Parallel and Distributed Computing

Dr. Ali Sayyed
Department of Computer Science
National University of Computer & Emerging Sciences

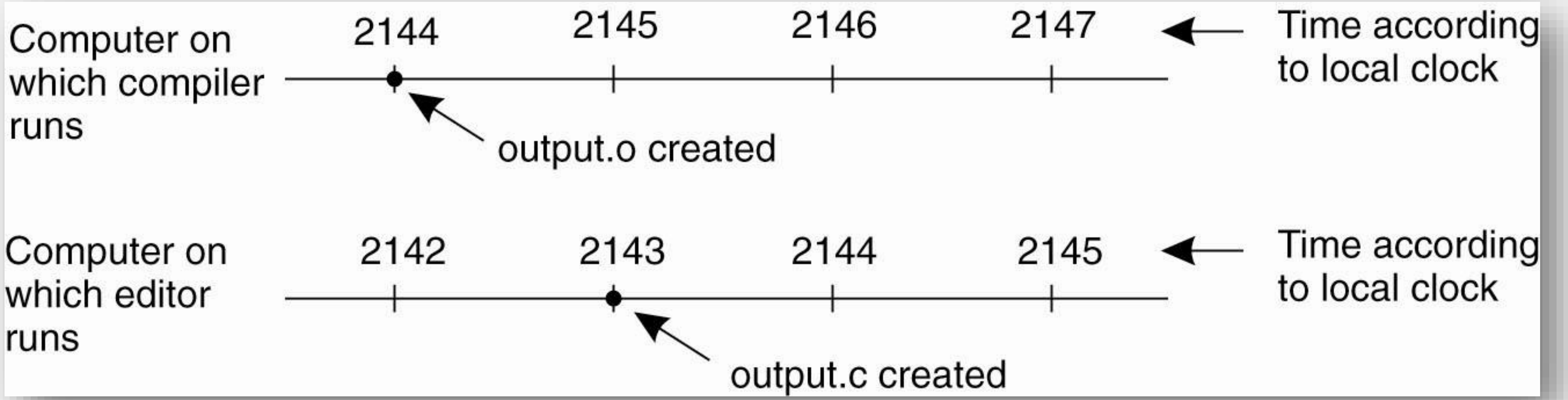
Synchronization and Coordination

Time in Centralized & Distributed Systems

- In a **centralized system**, time is consistent and unambiguous
  - there is **single system clock**.
- When a process requests the current time, it simply asks the operating system.
- If Process A requests the time first and Process B requests it shortly after, B will always receive a time that is equal to or greater than A's time—**never less**.
- In a **distributed system**, each machine has its **own independent clock**, leading to clock differences.
  - **Synchronizing** time across multiple machines becomes **complex and non-trivial**.

🕒 Clock Synchronization

- **Problem**: Different machines have independent clocks, leading to time differences.
- **Impact**: Events may be timestamped incorrectly, affecting synchronization.
- **Example**: In Unix make, if timestamps are misaligned, a file modified later may appear older, leading to errors.



⌛ Physical Clocks

- **Mean Solar Day**
  - Measured from the position of sun

- In the 1940s, it was established that the period of the earth's rotation is not constant and **gradually slowing down.**

- This change occurs due to gravitational interactions between the Earth, Moon, and Sun.

- Geologists now believe that **300 million years ago**, a year had **approximately 400 days** instead of 365.

⏱️ Atomic Clocks

- In 1948, it was possible to measure time more accurately using **cesium 133** atom.
- Cesium atomic clocks are now maintained in **multiple laboratories** worldwide for accurate time measurement.
- Periodically, **each lab reports its clock ticks** to the Bureau International de l'Heure (BIH) in Paris.
- The BIH averages the data to **compute International Atomic Time (TAI).**
- TAI seconds are **constant**, unlike solar time, which fluctuates due to Earth's slowing rotation.
- Leap seconds are added when needed to stay synchronized with the Sun.

Clocks in Computers

- Computers have a **circuit** for keeping track of time.
- This **circuit** has usually a precisely machined **quartz crystal** which **oscillate** at a well-defined frequency.
- Each crystal have two registers, a counter and a holding register.
- Each oscillation, decrements the counter by one. When the counter gets to zero, an interrupt is generated, and the counter is reloaded from the holding register.
- So, the circuit can be programmed to generate an interrupt (called one clock tick) 60 times a second, or at any other desired frequency.
- With a single computer and a single clock, it does not matter much if this clock is off by a small amount

Clock Skew

- The oscillation of the crystal oscillator is usually fairly stable, it is **impossible to guarantee** that the crystals in different computers all **run at exactly the same frequency**.
- In practice, when a system has n computers, all n crystals will run at slightly different rates, causing the clocks gradually to get out of sync.
- This difference in time values is called **clock skew**.
- As a consequence, programs that expect the time associated with a file, object, process, or message to be correct can fail, as we saw in the make example above.

Clock synchronization algorithms

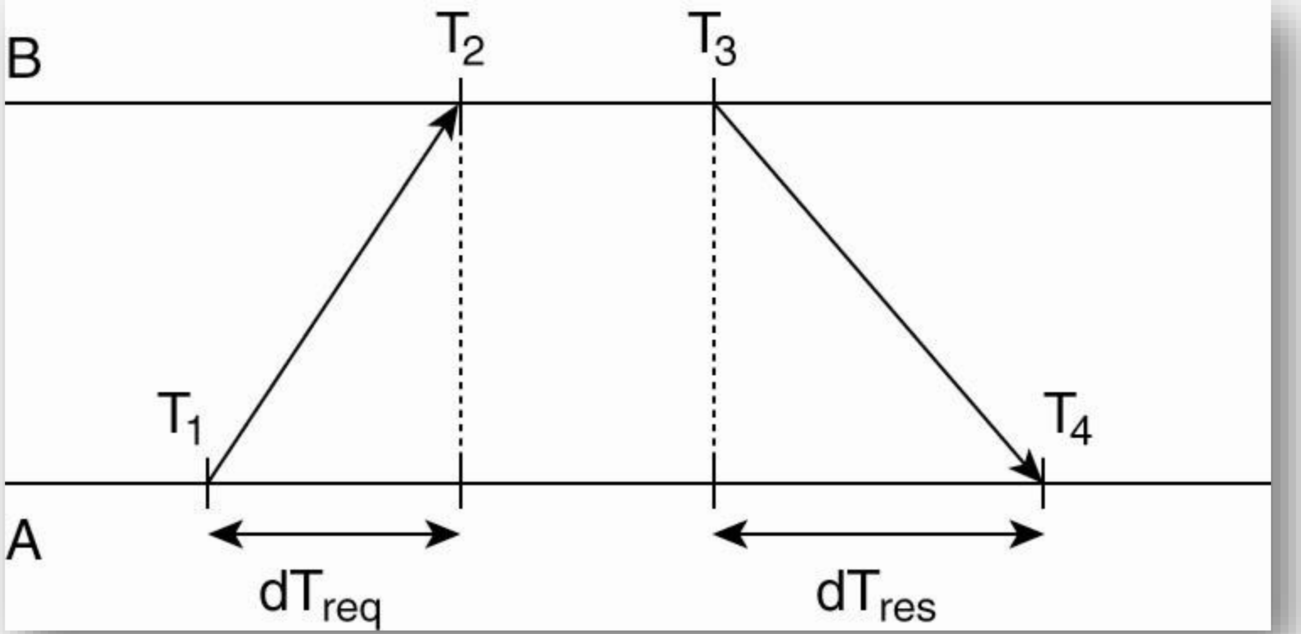
- The goal of clock synchronization algorithms is to keep the deviation between the clocks of any two machines in a distributed system, within a specified bound, known as the **precision**.

