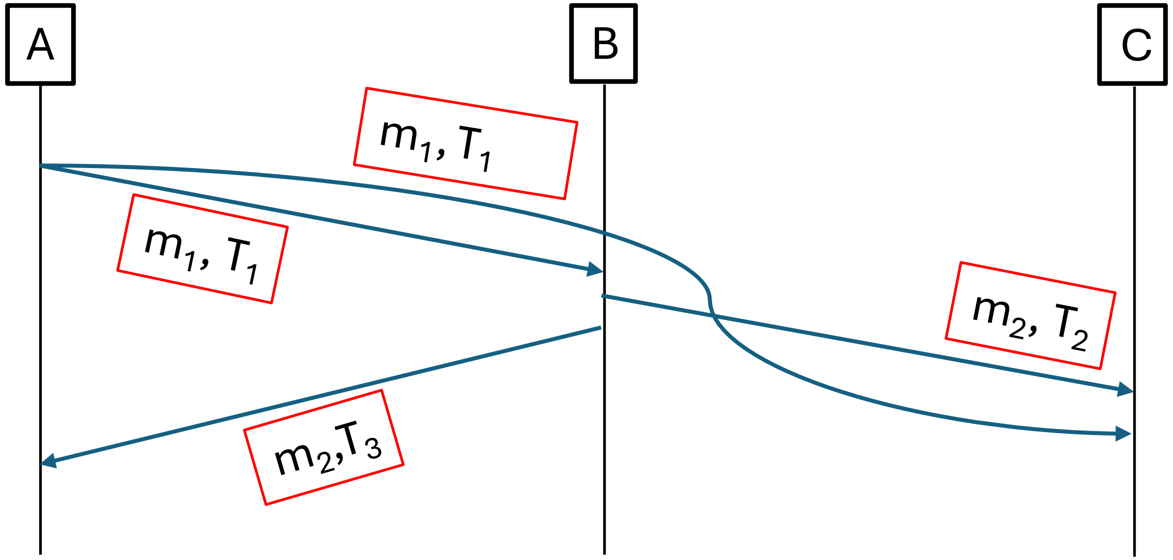
- How do systems keep accurate time?
- **How to order distributed events?**

Ordering of Events



- How can C know how to correctly order the messages?

Ordering of Events



- How can C know how to correctly order the messages?
- One idea:
  - A attaches its timestamp to m_1 at the time of sending m_1 to C and so does B when sending m_2 to C
  - C simply orders messages by their timestamps
- Would this provide the "**correct**" **order**?

Happens-before Relationship

- What do we mean by correct ordering?
  - A node would be interested in ordering causally-related events
  - “Creation of an object” is causally-related to “updating of the same object”
- We can only detect “potential causality”
  - If an event a happens-before b (i.e., a → b) then they are **potentially causal**; that is, “event a” **potentially caused** “event b”
- What events can we order in a distributed system?
  - Sending of a message at a node must happen before the receipt of the same message at a different node
  - Events happening on the same node can be ordered locally
- Apply transitive property: A sends message to B and C, B receives the message and then sends a message to A and C.
  - A’s sending of a message to B C’s → receiving of B’s message

