3. Time and Ordering

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2024/2025 Q2

Contents

- **Introduction**
- Physical clocks
- Logical clocks

Time

- Time in unambiguous in centralized systems
  - System clock keeps time, all entities use this
- No global time on distributed systems
  - Each node has a (crystal-based) system clock
    - Less accurate than atomic clocks
    - Results in **clock skew** (two clocks, two times) and **clock drift** (two clocks, two count rates)
    - Drifts 1 second every 11 days w.r.t. a perfect clock
  - Problem: An event that occurred after another may be assigned an earlier time
    - Use physical and logical clocks to deal with this

Contents

- Introduction
- **Physical clocks**
- Logical clocks

Physical clocks

- Physical clocks allow to synchronize nodes …
  i. with a master node (with a UTC receiver)
     - UTC: Universal Coordinated Time is an international standard based on atomic time that is broadcasted through short-wave radio and satellite

  ii. with one another
- … within a given bound
  - **Perfect** clock synchronization is not feasible
    - Synchronization limited by network jitter and clock drift
    - Typical accuracy of milliseconds
  - Clocks approximate real time but cannot generally be used to find out the **order** of events

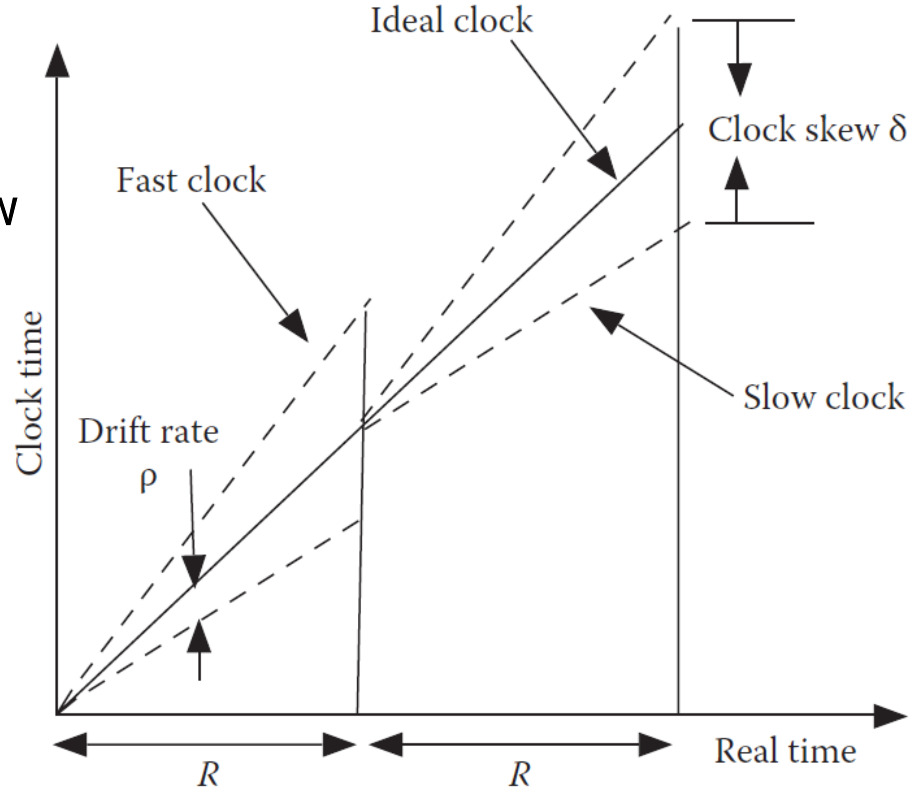
Physical clocks

- Synchronize at least every R < δ/2ρ to limit skew between two clocks to less than δ time units