- Note that precision refers to the deviation of clocks only between machines that are part of a distributed system. When considering an external reference point, like UTC, we speak of **accuracy**

- The whole idea of clock synchronization is that we keep clocks precise, referred to as **internal synchronization** or accurate, known as **external synchronization**

Network Time Protocol

- A common approach in many protocols and originally proposed by Cristian [1989], is to let clients contact a time server. The server can accurately provide the current time, for example, because it is equipped with a UTC receiver or an accurate clock.



Getting the current time from a time server.

Network Time Protocol

- **Round-Trip Delay Calculation**
  - d=(T4−T1)−(T3−T2)
  - (T4−T1) = Total time elapsed according to the client.
  - (T3−T2) = Time spent at the server (processing delay).
- **Clock Offset Calculation**

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

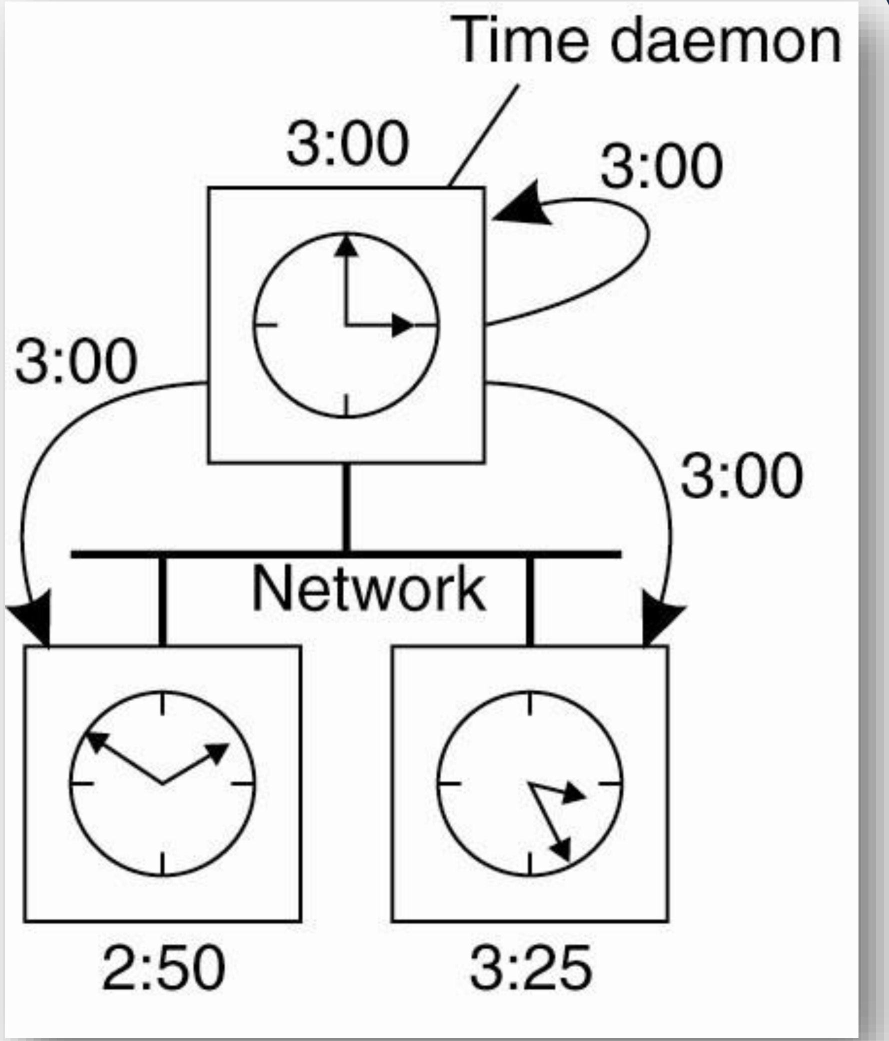
- If A’s clock is fast, θ < 0, A should, in principle, set its clock backward.
- Any such change must be introduced gradually.

The Berkeley Algorithm

- In many clock synchronization algorithms the time server is passive. Other machines periodically ask it for the time. All it does is respond to their queries.
- In Berkeley Unix exactly the opposite approach is taken. Here the time server (actually, a time daemon) is active, polling every machine from time to time to ask what time it is there.
- Based on the answers, it **computes an average time** and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved.
- The time daemon's time must be set manually by the operator periodically.

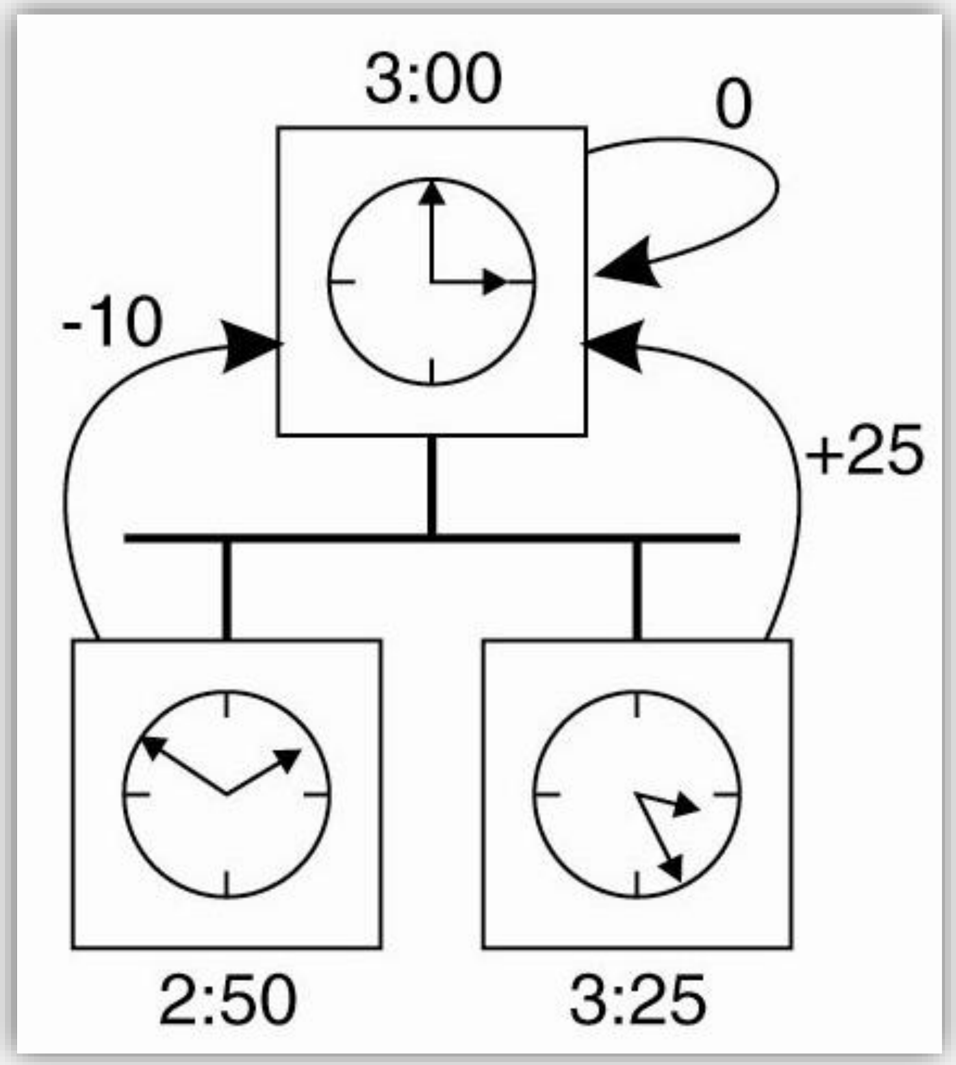
The Berkeley Algorithm (1)