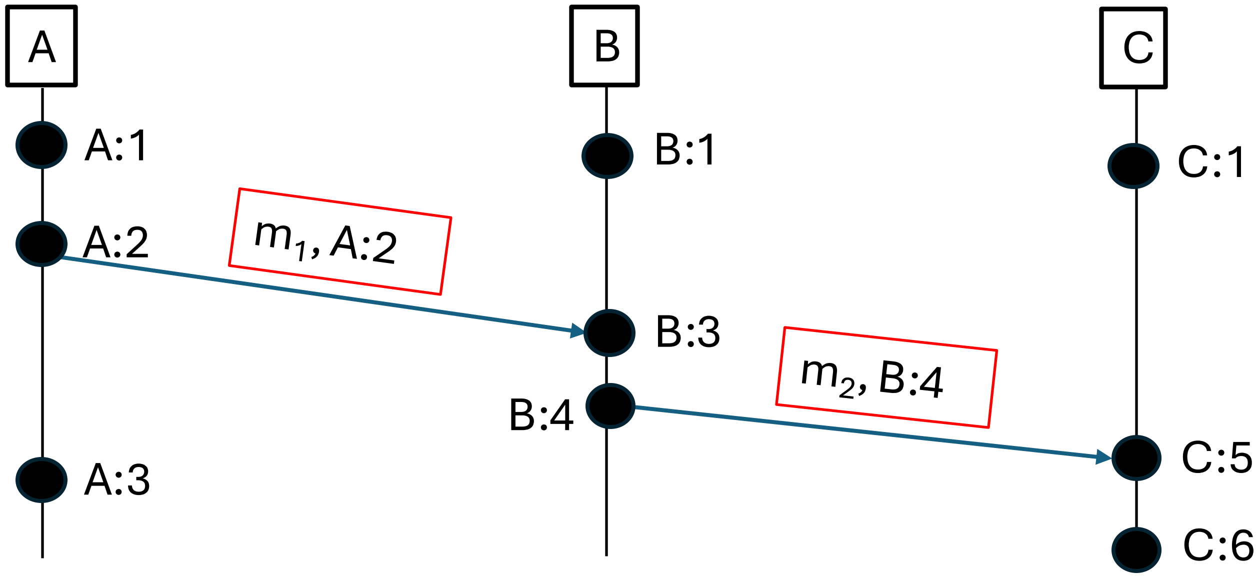
Logical Clock

- Logical clocks do not provide an actual, physical measure of time, but rather a consistent and meaningful ordering of events based on (potential) causality.
  - E.g., one can attach a monotonically increasing times to outgoing packets based on a (local) logical clock so that the receiver can order the incoming messages
- The objective is to assign a logical timestamp to every event…
  - So that we can determine the potential causality of events based on happens-before relationship:
    - If $T(e1) < T(e2)$ then $e1 \rightarrow e2$
- Two well-known examples of logical clock algorithms are:
  - **Lamport Clocks**
  - **Vector Clocks**

Lamport Clocks

- Each node maintains a local **counter c** that is incremented on each local "event"
  - E.g., sending of a message, local computation, I/O operations, system notifications, etc.
- When sending a message, the node attaches (i.e., piggybacks) the current value of c (as its timestamp for this event) in the message
- The receiver of a message **updates its local counter c** using the one in the message (m_c) as follows:
  - $c = \max(c, m_c) + 1$
- This leads to a partial ordering of events
  - Some events are not comparable, e.g., those with the same counter value