R: resynchronization interval

ρ: maximum clock drift rate

δ: maximum allowed clock skew

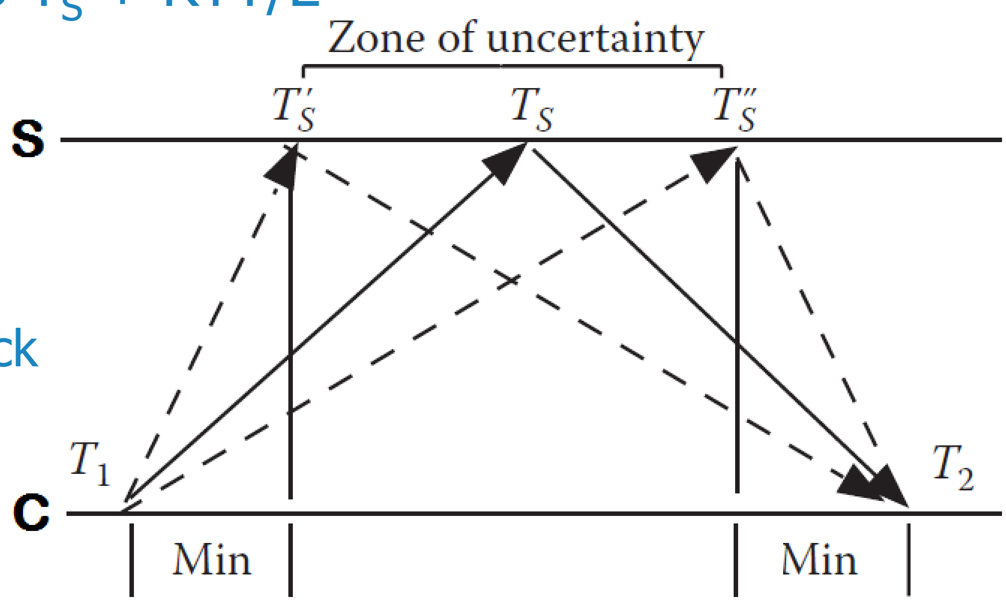


Cristian's algorithm

- Synchronize nodes with a server with UTC receiver <u>within a given bound</u>: **External synchronization**
  - Intended for intranets with a UTC-sync server

1. Each client asks the time to the server at every R interval
2. Client sets the time to T_S + RTT/2
   - RTT: round-trip time
     - $T_2 - T_1$
   - Assumes symmetrical latency
   - Accuracy of client's clock is ±(RTT/2 – Min)

