Al Sham Private University
Faculty of Informatics Engineering

جامعة الشام الخاصة
كلية الهندسة المعلوماتية

نظم معلومات موزعة
Distributed Information Systems

Lecture 7: Synchronization (Physical and Logical Time)

اعداد: أ. غاندي هسام

Introduction

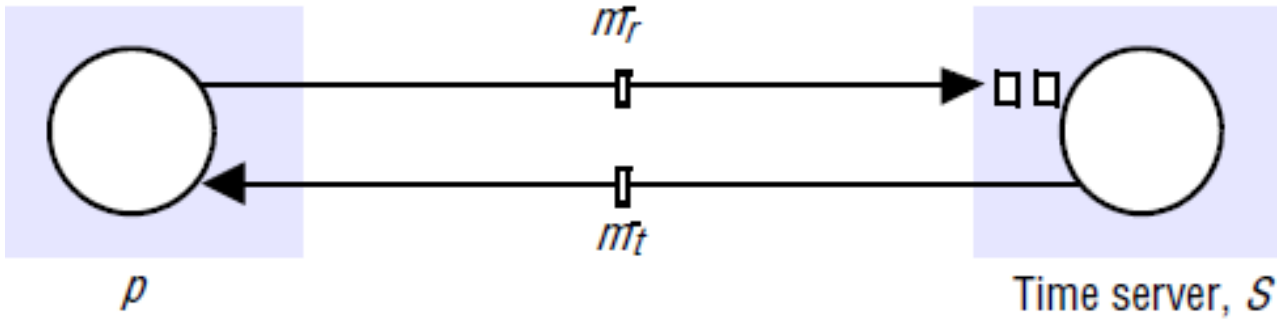
- Time is an important and interesting issue in distributed systems, for several reasons.
- First, time is a quantity we often want to measure accurately. In order to know at what time of day a particular event occurred at a particular computer it is necessary to synchronize its clock with an authoritative, external source of time.
- Second, algorithms that depend upon clock synchronization have been developed for several problems in distribution. These include maintaining the consistency of distributed data, checking the authenticity of a request sent to a server and eliminating the processing of duplicate updates.

- The relative order of two events can even be reversed for two different observers. But this cannot happen if one event causes the other to occur.
- In that case, the physical effect follows the physical cause for all observers, although the time elapsed between cause and effect can vary.
- The concept of physical time is also problematic in a distributed system.
- The problem is based on a similar limitation in our ability to timestamp events at different nodes sufficiently accurately to know the order in which any pair of events occurred, or whether they occurred simultaneously.
- There is no absolute, global time to which we can appeal.