Lamport Clocks

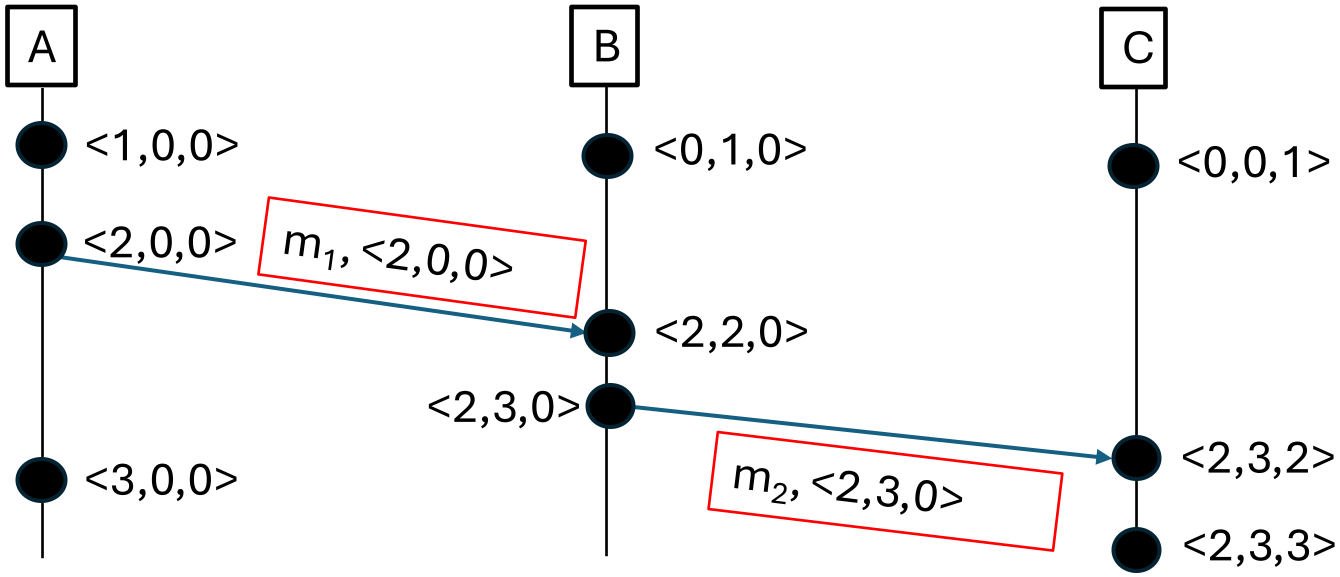


- A:1, B:1, and C:1 are incomparable – how about A:2 and C:1?
  - also incomparable
- If $e_1 \rightarrow e_2$ then $L(e_1) < L(e_2)$
- Is the converse true: If $L(e_1) < L(e_2)$, does this necessarily mean $e_1 \rightarrow e_2$?

Vector Clocks

- Assume n nodes in the system, **N** = $\langle N_1, N_2, \ldots, Nn \rangle$
- Vector timestamp of event e is $V_e = \langle T_1, T_2, \ldots, T_n \rangle$
  - Each node has a current timestamp T (number of local events at this node)
  - Each node also maintains the number of events at all the other nodes, forming a vector V
  - On event at node N_i, the node N_i increments its vector element V[i]
  - As before, attach current vector timestamp to each message
- Recipient merges message vector m_v into its local vector V
  - $V[i] = \max(m_v[i], V[i])$ for every $i \in \{1, \ldots, n\}$

