Clocks

ECE 454 / 751: Distributed Computing

Instructor: Dr. Wojciech Golab

wgolab@uwaterloo.ca

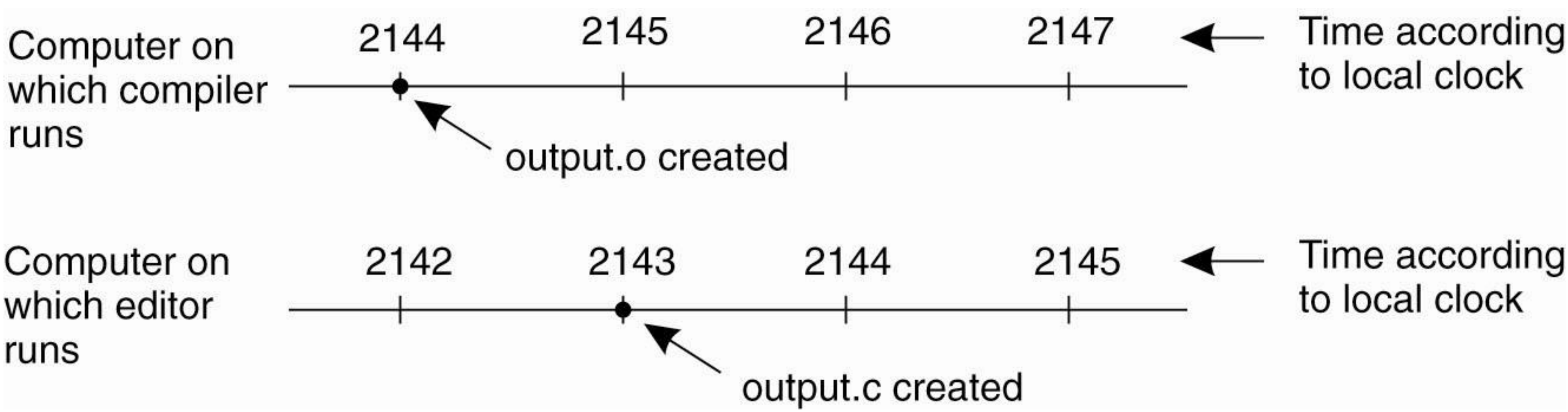
Slides are derived from A. S. Tanenbaum and M. Van Steen,
Distributed Systems: Principles and Paradigms, 2nd Edition, Pearson-Prentice Hall, 2006.

Learning objectives

- To develop a conceptual understanding of different timekeeping standards.
- To develop a working knowledge of the Network Time Protocol (NTP).
- To understand the theory of Lamport’s logical clocks and vector clocks.

Clock synchronization: motivation

Lack of synchronization among clocks of different machines leads to confusion, particularly regarding the order of events.



Calendars: counting days

Roman Calendar:

- lunar calendar, based on moon phases, months of 29 or 30 days
- initially 10 months per year = 304 days + 61 winter days unaccounted for
- reformed later on by adding two more months per year and an occasional intercalary month, but remained difficult to align with seasons