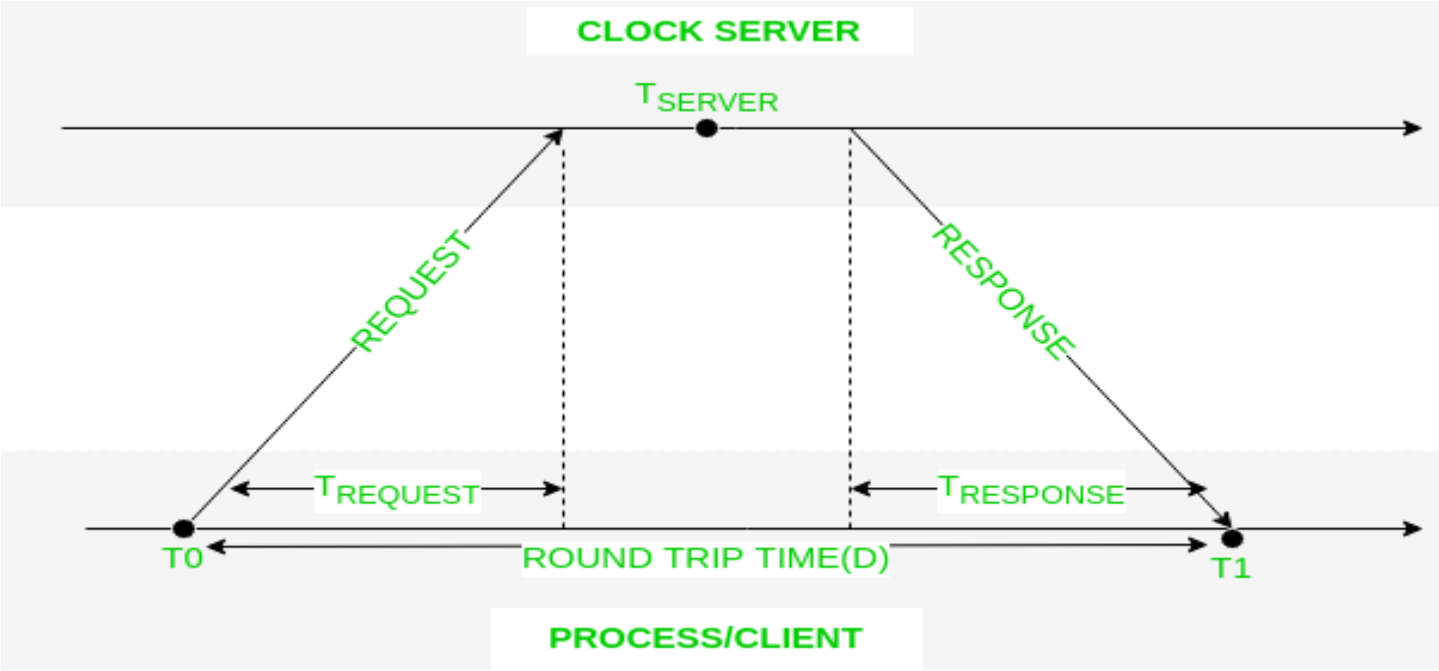
Synchronizing physical clocks

❑ Cristian's method:

- *Coordinated Universal Time* – abbreviated as UTC (from the French equivalent) – is an international standard for timekeeping.
- UTC signals are synchronized and broadcast regularly from land-based radio stations and satellites (*Global Positioning System* (GPS)) covering many parts of the world.
- Cristian [1989] suggested the use of a time server, connected to a device that receives signals from a source of UTC, to synchronize computers externally.
- Upon request, the server process S supplies the time according to its clock



- A process p requests the time in a message *mr* , and receives the time value *t* in a message *mt* (t is inserted in mt at the last possible point before transmission from S's computer).
- Process p records the total round-trip time Tround taken to send the request mr and receive the reply mt .
- A simple estimate of the time to which p should set its clock is *t* + Tround/2.



1) The process on the client machine sends the request for fetching clock time(time at server) to the Clock Server at time T0.

2) The Clock Server listens to the request made by the client process and returns the response in form of clock server time.