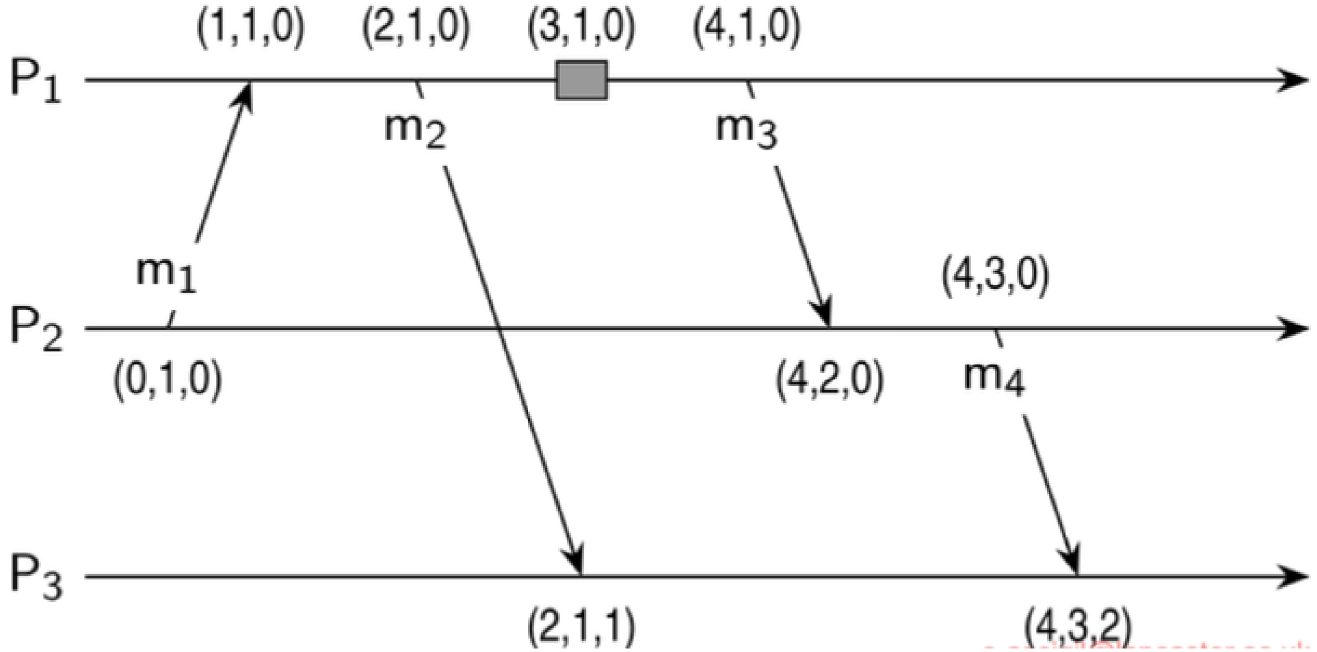
Vector Clocks



- $V_1 \leq V_2$ iff $V_1[i] \leq V_2[i]$ for all $i \in \{1,\ldots,n\}$
- For two events a and b:
  - If $V_a \leq V_b$ and $V_a \neq V_b$ then $a \rightarrow b$

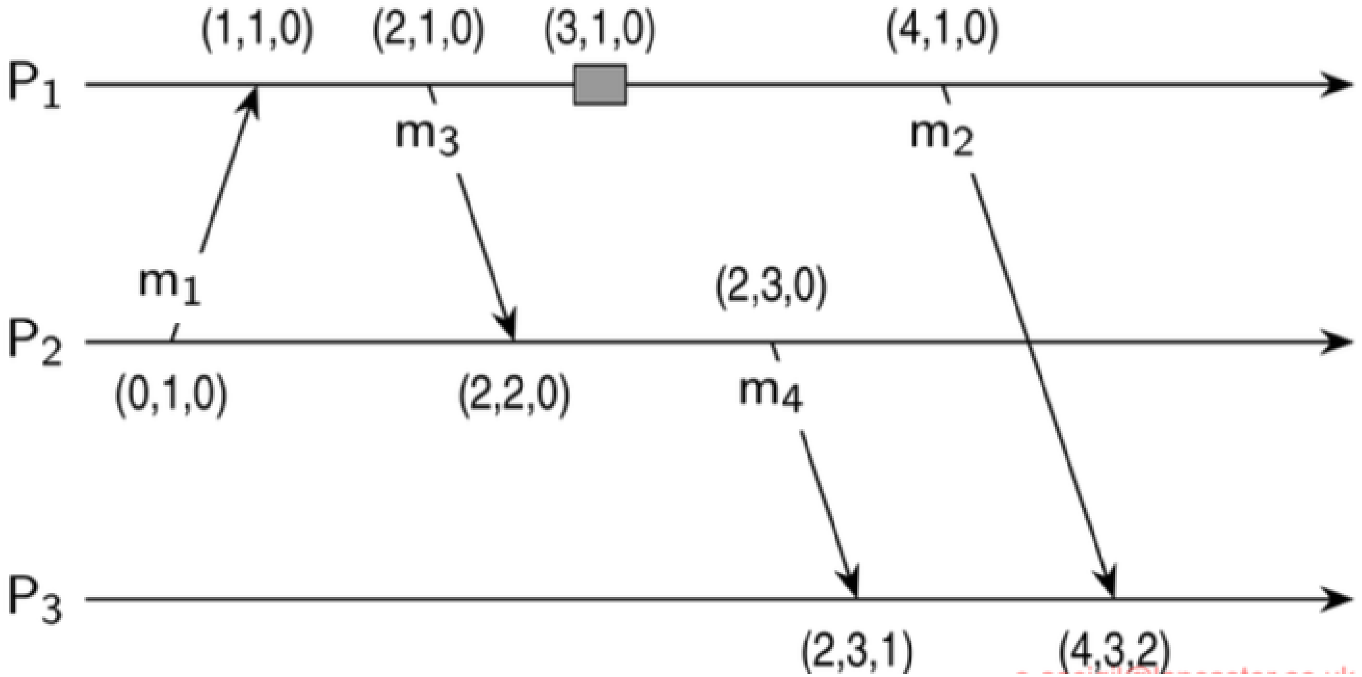
Vector Clock Examples (1)

- Which events are incomparable in the below example?



Vector Clock Examples (2)

- Which events are incomparable in the below example?