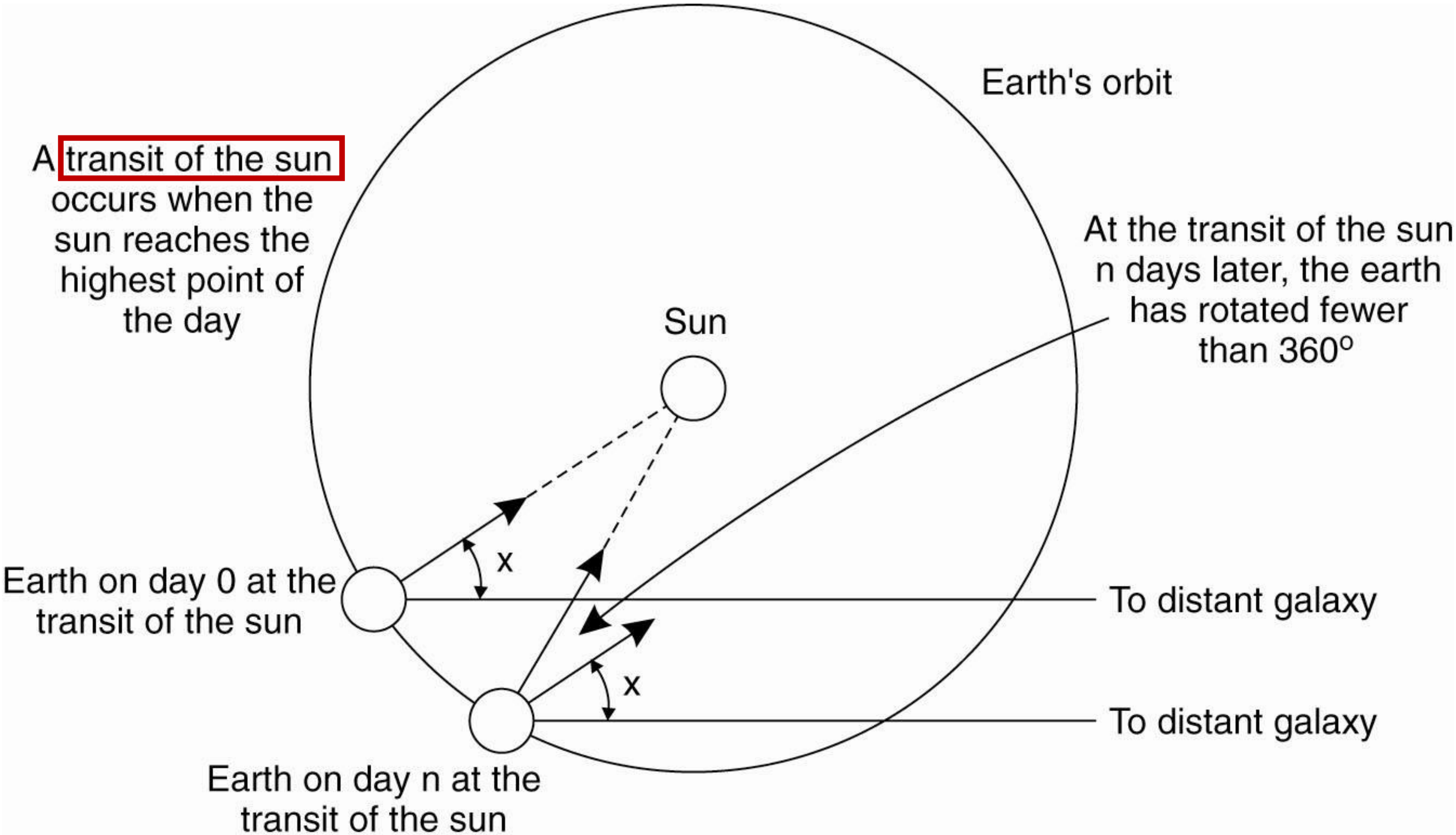
Julian Calendar (since 45 BCE, named after Julius Caesar):

- first solar calendar, based on Earth's rotation around the sun
- leap year was initially every three years, then every four years (too many!)
- not aligned with astronomical events like equinoxes and solstices

Gregorian Calendar (since 1582, named after Pope Gregory XIII)

- leap years calculated more carefully
- adopted by most of Canada in 1752 with 11 days skipped
- accuracy of 1 day in 7,700 years (with resect to vernal equinox)

Solar time: counting seconds



Timekeeping standards

Solar day: time interval between two consecutive transits of the sun. Not constant: seasonal variation in one year is up to 16 minutes from the mean!

TAI (Temps Atomique International): international time scale based on an average of multiple Cesium 133 atomic clocks.

UTC (Universal Coordinated Time): based on TAI and adjusted using leap seconds whenever the discrepancy grows to 800ms. Synchronized with Earth's rotation and currently behind TAI by tens of seconds.

