UNIVERSIDAD DE CASTILLA-LA MANCHA

Sistemas Distribuidos

Time
and
Global States

Contents

Time in distributed systems

- Time is a quantity we need to measure accurately:
  - Example: eCommerce transactions
- But measuring time is complex in distributed systems:
  - Parallelism between processors
  - Arbitrary processor speeds
  - No determinism in the delay of the messages.
  - Absence of global time

Objectives

To be able to timestamp events reliably to know an order of events

- Examples: transactions, ensure consistency of files.

To be able to analyze global states of the system to infer some relevant properties:

- Examples: garbage collection, termination, deadlock.

System model (1/2)

- A distributed system consists of a collection of n processes:
  - $p_i = \{p_1, p_2, \ldots, p_n\}$
  - Each process executes on a single processor,
  - The processors do not share memory,
  - The processes only can communicate through message passing
- Each process has a state s_i that **holds the variables** within it:
  - The execution of the process p_i transforms its state s_i
- An event e is the occurrence of a single action on a process:
  - Communication (send/receive)
  - State transformation

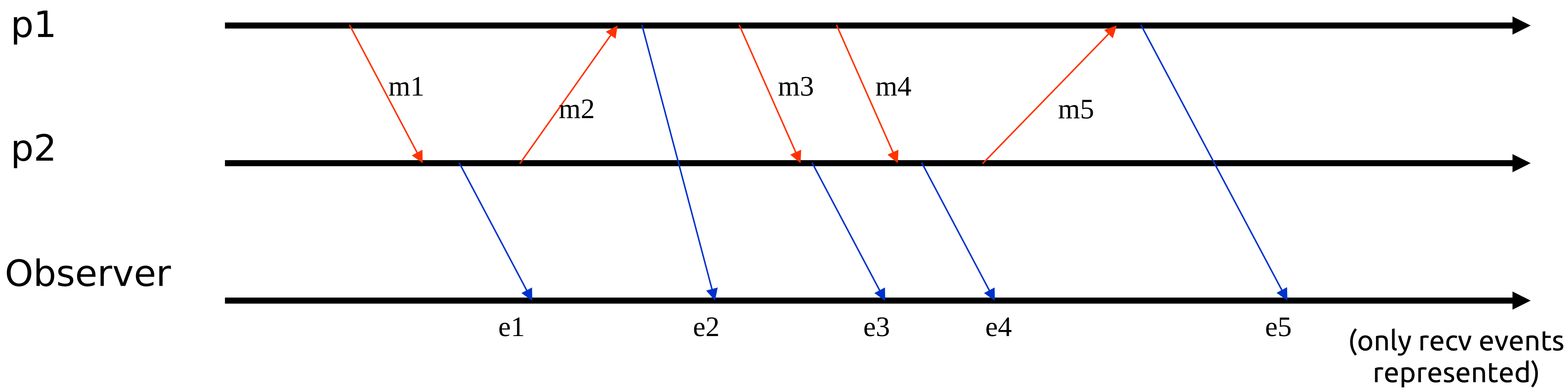
System model (2/2)

- The relation ($\rightarrow_i$) establishes a single, total order of events within a single process p_i
  - $e \rightarrow_i e'$ iff *the event e is previous to e' at* p_i

- The **history** of p_i, denoted as h_i, is the series of events that take place within it:
  - History(p_i)= h_i = $<e_i^0, e_i^1, e_i^2, \ldots>$,
  - where $e_i^k \rightarrow e_i^{k+1}$ for all events in p_i

Timestamps

- A timestamp is the date and time of day when the event happened.
- Each event in the process is assigned to a unique timestamp.
- Example:
  - p1 and p2 are processes communicating via send/receive primitives
  - Another process (observer) must be able to observe the same order in which the events happened
  - $e_i^x \rightarrow e_j^y$ iff $\text{Timestamp}(e_i^x) < \text{Timestamp}(e_i^y)$



How to define timestamps?

Physical clocks

- An electronic device (RTC) that counts oscillations occurring in a crystal at definite frequency
- It measures the real, physical time **t** for process $p_{i,}$ usually one for the whole computer

Logical clocks

- Logical time is an increasing monotone sequence of values
- Not related with real physical time

Physical clocks

To generate a timestamp for each instant of time **t** at process p_i:

- $H_i(t)$ is the hardware clock value
- $C_i(t)$ is the software clock, computed as $C_i(t) = a\,H_i(t) + b$, where:
  - a, b are values to scale $H_i(t)$
  - Example: 64-bit value of the number of nanoseconds that have elapsed at time *t* since a reference time

The clock resolution is the period between updates of the software clock value $C_i(t)$

- Two events e and e’, that happen at t and t’, respectively, will have different timestamps iff the resolution is smaller than t’-t.