Vector Clocks

- The vector clock algorithm is a mechanism to compute the happens-before relationship
  - Each node tracks the number of events processed by other nodes
  - For two vector clocks V_a and $V_{b,}$ if $V_a \leq V_b$ and $V_a \neq V_b$ then a → b
    - The same is not true for the Lamport clocks!
- The vector clocks form a **partial order** (as shown in the previous slide)
  - Two vector clocks may be incomparable, for example: <1,2,4> and <2,1,3>

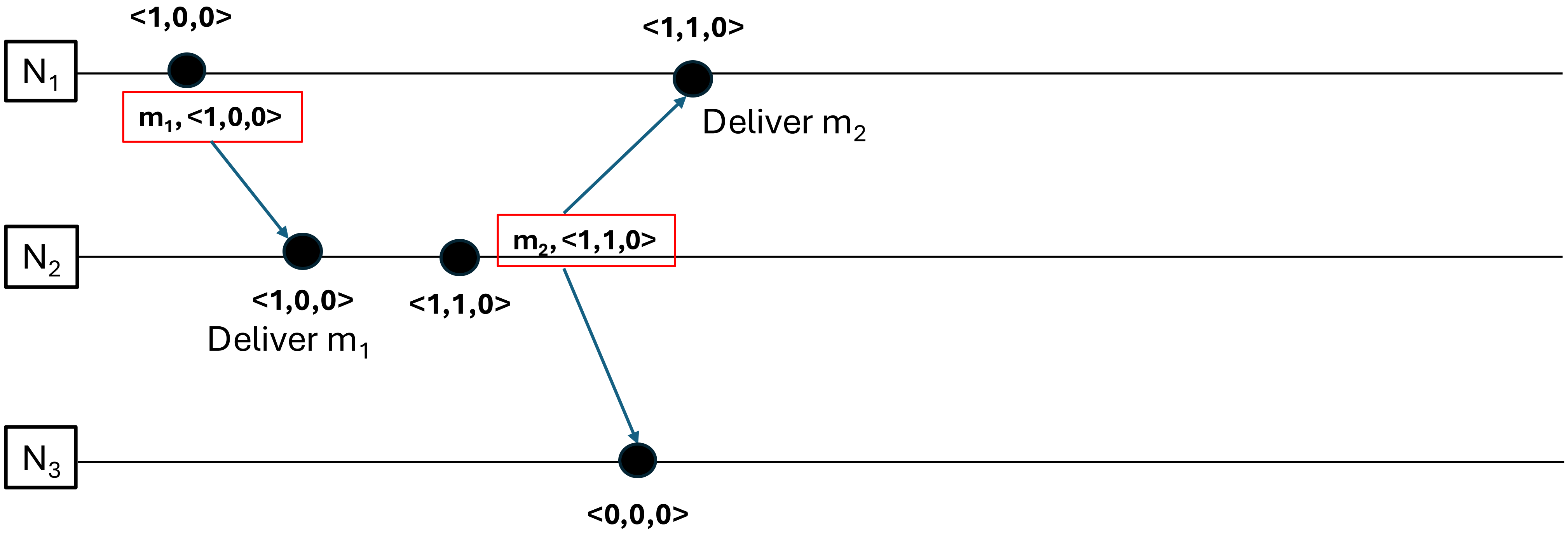
Causally Ordering Multicast Messages

- Remember causal ordering? (Week 5, Lecture 2)
- Let's assume that clocks are adjusted only when:
  - sending messages (at the sender) – same as before
  - **delivering messages** to the application (at the receiver) – **new rule**
  - messages are buffered until they can be delivered (see below) – **new rule**
- Vector clocks are updated following the same mechanisms as before:
  - Upon sending a message, a node N_i increments the i^{th} index of its vector clock $VC_i[i]$ by one (as discussed earlier)
  - When a Node N_j delivers a message m with timestamp TS(m), N_j will update $VC_j[i]$ to max{$VC_i[i]$, TS(m)[i]} for each i
- The delivery of a message m (sent by N_i) to the application layer at N_j is delayed until:
  - $TS(m)[i] = VC_i[i]+1$
  - $TS(m)[k] \leq VCj[k]$ for all $k \neq i$