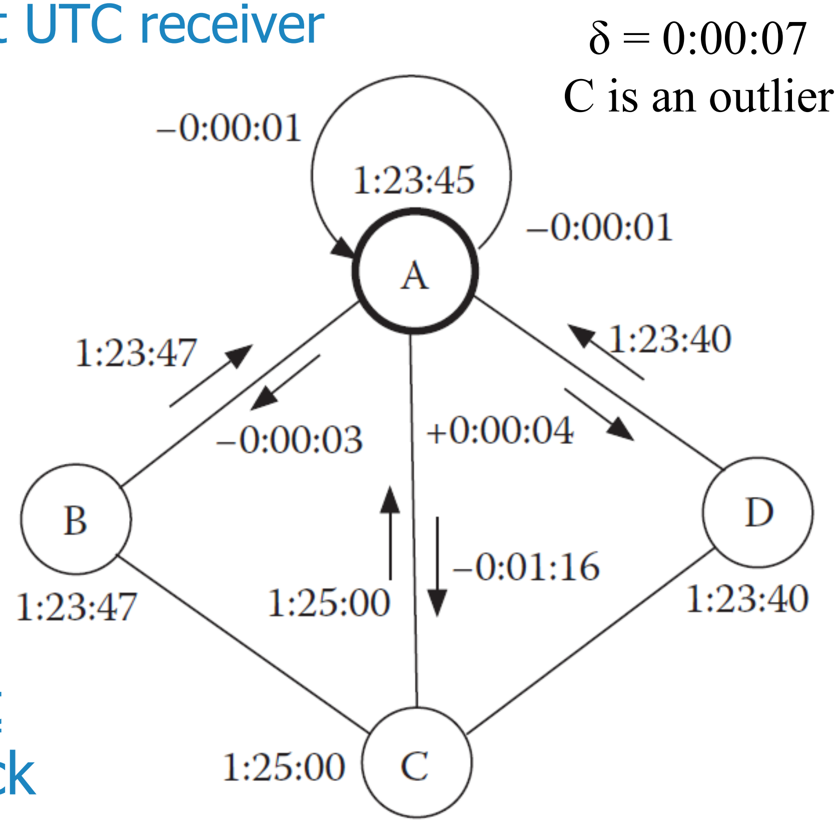
- NTP is based on a similar concept



Berkeley algorithm

- Keep clocks synchronized with one another within a given bound: **Internal synchronization**
  - Intended for intranets without UTC receiver

1. Master polls the clocks of all the slaves at every R interval
   - Adapts them by considering round-trip times
2. Master calculates a fault-tolerant average
   - Clocks lying outside the given bound are discarded
3. Master sends the adjustment to be made to each local clock



System clock discipline

- Time stepping can have side effects
    i. When setting the time forward
        - Some time instants are lost
        - Events scheduled at these times can be affected
    ii. When setting the time backward
        - Monotonicity (time always moves forward) is violated
        - Subsequent events can appear earlier than previous ones
- <u>Workaround</u>: Slew (<u>change gradually</u>) the time by speeding it up or slowing it down until the adjustment has been achieved

READING REPORT

[Neville-Neil16] Neville-Neil, G.V., *Time Is an Illusion, Lunchtime Doubly So,* Communications of the ACM, Vol. 59, No. 1, pp. 50-55, January 2016

Contents

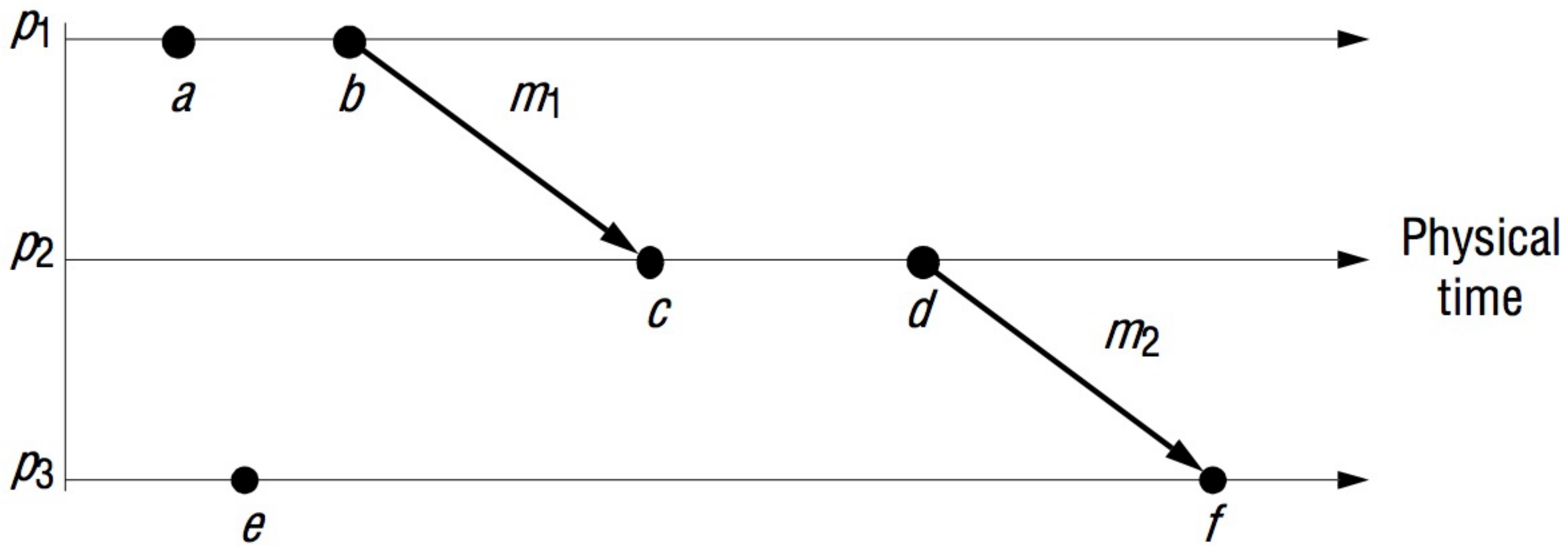
- Introduction
- Physical clocks
- **Logical clocks**