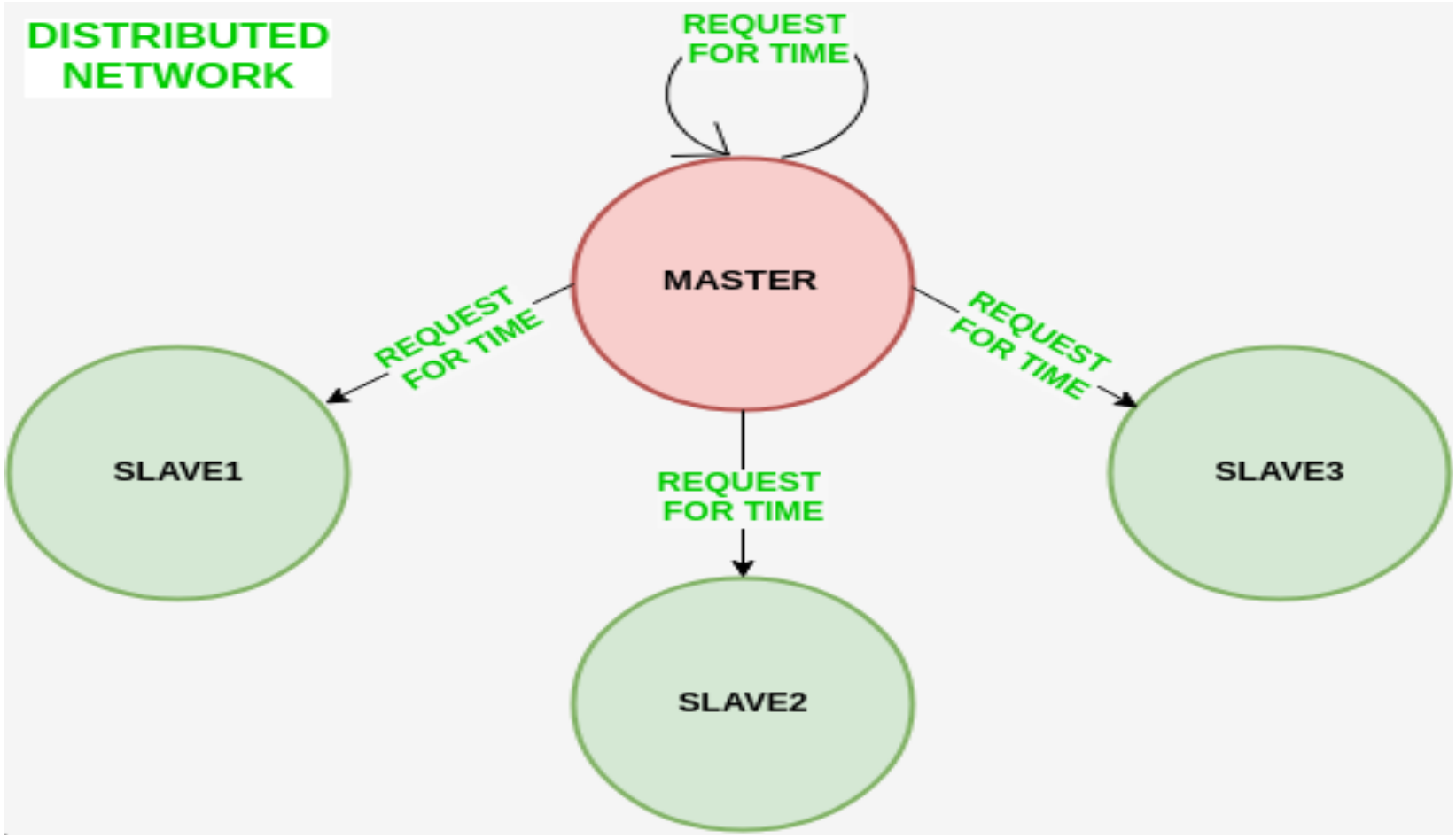
3) The client process fetches the response from the Clock Server at time T1 and calculates the synchronized client clock time using the formula given below.

$$T_{CLIENT} = T_{SERVER} + (T_1 - T_0)/2$$

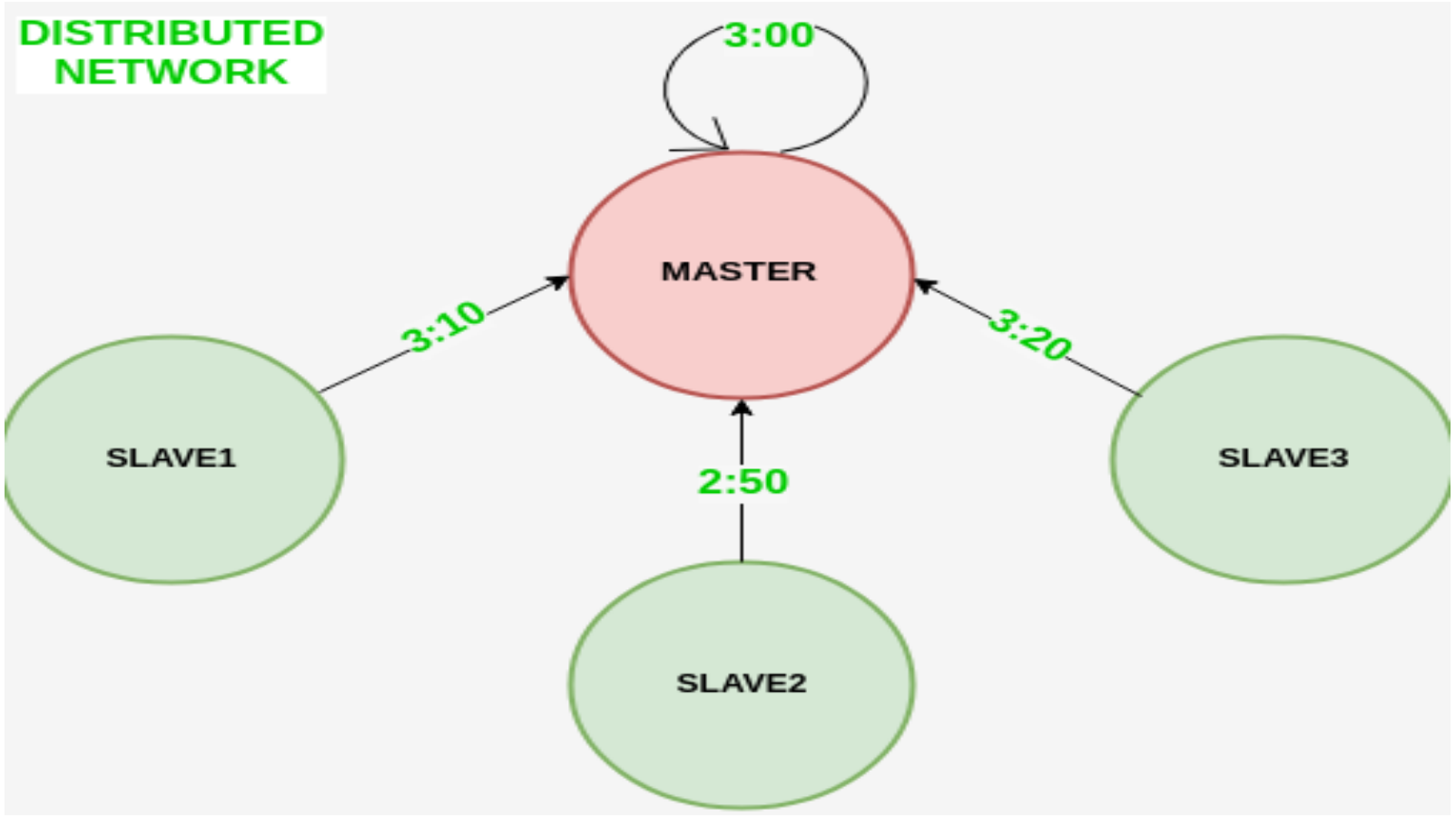
❑ The Berkeley algorithm:

- A coordinator computer is chosen to act as the *master*. Unlike in Cristian's protocol, this computer periodically polls the other computers whose clocks are to be synchronized, called *slaves*.
- The slaves send back their clock values to it. The master estimates their local clock times by observing the round-trip times and it averages the values obtained.
- The accuracy of the protocol depends upon a nominal maximum round-trip time between the master and the slaves.
- The master sends the amount by which each individual slave's clock requires adjustment. This can be a positive or negative value.

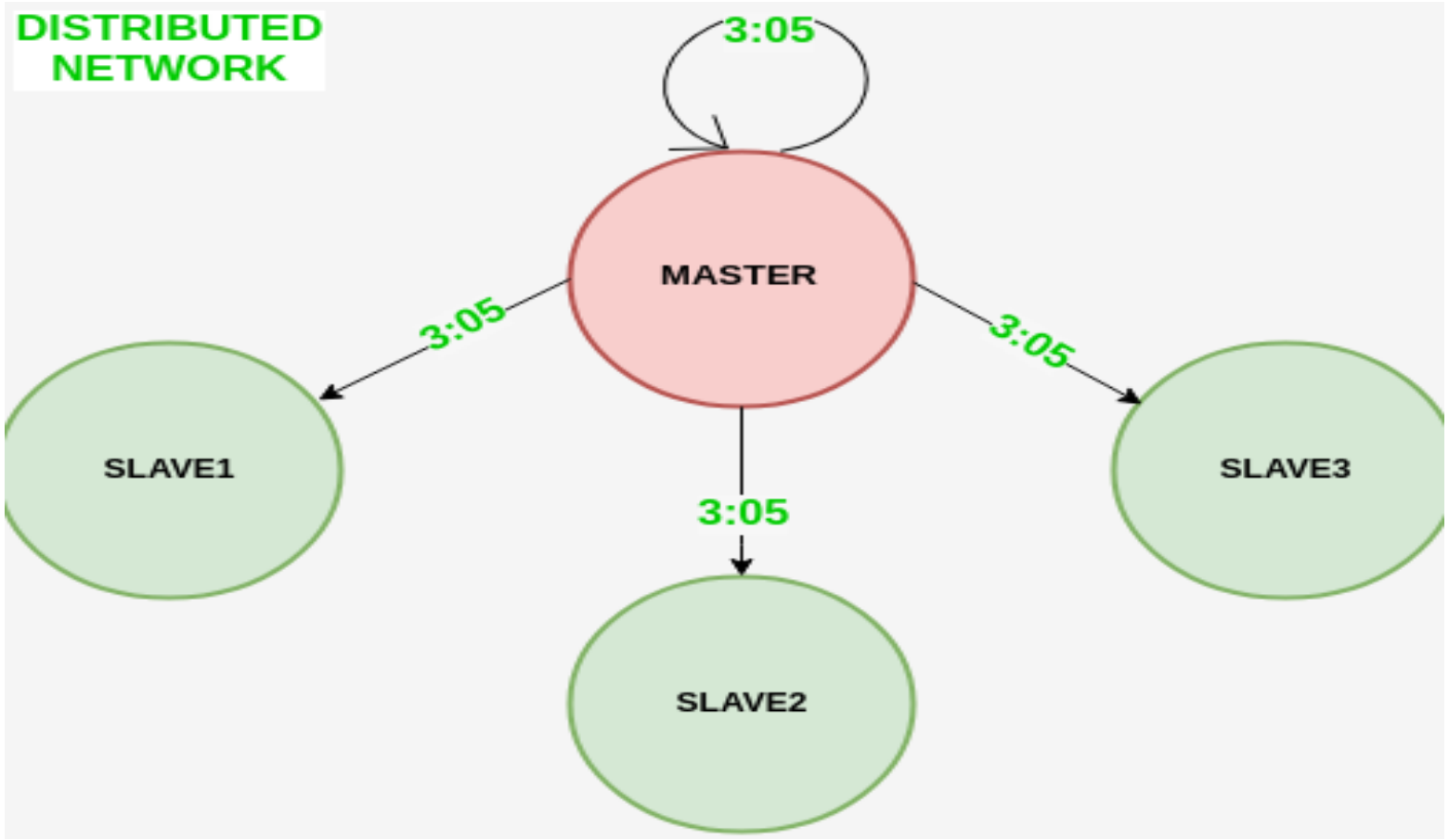
- Diagram below illustrates how the master sends request to slave nodes:



- Diagram below illustrates how slave nodes send back time given by their system clock:



- Diagram below illustrates the last step of Berkeley's algorithm:



Logical time and logical clocks