Limitations of atomic clocks

- The **Hafele–Keating experiment**, conducted in 1971, confirmed Albert Einstein's theory of relativity with respect to time dilation.
- Four atomic clocks were flown around the world in opposite directions and then compared against clocks that remained "stationary" on the ground.
- Eastbound and westbound clocks both gained time due to gravitational time dilation.
- The eastbound clock lost time due to kinematic time dilation, while the westbound clock gained time.
- In the end, clocks flown in opposite directions differed by more than 200ns, in agreement with theoretical predictions.

Image source: https://en.wikipedia.org/wiki/Hafele%E2%80%93Keating_experiment

Textbook definitions

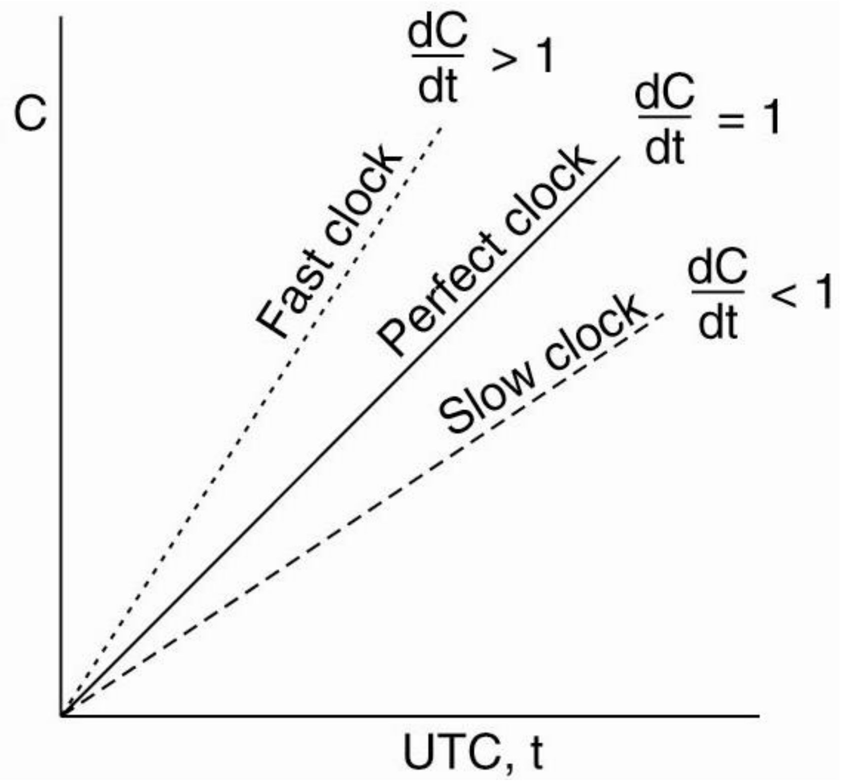
In the diagram C denotes a clock and t denotes a specific reference time, such as UTC. C(t) denotes the value of clock C at reference time t.

The **clock skew** of C relative to t is dC/dt – 1.

The **offset** of C relative to t is C(t) – t.