The time daemon asks all the other machines for their clock values.



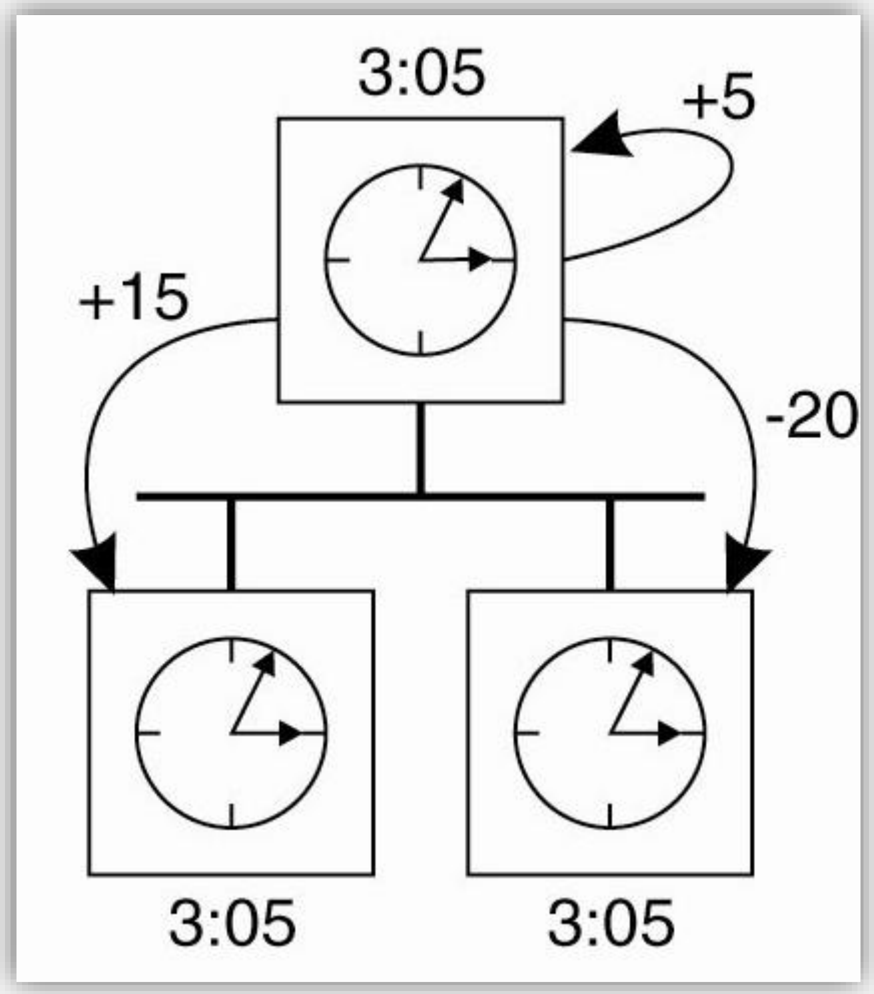
The Berkeley Algorithm (2)

The machines answer



The Berkeley Algorithm (3)

The time daemon tells everyone how to adjust their clock.



Berkeley vs NTP

Feature	Berkeley Synchronization Algorithm	Network Time Protocol (NTP)
Architecture	Uses a **master-slave** model where one node (master) computes the average time and tells all others to adjust.	Uses a **hierarchical model** with multiple layers (stratum levels) where clocks synchronize with more accurate sources (e.g., atomic clocks, GPS).
Time Source	No external reference (like GPS or atomic clock); instead, it takes the **average** of all participating clocks.	Synchronizes with external **authoritative** time sources (NTP servers) to provide a more **accurate** and reliable time.
Clock Adjustment	The master calculates a new time based on the average and instructs slaves to adjust accordingly.	Each node adjusts its clock based on time updates received from **higher-stratum** NTP servers.
Accuracy	Less accurate since it relies on an average of system clocks, which may all have drift.	Highly accurate, as it adjusts time based on reference clocks
Fault Tolerance	**Single point of failure**: If the master node fails, synchronization stops.	**Highly fault-tolerant**: If one NTP server fails, clients can switch to another.
Use Case	Best suited for **local area networks (LANs)** where external time sources are unavailable.	Used for **global time synchronization** across the internet, ensuring consistency across different locations.

Lamport’s Logical Clocks

- Clock synchronization is naturally related to time, although it may not be necessary to have an accurate account of the real time.
- It may be sufficient that every node in a distributed systems agrees on a current time.
- *For these algorithms, it is conventional to speak of the clocks as logical clocks.*