Happened-before relation

- Processes need to know if event 'a' happened before or after event 'b'
  - Agree on the **order** in which events occur rather than the **time** at which they occurred
- The happened-before relation
  - If **a** and **b** are two events in the same process, and **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**

Happened-before relation

- Example



a→b, b→c, c→d, d→f, a→f but a||e (concurrent)

➢ Happened-before relation defines a **partial ordering**

Logical clocks

- To capture the happened-before relation, we attach a timestamp **C(e)** to each event **e**, satisfying the following properties:
  1. If **a** and **b** are two events in the same process, and **a** $\rightarrow$ **b**, then **C(a) < C(b)**
  2. If **a** corresponds to sending a message **m**, and **b** to the receipt of **m**, then **C(a) < C(b)**
- How to attach a timestamp to an event when there is no global clock?

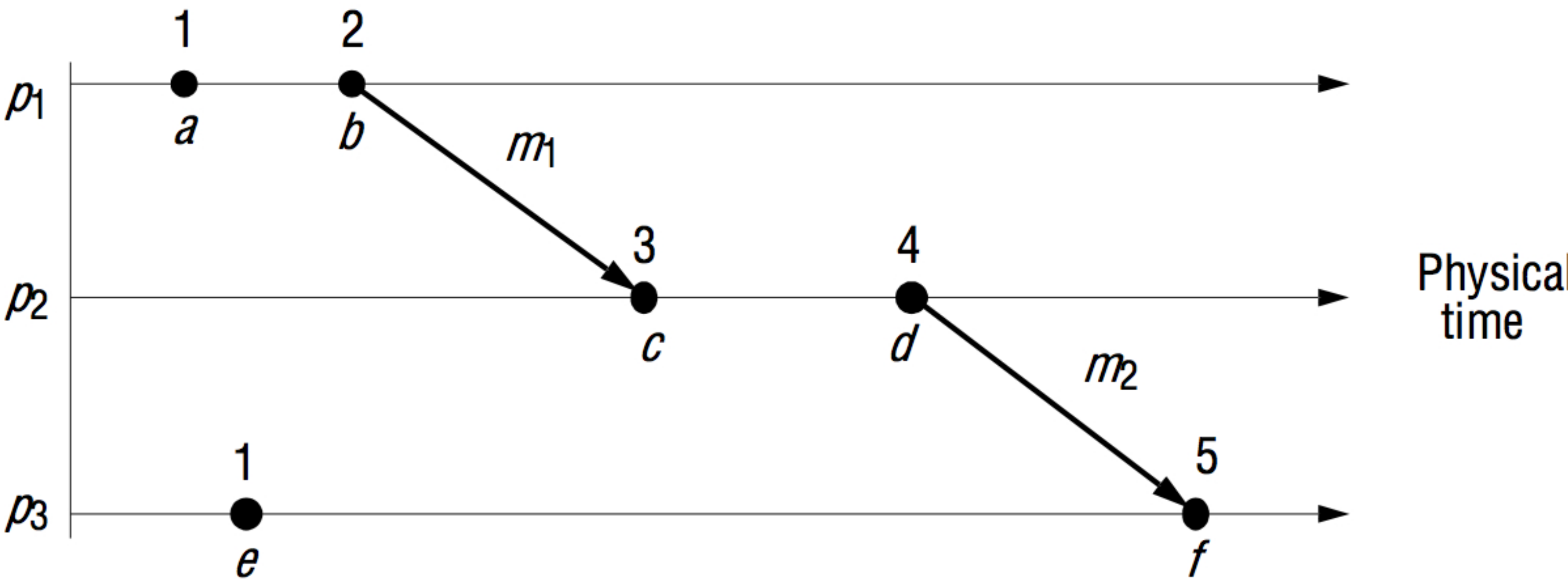
  $\Rightarrow$ Use <u>Lamport's logical clocks</u>