Clock resolution

a)

b)

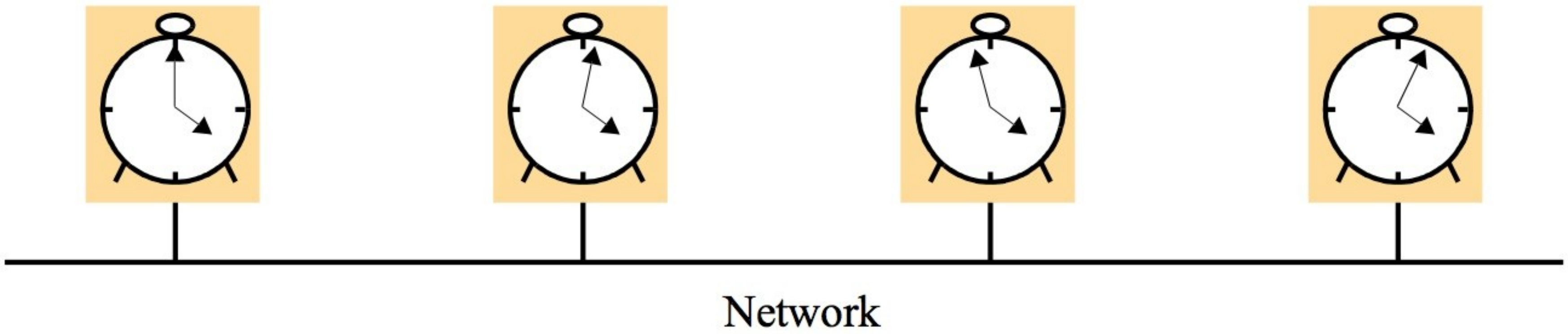
"a → b" or "b → a"?

Clock skew and clock drift

- The clock skew is the instantaneous difference between the readings of any two clocks at instant *t*

- The clock drift is the difference between frequencies of two clocks

- Two clocks on two different computers give different measurements



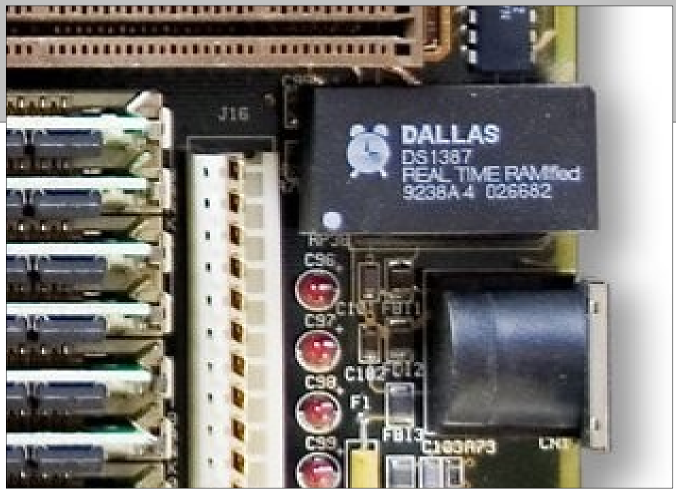
Need to synchronization

UNIVERSIDAD DE CASTILLA-LA MANCHA

RTC: Real Time Clock → $H_i(t)$

DIY

```
$ sudo /sbin/hwclock --test
hwclock from util-linux 2.33.1
System Time: 1605704670.053769
Trying to open: /dev/rtc0
Using the rtc interface to the clock.
Last drift adjustment done at 1605704666 seconds after 1969
Last calibration done at 1605704666 seconds after 1969
Hardware clock is on UTC time
Assuming hardware clock is kept in UTC time.
Waiting for clock tick...
...got clock tick
Time read from Hardware Clock: 2020/11/18 13:04:31
Hw clock time : 2020/11/18 13:04:31 = 1605704671 seconds since 1969
Time since last adjustment is 5 seconds
Calculated Hardware Clock drift is 0.000000 seconds
2020-11-18 14:04:30.045788+01:00
```



Wikipedia: RTC

Software clock: $C_i(t)$

DIY

```
$ timedatectl status
               Local time: lun 2019-11-18 12:54:14 CET
           Universal time: lun 2019-11-18 11:54:14 UTC
                 RTC time: lun 2019-11-18 11:54:14
                Time zone: Europe/Madrid (CET, +0100)
System clock synchronized: yes
              NTP service: inactive
          RTC in local TZ: no
```