- **Lamport** [1978] pointed out, since we cannot synchronize clocks perfectly across a distributed system, we cannot in general use physical time to find out the order of any arbitrary pair of events occurring within it.
- In general, we can use a scheme that is similar to physical causality but that applies in distributed systems to order some of the events that occur at different processes.
- Lamport called the partial ordering obtained by generalizing these two relationships the *happened-before* relation. It is also sometimes known as the relation of *causal ordering* or *potential causal ordering*.

❑ Logical clocks (Lamport):

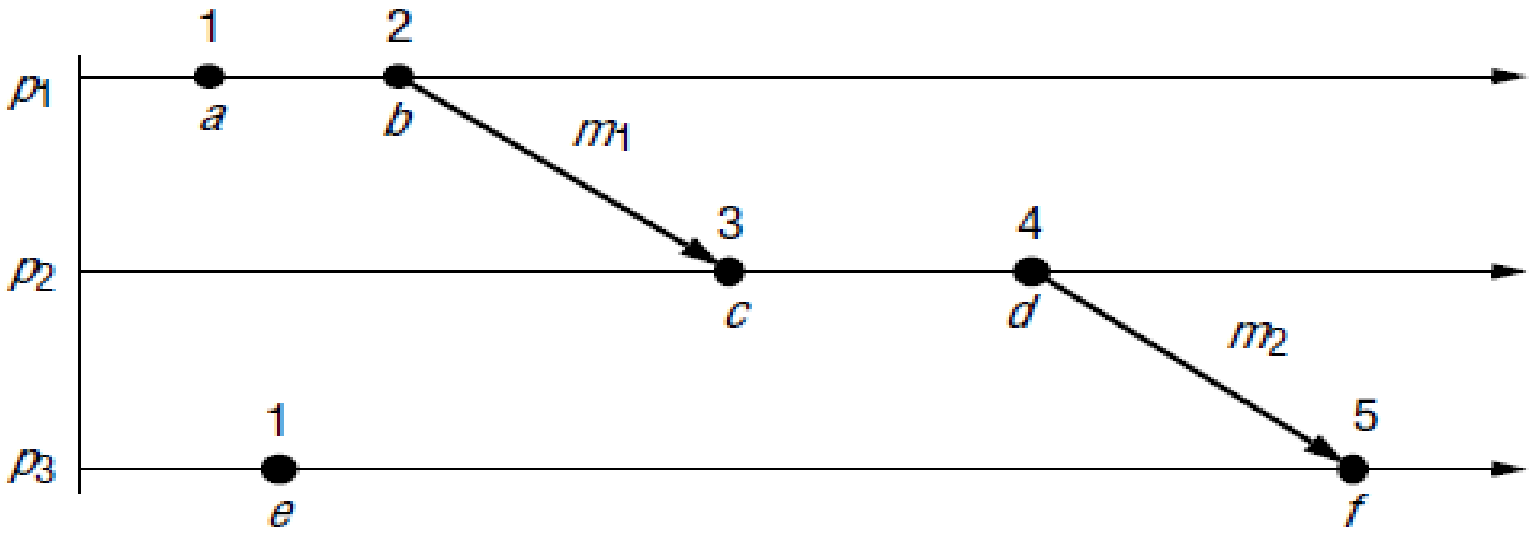
- A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock.
- Each process p_i keeps its own logical clock, L_i , which it uses to apply so-called Lamport timestamps to events.
- We denote the timestamp of event e at p_i by $L_i(e)$, and by L(e) we denote the timestamp of event e at whatever process it occurred at.
- To capture the happened-before relation →, processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