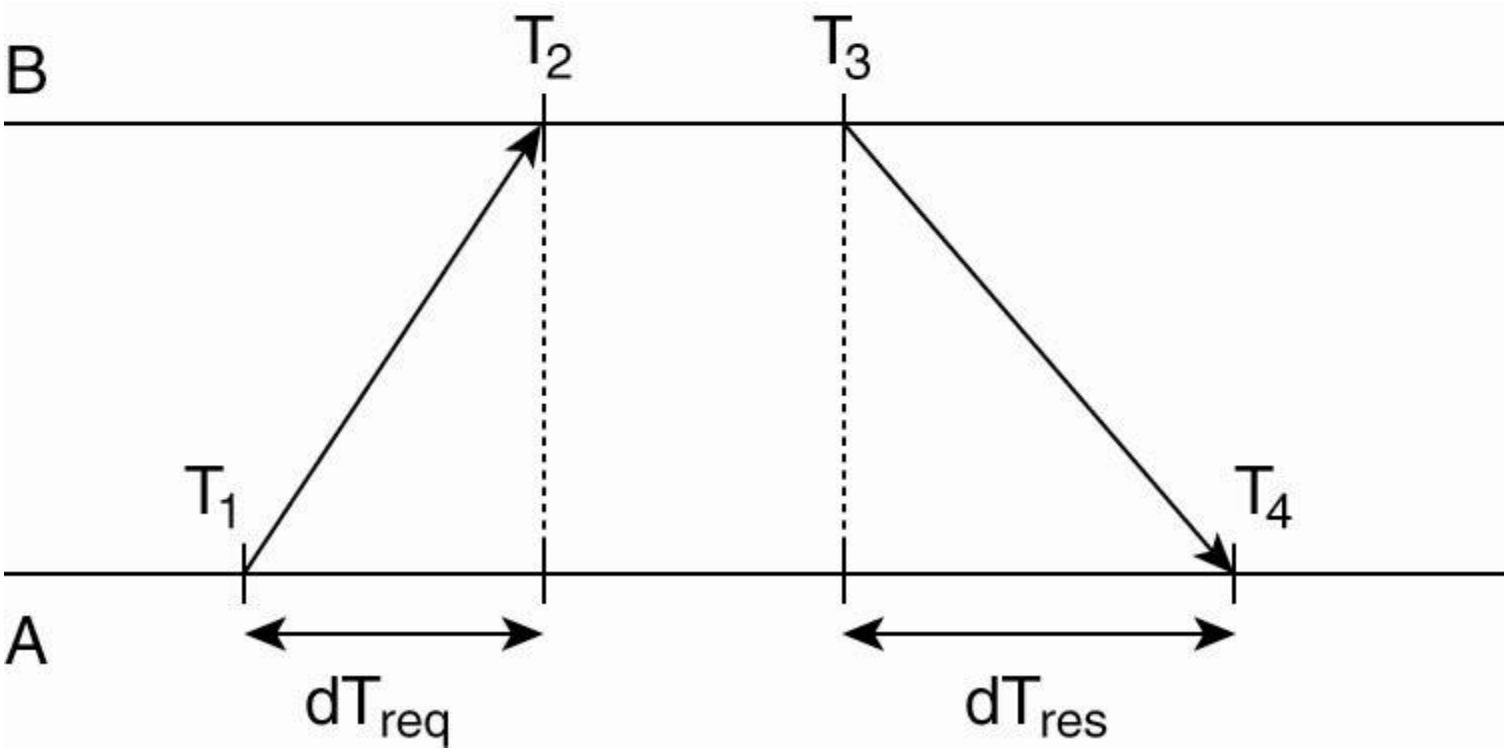
The **maximum drift rate** of C is a constant ρ such that

$1 - \rho \leq dC/dt \leq 1 + \rho$



Network Time Protocol (NTP)

Client at host A polls server at host B.



NTP offset and delay formulas

The offset of B relative to A is estimated as

$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

The (one-way) network delay between A and B is estimated as

$$\delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

NTP collects multiple (θ, δ) pairs, and uses the minimum value of δ as the best estimate of the delay. The corresponding θ is taken as the most reliable estimate of the offset.

NTP example calculation

- Assume client host A and server host B, as in the earlier slide.
- Let C_A and C_B denote the clocks at hosts A and B, respectively.
- Fix a reference time t and let $C_B(t) = C_A(t) + 20\text{ms}$ for all t.
- Assume Let $dT_{req} = 10\text{ms}$, $dT_{res} = 12\text{ms}$.
- Then NTP computes the following estimates:

$$\theta = \frac{30\text{ms} + 8\text{ms}}{2} = 19\text{ms}$$

$$\delta = \frac{\big(10\text{ms} + 12\text{ms} + (T_3 - T_2)\big) - (T_3 - T_2)}{2} = 11\text{ms}$$

- **Note:** the error in θ depends on the difference between dT_{req} and dT_{res} rather than on the absolute value of these quantities.