Lamport's logical clocks

- Each process P_i maintains a local counter C_i
- C_i is used to attach a timestamp to each event at P_i
- C_i is adjusted according to the following rules:
  1. When an event happens at P_i, it increases C_i by 1
  2. When P_i sends message m to P_j, sets $ts(m) = C_i$
  3. When P_j receives m, sets $C_j = max(C_j, ts(m))$, and then increases by 1

Lamport's logical clocks

- Example



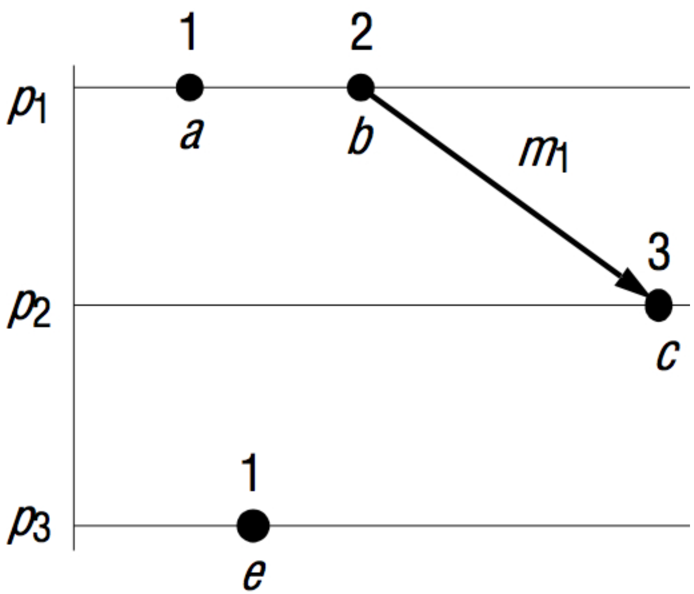
Note that C(e) < C(b) but b||e

Lamport's logical clocks

- Lamport's clocks define a partial order that is consistent with the happened-before relation
  - i.e. it is consistent with causal order
- What if we need totally-ordered clocks?
  - Use Lamport's clocks, but in case two of them are equal, process IDs will be used to break the tie
    - $\{C_i(a), i\} < \{C_j(b), j\}$ if:
      - $C_i(a) < C_j(b)$ OR
      - $C_i(a) = C_j(b)$ AND $i < j$
  - Can be used for instance to order the entry of processes to a critical section

Logical clocks

- Lamport's clocks don't guarantee that if **C(a) < C(b)** then 'a' causally preceded 'b' (**a** → **b**)



- C(a) < C(c), and 'a' causally preceded 'c' (a → c)
- C(e) < C(c), but 'e' did not causally precede 'c' (c || e)

⇒ Use vector clocks

Vector clocks

- Each process P_i has an array $VC_i[1…N]$
  - $VC_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j
    - i.e. $VC_i[j]$ is the logical clock of P_j at process P_i
- VC is adjusted as follows:
  1. When P_i sends a message m, it adds 1 to $VC_i[i]$, and sends VC_i with m as <u>vector timestamp</u> ts(m)
  2. When P_j receives a message m from P_i, it updates each $VC_j[k]$ to max($VC_j[k]$, ts(m)[k]), and then increments $VC_j[j]$ by 1