LC1: L_i is incremented before each event is issued at process p_i: $L_i := L_i + 1$.

LC2: (a) When a process p_i sends a message m, it piggybacks on m the value $t = L_i$.

(b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive*(m).

- It can easily be shown, by induction on the length of any sequence of events relating two events e and e', that $e \rightarrow e' \Rightarrow L(e) < L(e')$.
- Note that the converse is not true. If $L(e) < L(e')$ then we cannot infer that $e \rightarrow e'$.



- Each of the processes p1 , p2 and p3 has its logical clock initialized to 0.
- The clock values given are those immediately after the event to which they are adjacent.
- Note that, for example, $L(b) > L(e)$ but $b \parallel e$.

❑ Vector clocks:

- to overcome the shortcoming of Lamport's clocks: the fact that from $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$.
- A vector clock for a system of N processes is an array of N integers.
- Each process keeps its own vector clock, Vi , which it uses to timestamp local events.
- there are simple rules for updating the clocks:

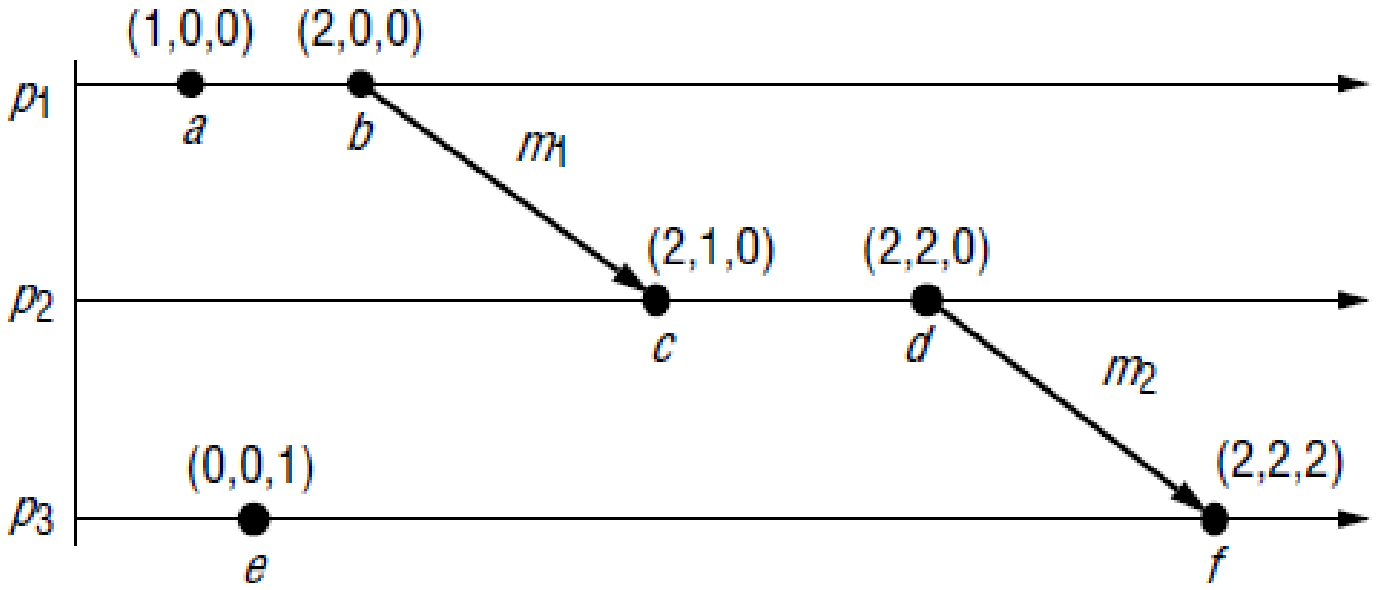
VC1: Initially, $V_i[j] = 0$, for $i, j = 1, 2 \ldots, N$.

VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$.

VC3: p_i includes the value $t = V_i$ in every message it sends.

VC4: When p_i receives a timestamp t in a message, it sets $V_i[j] := max(V_i[j], t[j])$, for $j = 1, 2\ldots, N$. Taking the component-wise maximum of two vector timestamps in this way is known as a *merge* operation.

- Example #1:



- We may compare vector timestamps as follows:

$$V = V' \;\; iff \;\; V[j] = V'[j] \;\; \text{for } j = 1, 2 \ldots, N$$

$$V \leq V' \;\; iff \;\; V[j] \leq V'[j] \;\; \text{for } j = 1, 2 \ldots, N$$

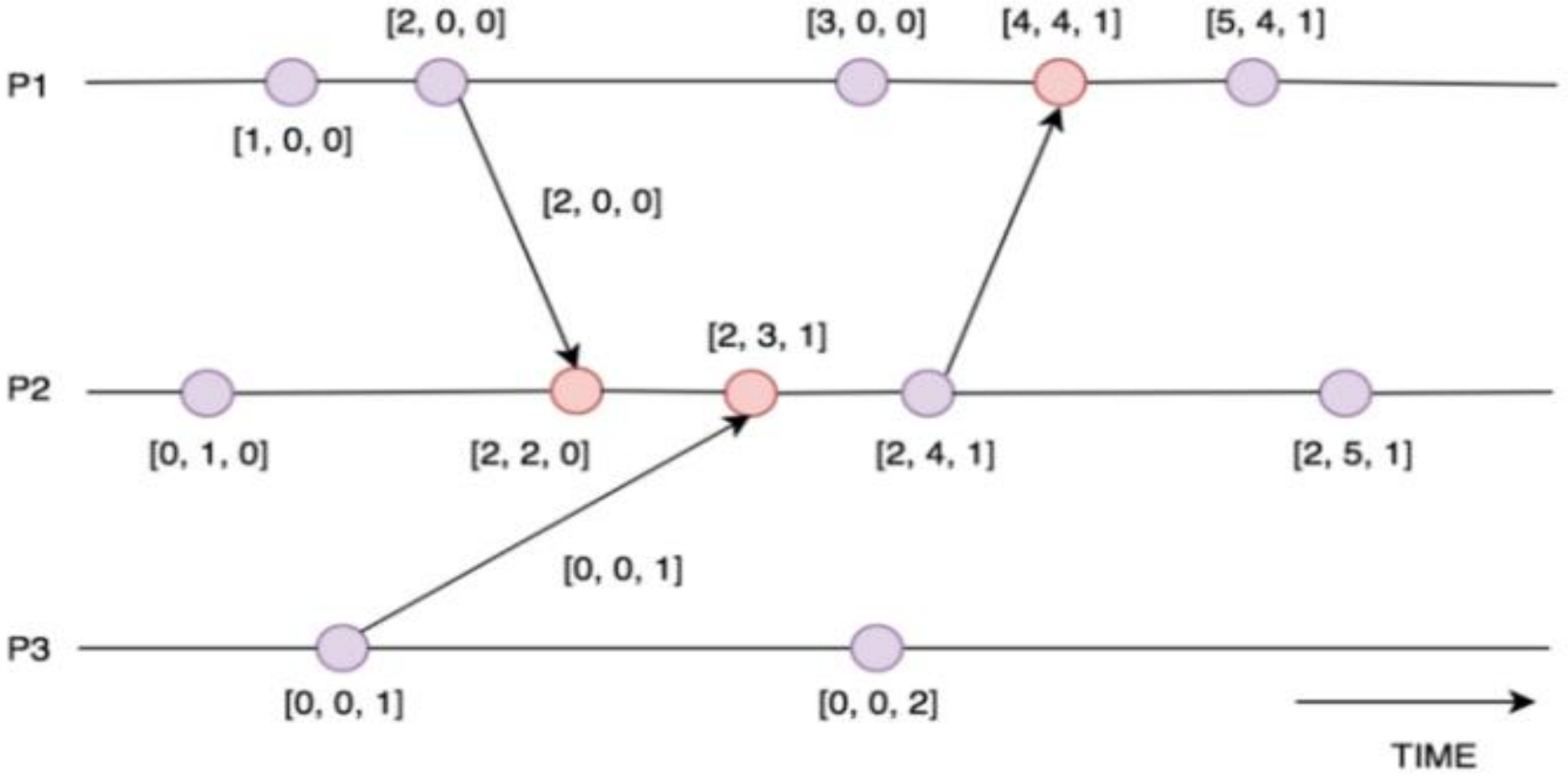
$$V < V' \;\; iff \;\; V \leq V' \wedge V \neq V'$$