NTP practical considerations

- A reference clock such as an atomic clock is said to operate at **stratum 0**. A server with such a clock is a **stratum 1 server**.
- When host A contacts host B, it will only adjust its time if its own stratum level is higher than that of B. If A does adjust its time then A's stratum level becomes one higher than B's.
- Clocks must be adjusted (by slewing or stepping) carefully to ensure that time does not appear to flow backward.
- NTP accuracy is generally measured in tens of milliseconds.
- The Precision Time Protocol (PTP) promises to achieve much better accuracy (< 100ns) by leveraging hardware timestamping.

NTP on ecelinux

```
wgolab@ecehadoop:~$ ntpstat
synchronised to NTP server (129.97.128.9) at stratum 3
   time correct to within 46 ms
   polling server every 1024 s
```

Lamport clocks

In the absence of tightly synchronized clocks, processes can still agree on a meaningful **partial order of events**.

Leslie Lamport (2013 Turing Award winner) defined the partial order using the famous **“happens-before” relation**, often denoted by $\rightarrow$, which is the transitive closure of the following:

1. If *a* and *b* are events in the same process, and *a* occurs before *b*, then $a \rightarrow b$ is true.
2. If *a* is the event of a message being sent by one process, and *b* is the event of the message being received by another process, then $a \rightarrow b$ is also true.

Events *a* and b are **concurrent** if neither $a \rightarrow b$ nor $b \rightarrow a$.

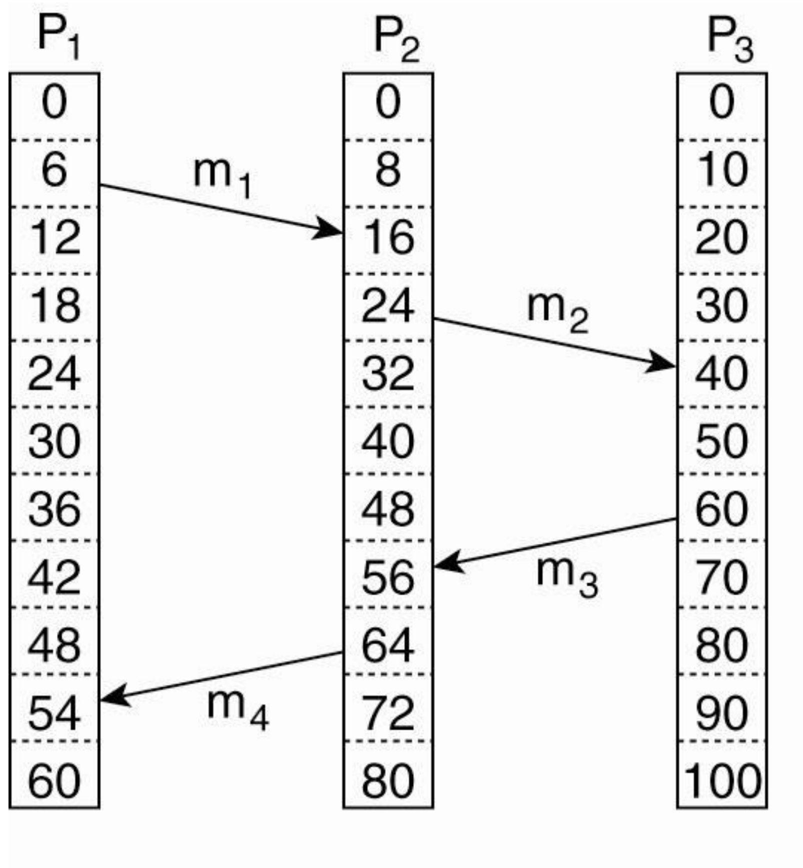
Note: Lamport’s “happens-before” relation is <u>not</u> the same as the one used by Herlihy and Wing to define linearizability.

Before Lamport clocks …

Three processes are shown with local clocks running at different frequencies.