Causally Ordering Multicast Messages

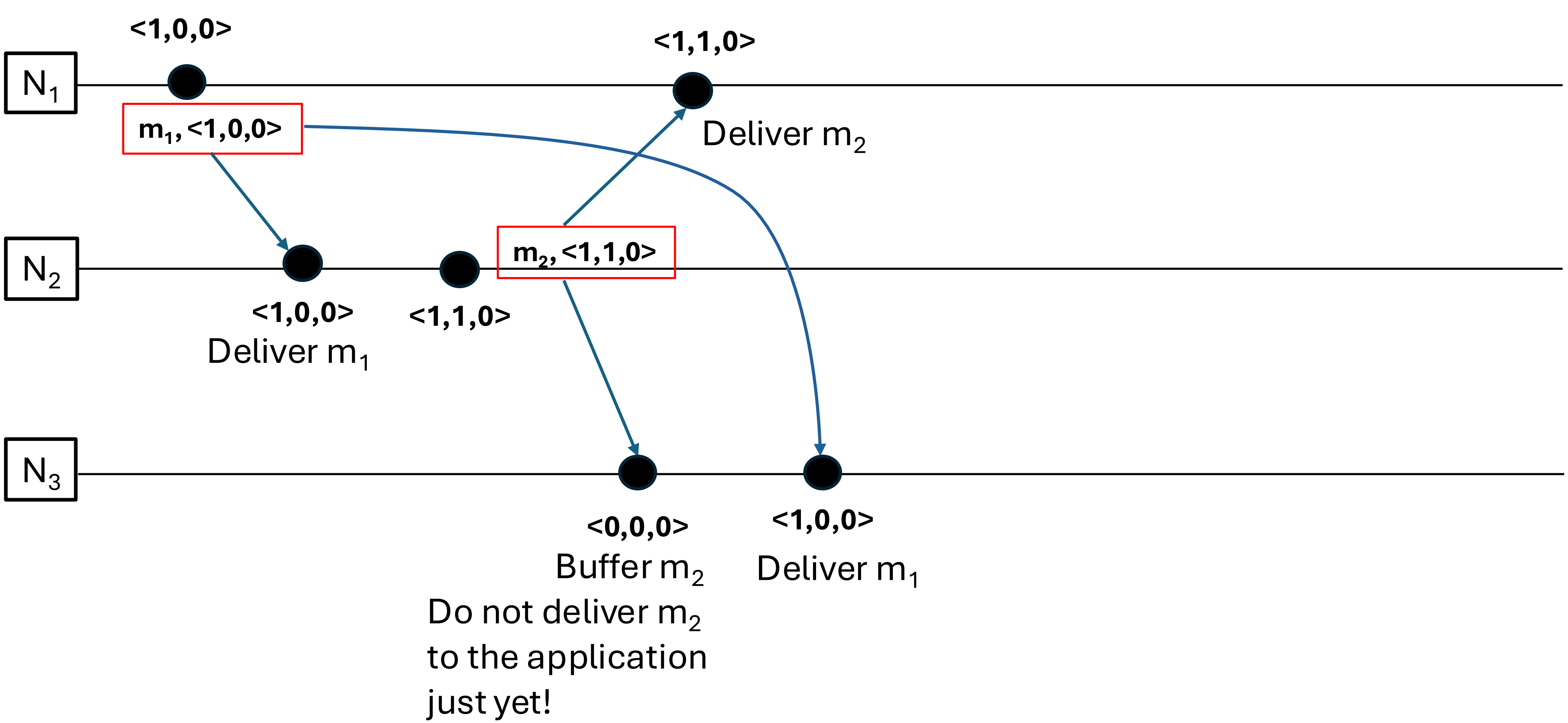
- Remember causal ordering? (Week 5, Lecture 2)
- Can we leverage Vector clocks to causally order group communication messages? Yes! (see below)
- Let's assume that clocks are adjusted only when:
  - sending messages (at the sender) – same as before
  - **delivering messages** to the application (at the receiver) – **new rule**
  - messages are buffered until they can be delivered (see below) – **new rule**
- Vector clocks are updated as follows:
  - Upon sending a message, a node N_i increments the i^{th} index of its vector clock $VC_i[i]$ by one (as discussed earlier)
  - When a Node N_j delivers a message m with timestamp TS(m), N_j will update $VC_j[i]$ to $\max\{VC_i[i], TS(m)[i]\}$ for each i
- The delivery of a message m (sent by N_i) to the application layer at N_j is delayed until:
  - $TS(m)[i] = VC_j[i]+1$
  - $TS(m)[k] \leq VC_j[k]$ for all $k \neq i$