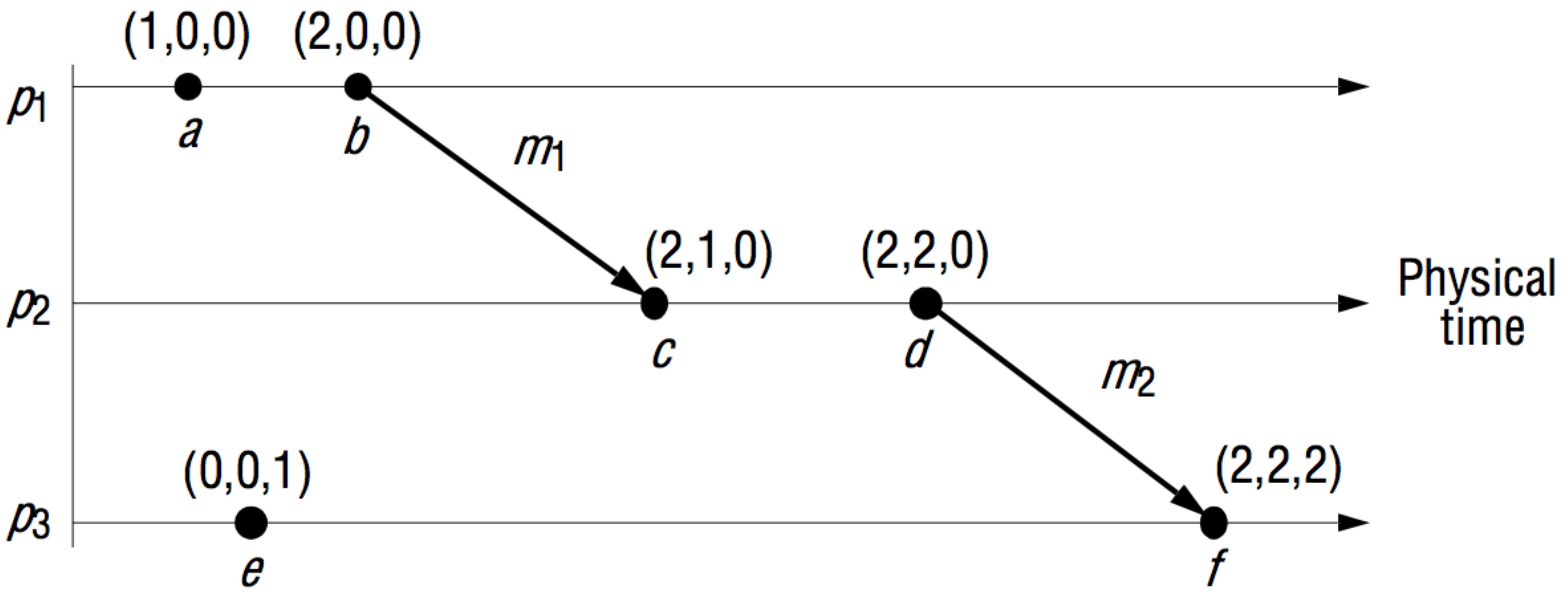
Vector clocks

- Vector clocks allow to detect causality
  - **$VC(a) < VC(b) \Leftrightarrow a \rightarrow b$**
    - If neither $VC(a) < VC(b)$ nor $VC(b) < VC(a)$, then $a||b$
- How to compare vector clocks?
  - $VC(a) < VC(b)$ if and only if:
    - $\forall k: 1,\ldots,N: VC(a)[k] \leq VC(b)[k]$

    AND
    - $\exists k: 1,\ldots,N: VC(a)[k] < VC(b)[k]$

Vector clocks

- Example



Neither VC(b) < VC(e) nor VC(e) < VC(b), so b||e

Summary

- We can synchronize physical clocks, but only within a given bound
- We cannot in general use physical time to find out the order of events
- Use logical clocks to find out events ordering
  - Lamport's clocks: $a \rightarrow b \Rightarrow C(a) < C(b)$
  - Vector clocks: $a \rightarrow b \Leftrightarrow VC(a) < VC(b)$
- Further details:
  - [Tanenbaum]: chapters 5.1 and 5.2
  - [Coulouris]: chapter 14