Time standards

- Astronomic Time
  - Based on rotation Earth's rotational period
- International Atomic Time (TAI)
  - 1 second is 9 192 631 770 periods of transition between the two hyperfine levels of the ground state of Cs^{133} (1967)
  - It tends to separate from astronomic time
  - It's exactly 37 seconds ahead of UTC
- Coordinated Universal Time (UTC)
  - International standard that regulates clocks and time
  - Based on TAI with 'leap seconds' added/deleted at irregular intervals
  - UTC signals are broadcast regularly from land-based radio stations and satellites

Synchronizing physical clocks

External synchronization:

- $C_i(t)$ synchronizes with regard to an authoritative external source of time (e.g. UTC)
  - S is an external source of time
  - S(t) is the value returned by S for time t
  - C_i (t) ***is synchronized with S within the bound D*** if:

$$|S(t) - C_i(t)| < D$$

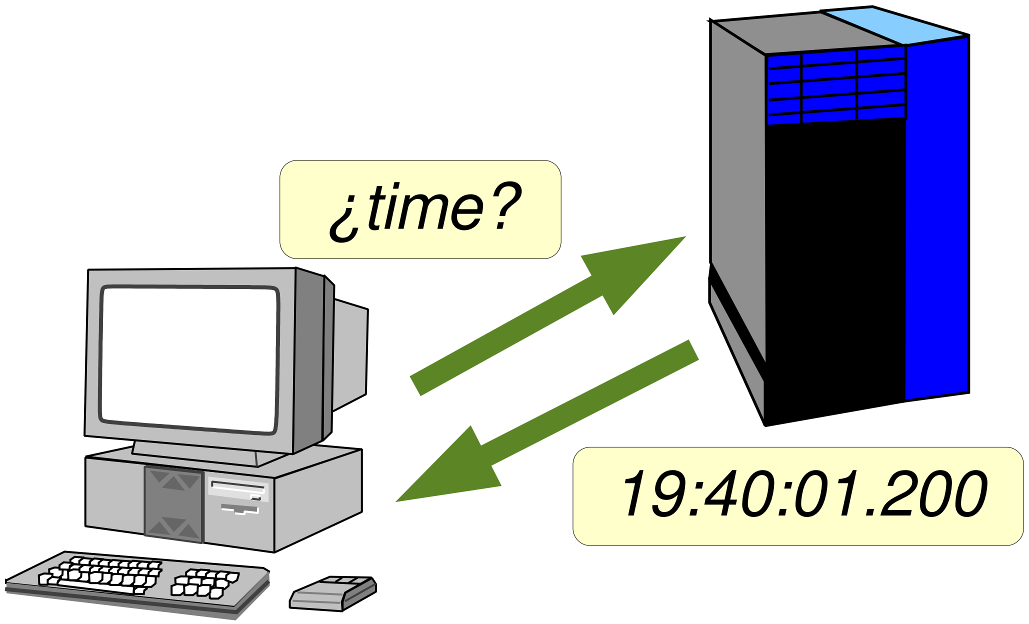
Internal synchronization:

- The difference between two clocks at two processes i and j, $C_i(t)$ and $C_j(t)$, respectively, is limited
  - C_i y C_j **are *synchronized within the bound D***