Lamport's Logical Clocks

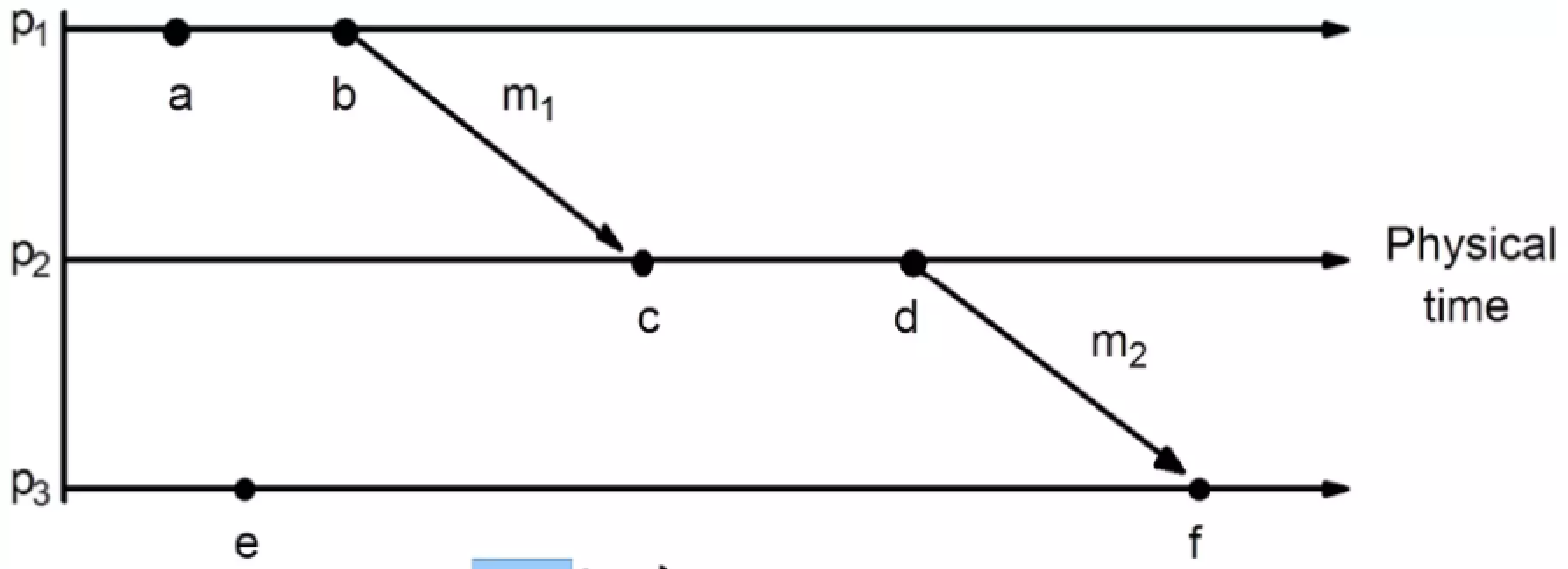
- Lamport pointed out that what usually matters is not that all processes agree on exactly what time it is, but rather that they agree on the order in which events occur.
- The "**happens-before" relation** $\rightarrow$ can be observed directly in two situations:
  - If *a* and *b* are events in the same process, and *a* occurs before *b*, then
    - ***a*** $\rightarrow$ ***b* is true.**
  - 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 true.**

Lamport's Logical Clocks

- Happens-before is a **transitive** relation
  - If **a → b and b → c, then a → c**

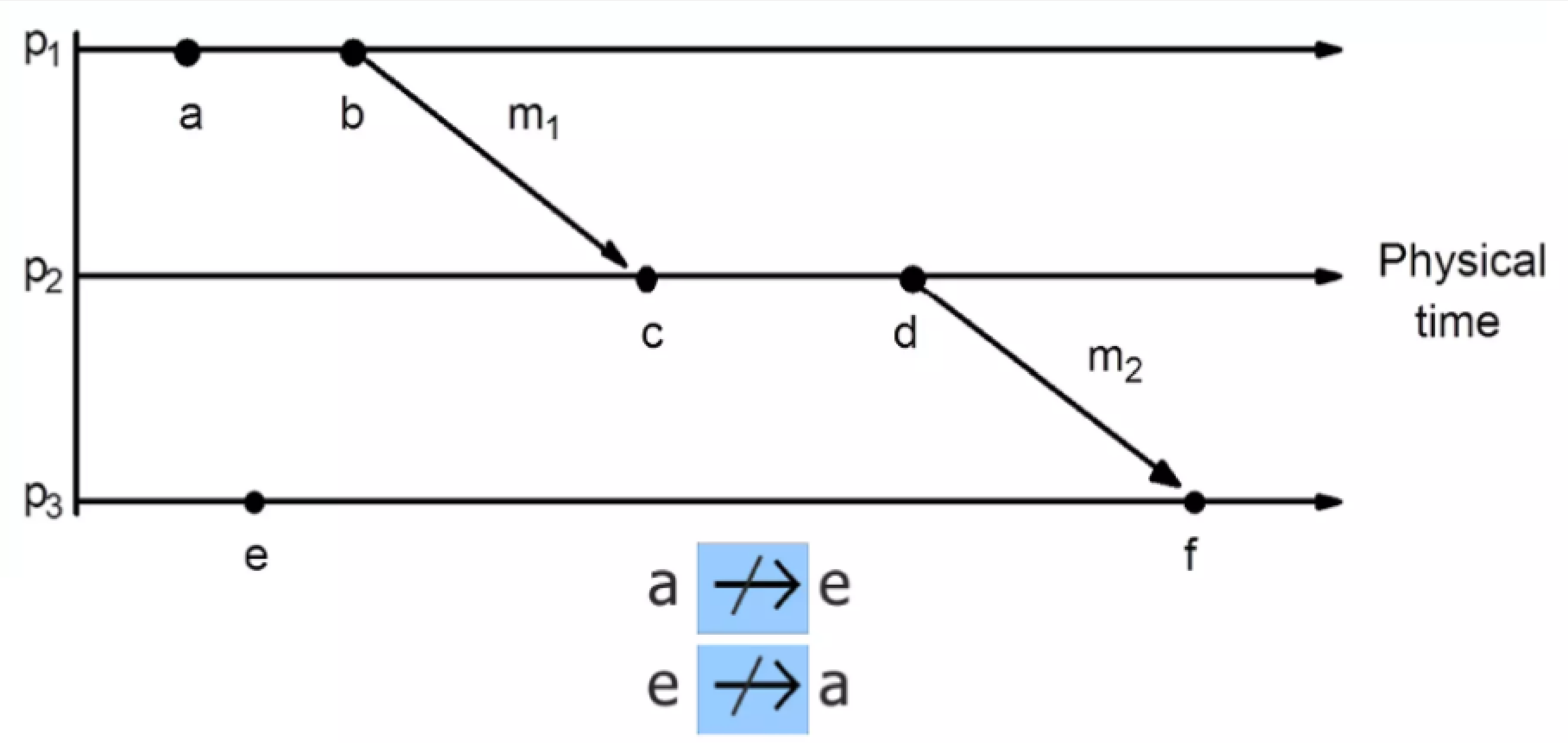
- If two events, x and y, happen in different processes that **do not exchange messages** (not even indirectly via third parties), then x → y is not true, but neither is y → x.
  - These events are said to be **concurrent**, which simply means that nothing can be said (or need be said) about when the events happened, or which event happened first.