The numbers shown in boxes are times at which different events happen, such as the sending of message m_1 or receiving of message m_2.

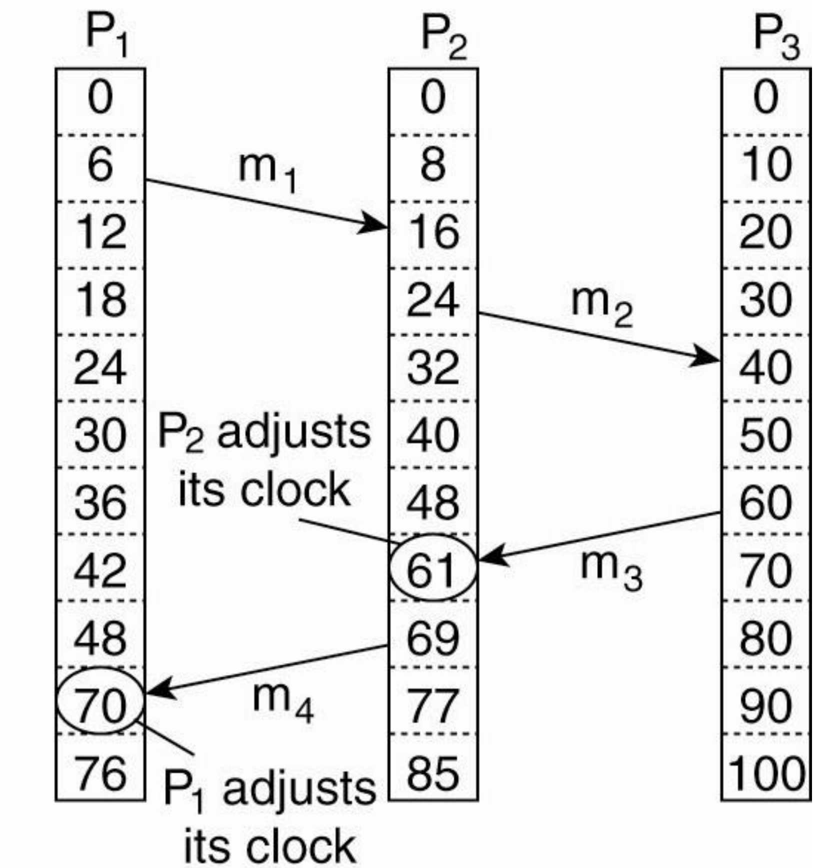
Message m_4 is sent at time 64 and received at time 54, leading to confusion.



(a)

Lamport clocks in action

Lamport's algorithm corrects the clocks, ensuring that the logical time at each process is consistent with the "happens before" relation.



(b)

Lamport clock algorithm

Algorithm for updating counter C_i at process P_i:

- In general, before executing an event (i.e., before sending a message or before delivering a received message to the application), process P_i increases its own counter C_i.
- When P_i sends a message *m* to P_j, it tags *m* with a timestamp *ts*(*m*) equal to the time C_i after incrementing C_i.
- Upon the receipt of a message *m*, process P_j adjusts its own local counter to $C_j := \max\{ C_j , ts(m) \}$, then increments C_j before delivering the message to the application.