Causally Ordering Multicast Messages

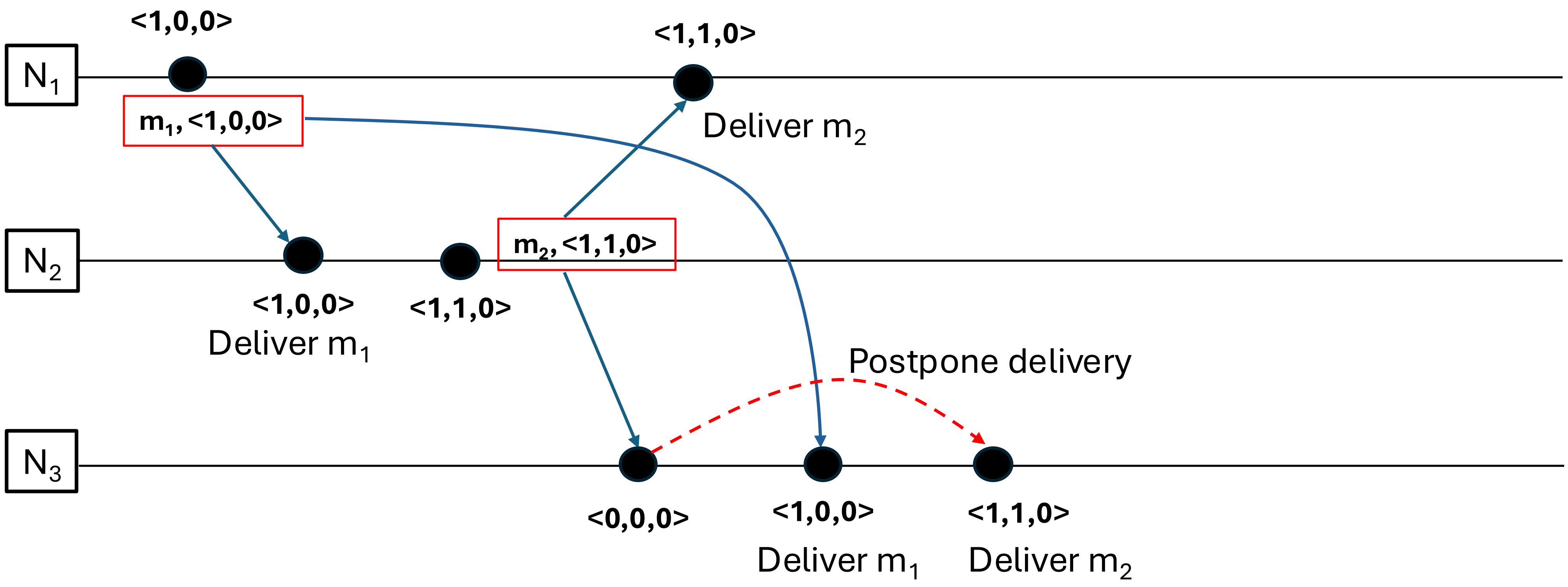


$(TS(m)[1] = 1) \leq (VC_3[1] = 0)$ does not hold! This means I am missing a message from N_1 – Place m_2 in a buffer

Causally Ordering Multicast Messages



Causally Ordering Multicast Messages



Learning Outcomes

- Understand how NTP protocol is used to synchronise physical clocks
- Understand the distinction between physical and logical clocks
- Differentiate between the two logical clocks, Lamport's and Vector Clocks, as mechanisms for ordering events in a distributed system
- Understand how Lamport's Clocks capture the partial ordering of events
- Grasp the concept of Vector Clocks and their ability to detect causality between multicast messages in a group communication

Reading Material

- Coulouris & al – Chapters 14.2, 14.4, 15.5.3