Happen Before Relation



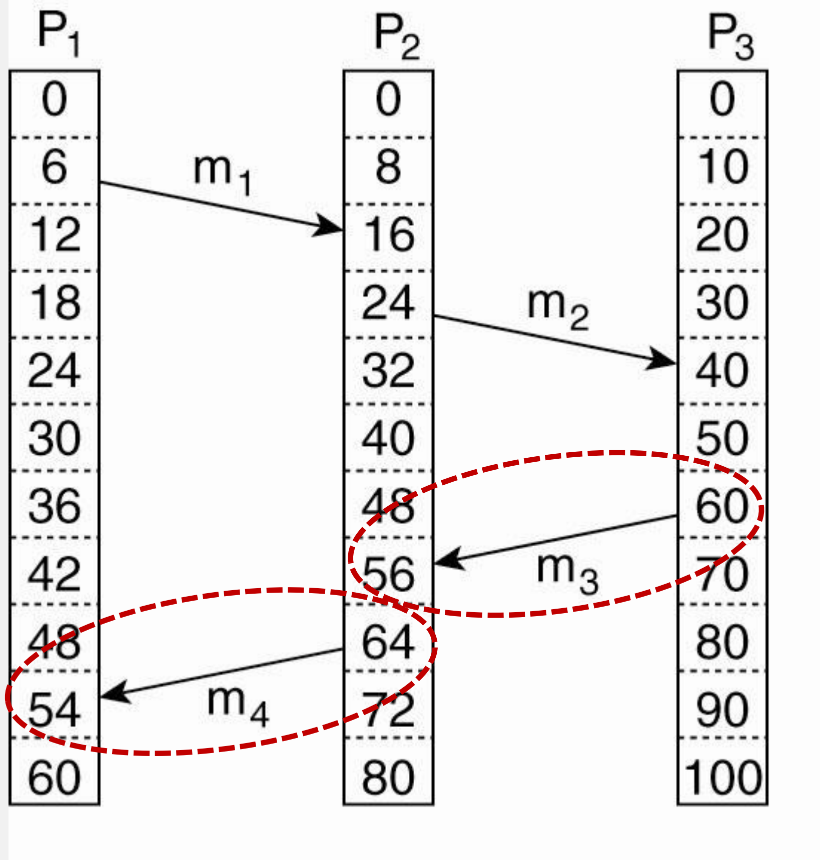
$$\left.\begin{array}{l} a \rightarrow b \\ b \rightarrow c \\ c \rightarrow d \\ d \rightarrow f \end{array}\right\} \Rightarrow a \rightarrow f$$

Happen Before Relation



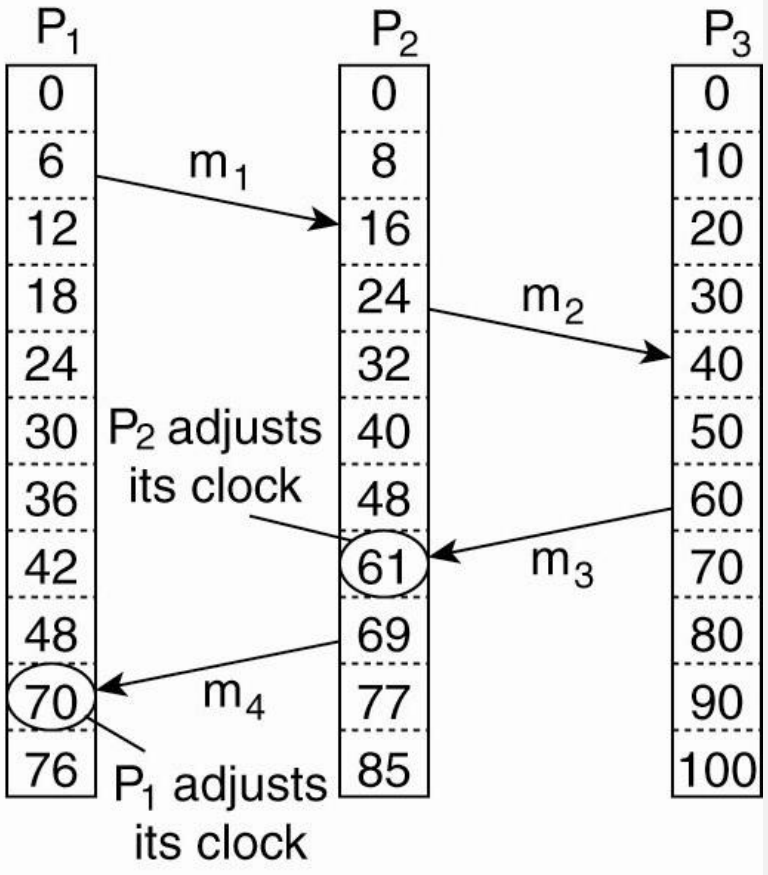
Lamport's Logical Clocks – Example

- **Three processes, each with its own clock, running at different rates.**
- The clock in process P1 is incremented by 6 units, 8 units in process P2, and 10 units in process P3, respectively.
- Message m1 from P1 to P2 takes 10 ticks, which is a plausible value and can be considered true.
- Message m2 from P2 to P3 takes 16 ticks, again a plausible value.
- However, m3 left at 60, must arrive at 61 or later but had arrived at 56 which cannot be considered true.



Lamport's Logical Clocks – Example

- Lamport's algorithm corrects the clocks as follow.
- Each message carries the sending time according to the sender's clock.
- When a message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time.
- In the figure we see that m3 now arrives at 61. Similarly, m4 arrives at 70 after the adjustments.



(b)

Lamport's Algorithm

- Updating local counter C_i for process P_i

1. Before executing an event P_i executes $C_i \leftarrow C_i + 1$.

2. When process P_i sends a message m to P_j, it sets *m*'s timestamp *ts (m)* equal to C_i after having executed the previous step.

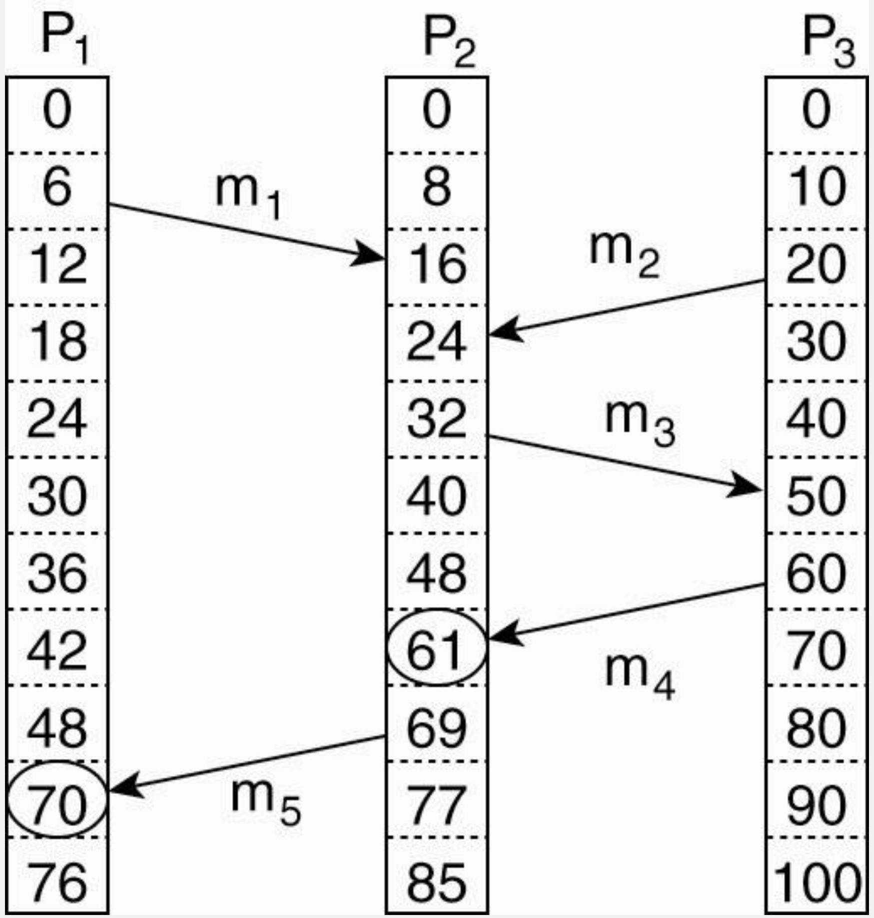
3. Upon the receipt of a message *m*, process P_j adjusts its own local counter as $C_j \leftarrow \max\{C_j , ts\ (m)\}$, after which it then executes the first step and delivers the message to the application.