$$', \text{ that } e \rightarrow e' \Rightarrow V(e) < V(e')$$
$$\text{if } V(e) < V(e'), \text{ then } e \rightarrow e' .$$

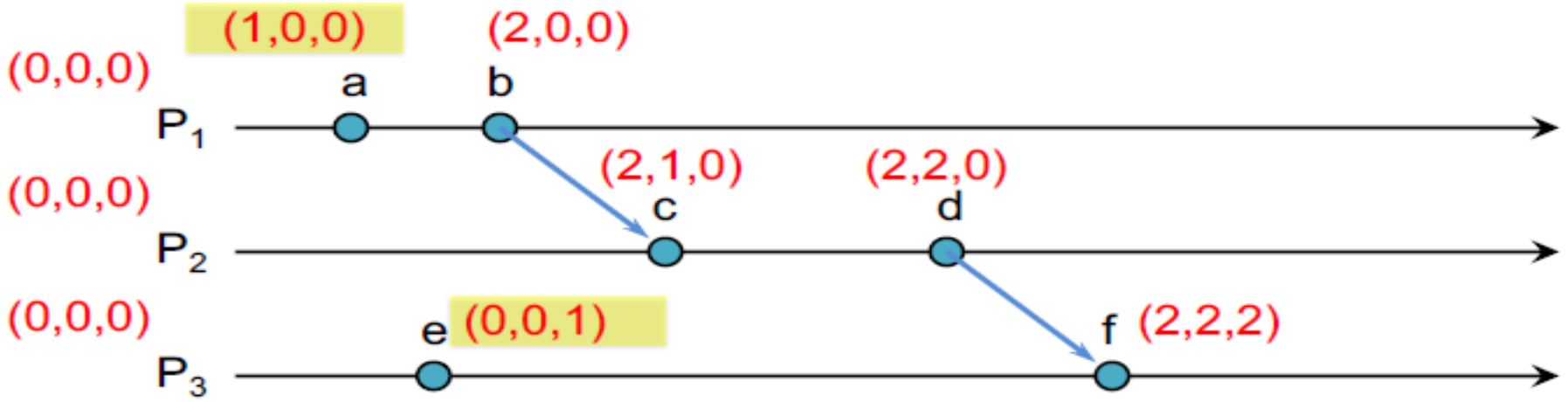
- In our example: We can tell when two events are concurrent by comparing their timestamps.

$c \parallel e$ can be seen from the facts that neither $V(c) \leq V(e)$ nor $V(e) \leq V(c)$.

- Example #2:


[2, 0, 0]
[3, 0, 0]
[4, 4, 1]
[5, 4, 1]
P1
[1, 0, 0]
[2, 0, 0]
[2, 3, 1]
P2
[0, 1, 0]
[2, 2, 0]
[2, 4, 1]
[2, 5, 1]
[0, 0, 1]
P3
[0, 0, 1]
[0, 0, 2]
TIME

- Example #3:



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

End of Lecture 7