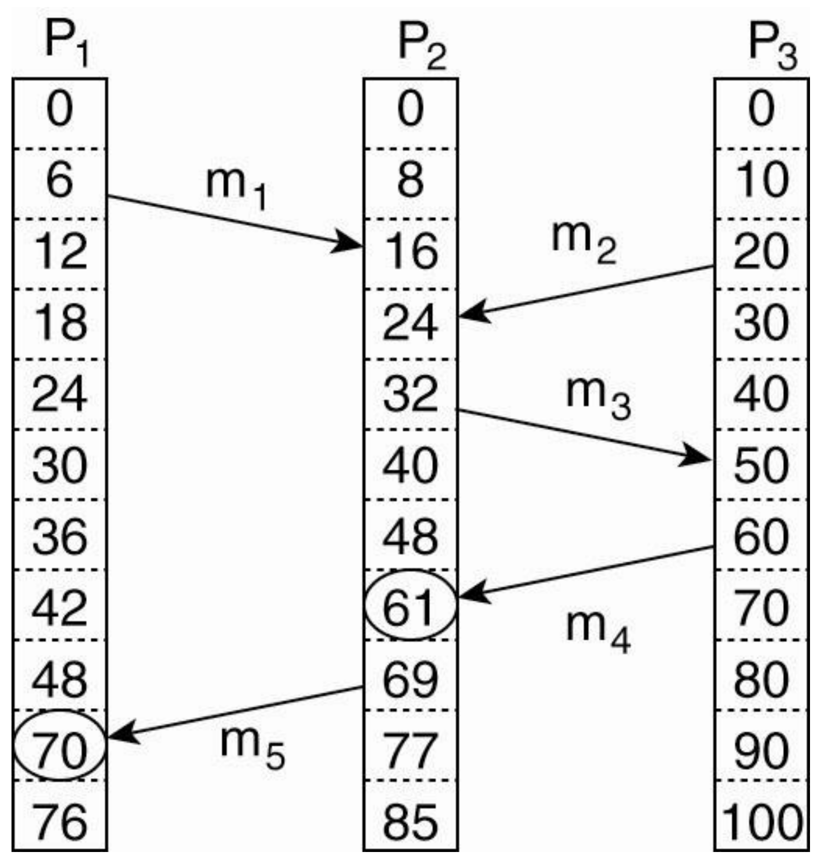
Note: The counter C_i represents the **logical time** at process P_i.

Lamport clocks in action (cont.)

Lamport clocks ensure that if $a \rightarrow b$ then $C(a) < C(b)$, where $C(a)$ and $C(b)$ denote the logical times of events a and b, respectively.

However, $C(a) < C(b)$ does not imply $a \rightarrow b$. In that sense, Lamport clocks do not properly capture causality.

Compare the logical times of the receive event of m_1 and the send event of m_2.



Vector clocks

Vector clocks represent logical time, similarly to Lamport clocks. They are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

1. VC_i [i] is the number of events that have occurred so far at P_i. In other words, VC_i [i] is the local logical clock at process P_i.
2. If VC_i [j] = k then P_i knows that k events have occurred at P_j. Thus, it is P_i's knowledge of the local time at P_j.

Note: In this context, "knows" means that one process learned about the state of another process by receiving a message from it, either directly or indirectly through another process.

Vector clock algorithm

Algorithm for updating vector clock at process P_i:

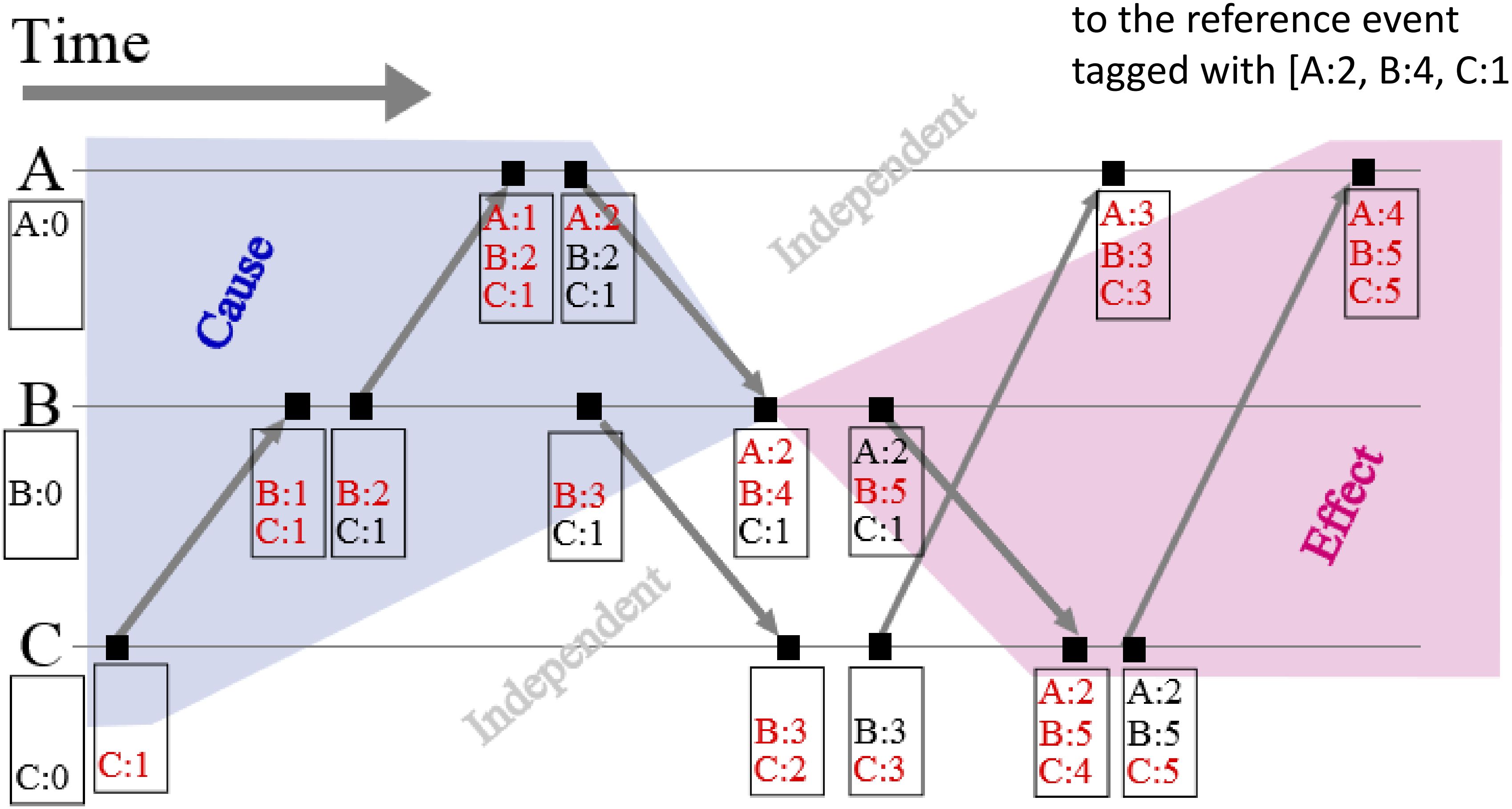
- In general, before executing an event, process P_i increments its own counter by assigning VC_i [*i*] := VC_i [*i*] + 1.
- When process P_i sends a message *m* to P_j, it sets *m*'s (vector) timestamp *ts*(*m*) equal to VC_i after incrementing VC_i [*i*].
- Upon the receipt of a message *m*, process P_j adjusts its own vector by setting

  VC_j [*k*] := max{ VC_j [*k*], *ts*(*m*)[*k*] } for each *k*,

  and then increments VC_j [*j*] before delivering the message to the application.

Vector clocks in action

Note:
"Independent" means concurrent with respect to the reference event tagged with [A:2, B:4, C:1].



Source: http://upload.wikimedia.org/wikipedia/commons/5/55/Vector_Clock.svg

Vector clock properties

Vector clocks provide a complete characterization of causality among pairs of events. Given two vector clocks $VC_i[1..N]$ and $VC_j[1..N]$, representing events *i* and *j*, we say that:

- event *i* happens before event *j* if $VC_i[k] \leq VC_j[k]$ for all elements k, and $VC_i[k'] < VC_j[k']$ for at least one element k'
- event *j* happens before event *i* if $VC_j[k] \leq VC_i[k]$ for all elements k, and $VC_j[k'] < VC_i[k']$ for at least one element k'
- otherwise, events *i* and *j* are concurrent

Note: To apply the above definition to the previous slide, treat any missing vector elements as zeroes.