Problems with Lamport Clocks

- Lamport timestamps do not capture **causality**.
- A Lamport clock may be used to create a **partial ordering of events**. For example,

$$\text{if } a \rightarrow b \text{ then } C(a) < C(b)$$

- if one event comes before another, then that event's logical clock comes before the other's.
- ***m3*** was indeed sent after the receipt of message ***m1***.



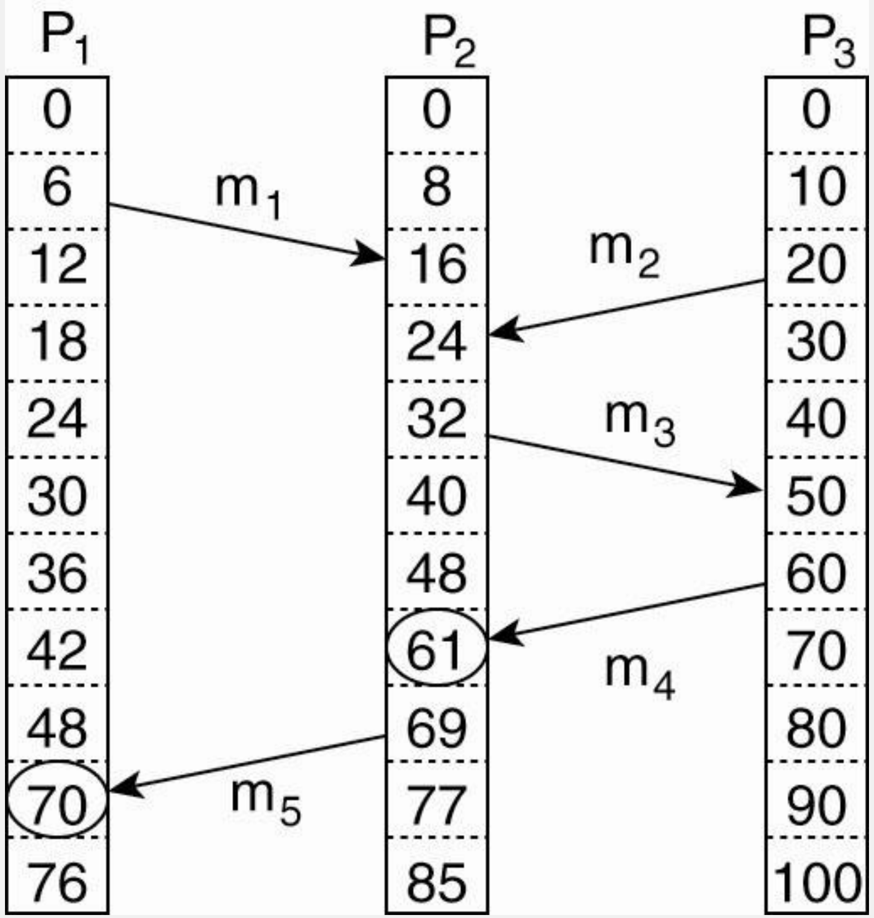
Problems with Lamport Clocks

- A Lamport clock, however, cannot establish **the strong clock consistency condition**, which is two-way condition

$$\text{if } C(a) < C(b) \text{ then } a \rightarrow b$$

- This does not necessarily imply that ***a*** indeed happened before ***b***.
- For example, the sending of ***m2*** (20) is greater than the receipt of **m1** (16), but we cannot establish a causality between the two events using Lamport clock.
- However, it's true that

$$C(a) \not< C(b) \text{ implies } a \not\rightarrow b$$



Vector Timestamp

- The main problem is that a **simple integer clock** cannot order both events within a process and events in different processes.

- Fidge developed an algorithm that overcomes this problem.

- Fidge's clock is represented as a vector $[v_1,v_2,\ldots,v_n]$ with an integer clock value for each process (v_i contains the clock value of process *i*).

- This is a **vector timestamp**.

Vector Clocks (2)

- Vector clocks 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. It is thus P_i's knowledge of the local time at P_j .

Vector Clocks (3)

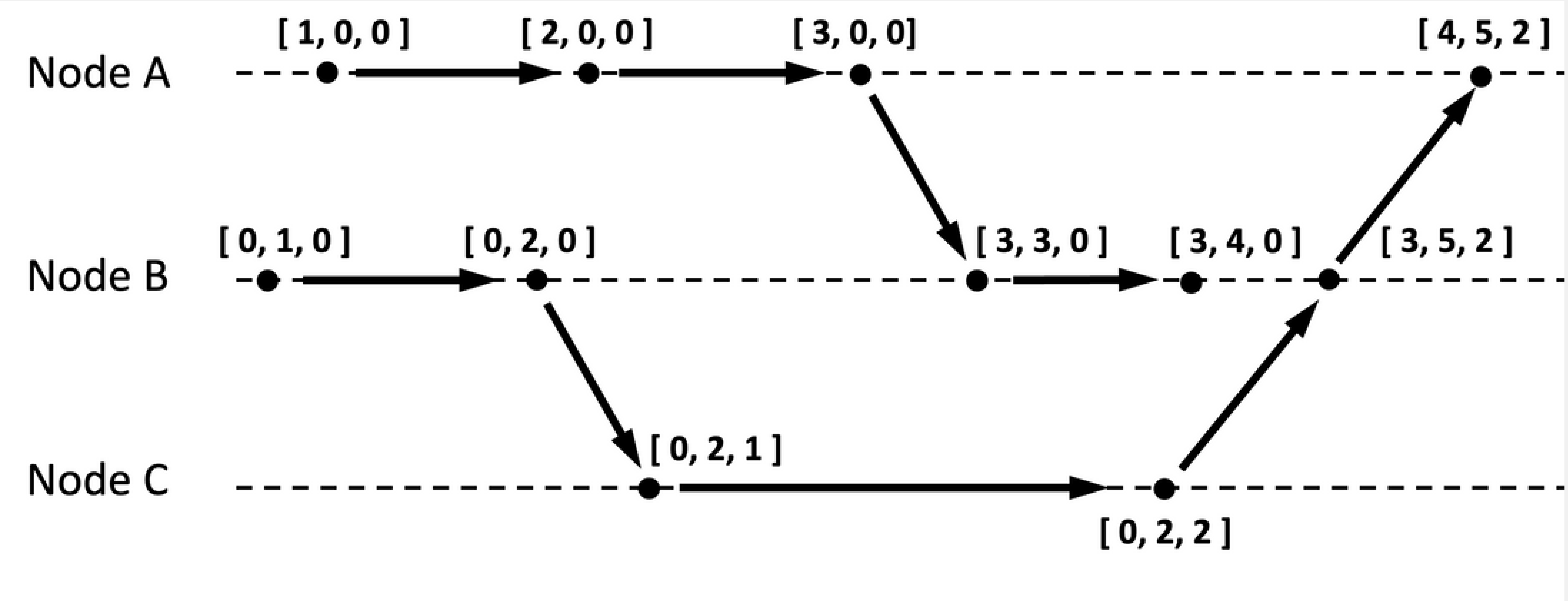
Steps carried out to accomplish property 2 of previous slide:

1. Before executing an event P_i executes $VC_i[\,i\,] \leftarrow VC_i[i\,] + 1$.

2. When process P_i sends a message **m** to P_j, it sets *m*’s (vector) timestamp *ts (m)* equal to VC_i.

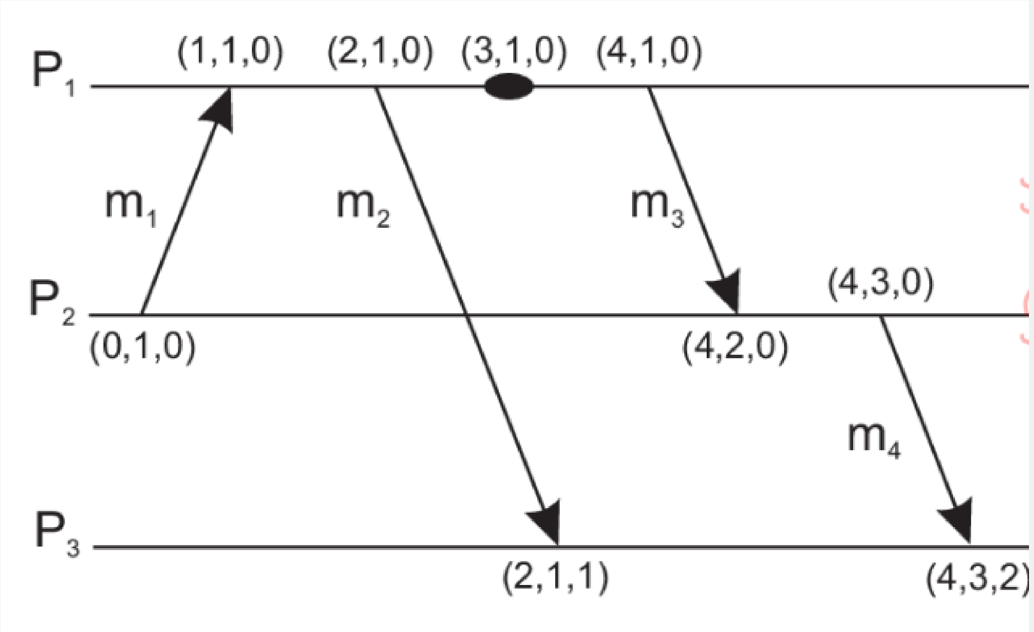
3. Upon the receipt of a message m, process P_j adjusts its own vector by setting
$VC_j[k\,] \leftarrow \max\{VC_j[k\,], ts(m)[k\,]\}$ for each *k*, after which it executes the first step and delivers the message to the application.

Example



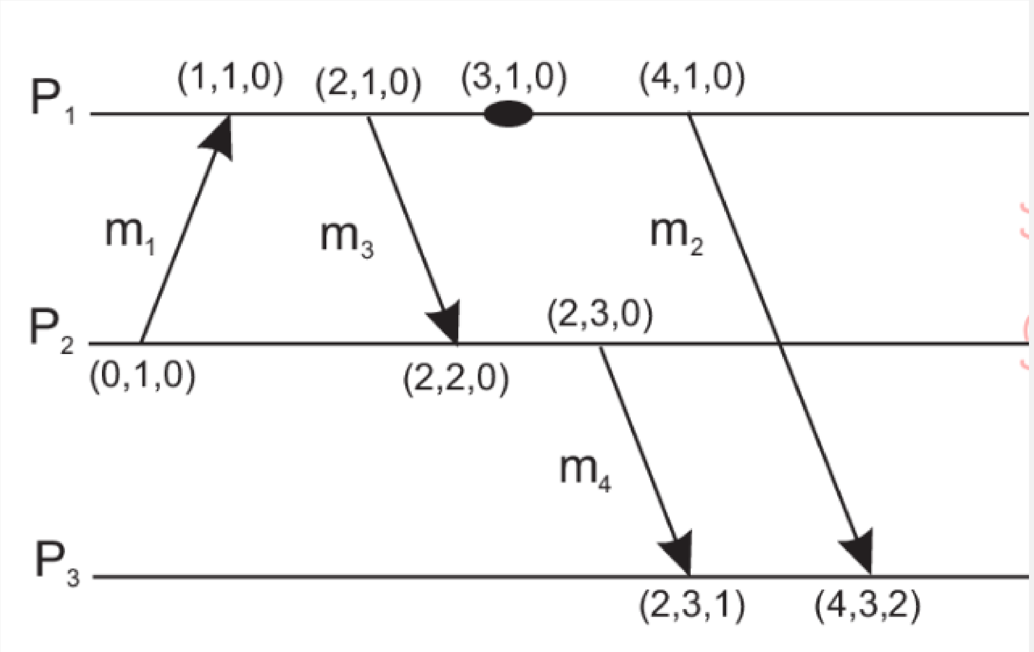
Example

- P2 sends a message m1 at logical time VC2 = (0, 1, 0) to process P1.
- Message m1 thus receives timestamp ts(m1) = (0, 1, 0). Upon its receipt, P1 adjusts its logical time to VC1 ← (1, 1, 0) and delivers it.
- Message m2 is sent by P1 to P3 with timestamp ts(m2) = (2, 1, 0). Before P1 sends another message, m3, an event happens at P1, eventually leading to timestamping m3 with value (4, 1, 0).
- After receiving m3, process P2 sends message m4 to P3, with timestamp ts(m4) = (4, 3, 0).



Example

- Here, we have delayed sending message m2 until after message m3 has been sent, and after the event had taken place.
- It is not difficult to see that ts(m2) = (4, 1, 0), while ts(m4) = (2, 3, 0).



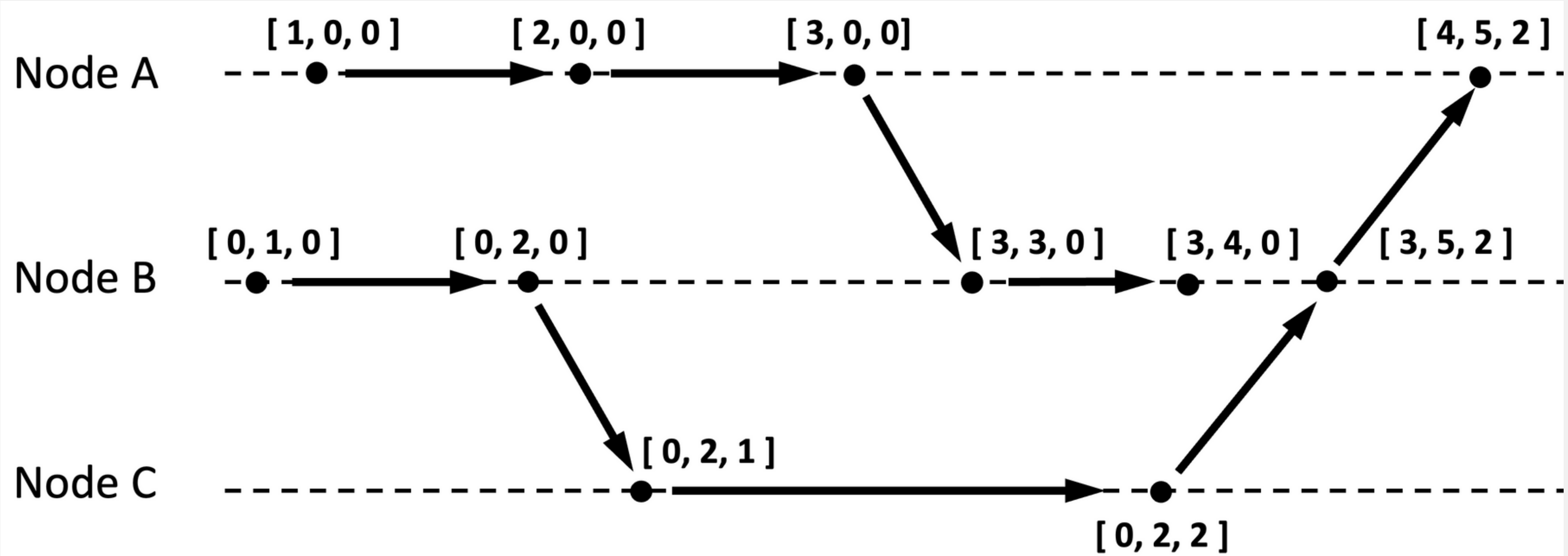
Example

- Comparing we have the following situation

Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	Conclusion
Figure 6.13(a)	$(2,1,0)$	$(4,3,0)$	Yes	No	m_2 may causally precede m_4
Figure 6.13(b)	$(4,1,0)$	$(2,3,0)$	No	No	m_2 and m_4 may conflict

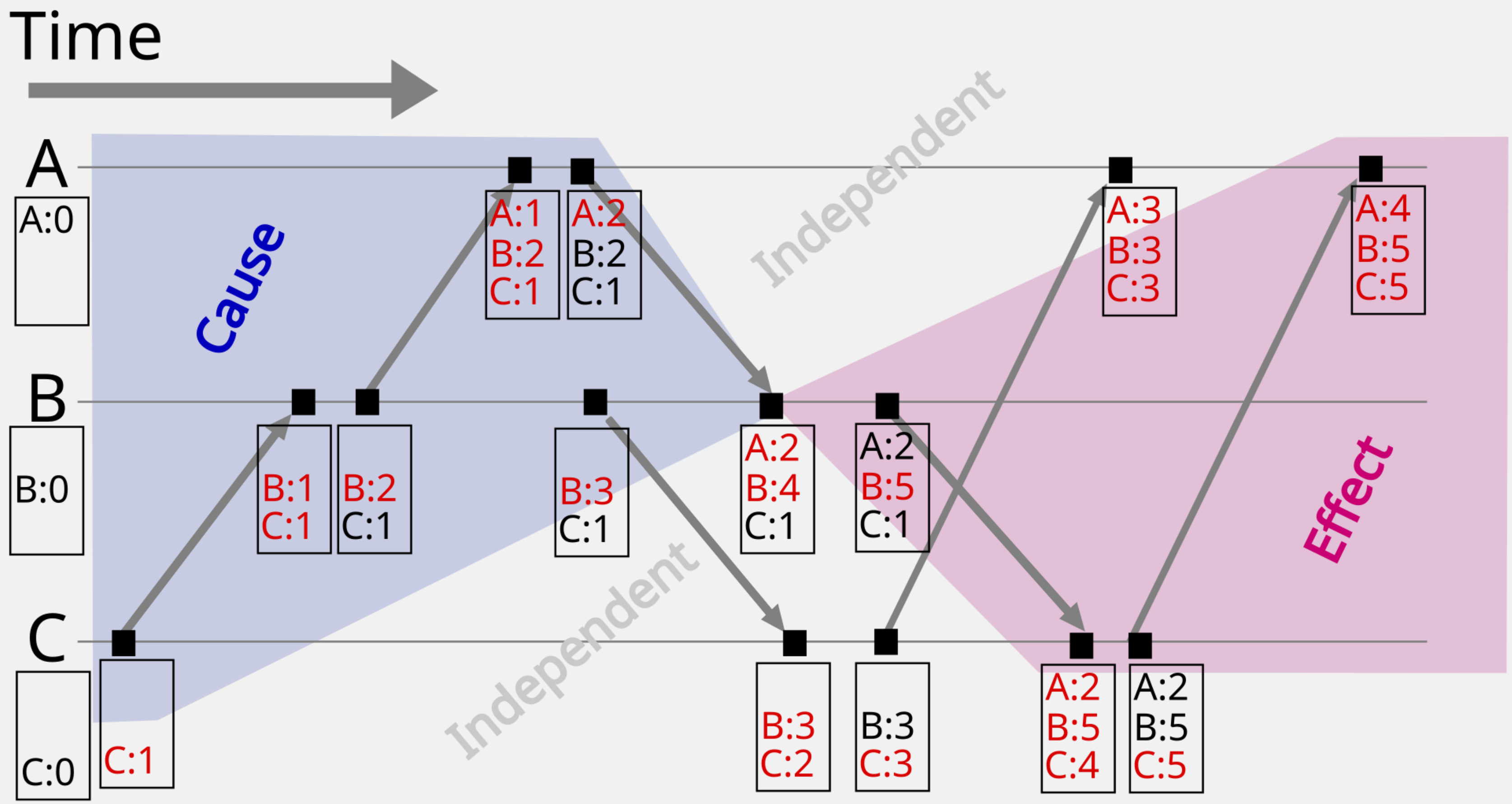
- Compare each entry from one n-tuple timestamp with the corresponding entry in another n-tuple timestamp.
- We use the notation ts(a) < ts(b) if and only if for all k, ts(a)[k] ≤ ts(b)[k] and there is at least one index k' for which ts(a)[k'] < ts(b)[k'].
- Thus, by using vector clocks, process P3 can detect whether m4 may be causally dependent on m2, or whether there may be a **potential conflict (events are concurrent)**.

Comparing two Events



- If some entries are less or equal, and some entries are greater, the timestamps are **concurrent**, e.g., [**3**, **4**, 0] and [0, 2, **2**]. The greater entries are in bold.
- If one or more entries are less and none are greater, the timestamp with lower entry values precedes the other timestamp, e.g., [3, 4, 2] precedes [**4**, **5**, 2].

Cause and Effects



Events in the blue region are the causes leading to event B4, whereas those in the red region are the effects of event B4.