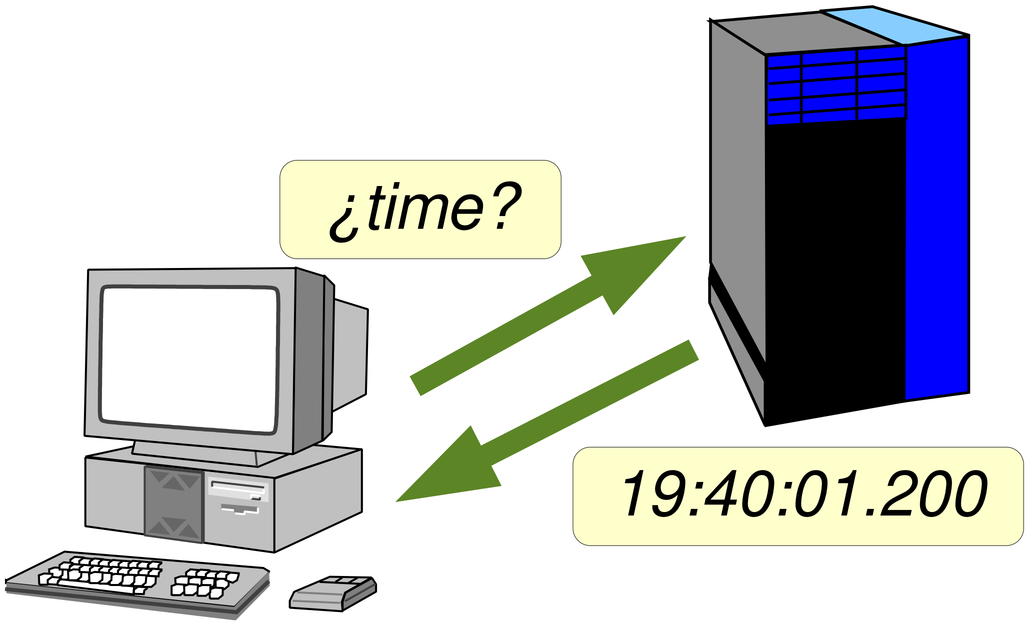
$$|C_j(t) - C_i(t)| < D$$

external sync $\longrightarrow$ internal sync
internal sync $\not\rightarrow$ external sync

Cristian's algorithm for external sync

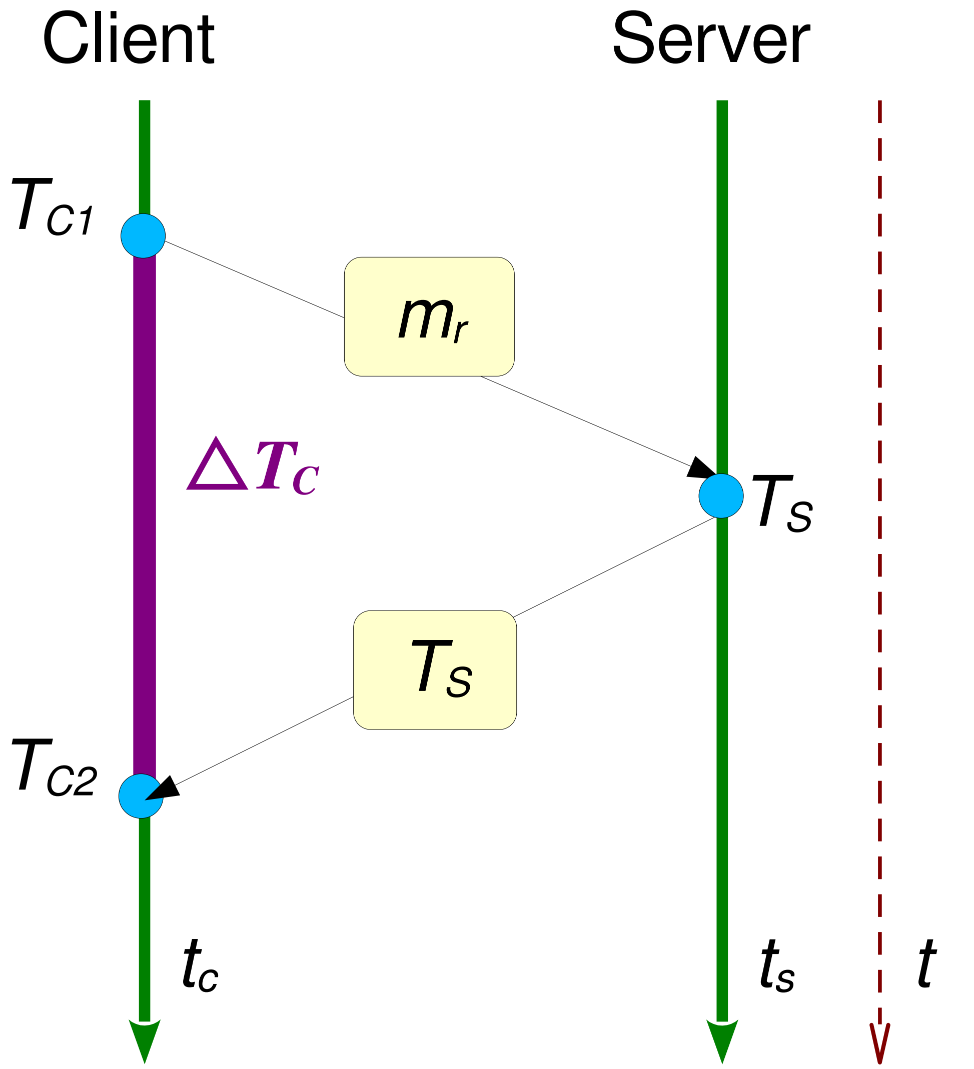


Cristian's algorithm for external sync

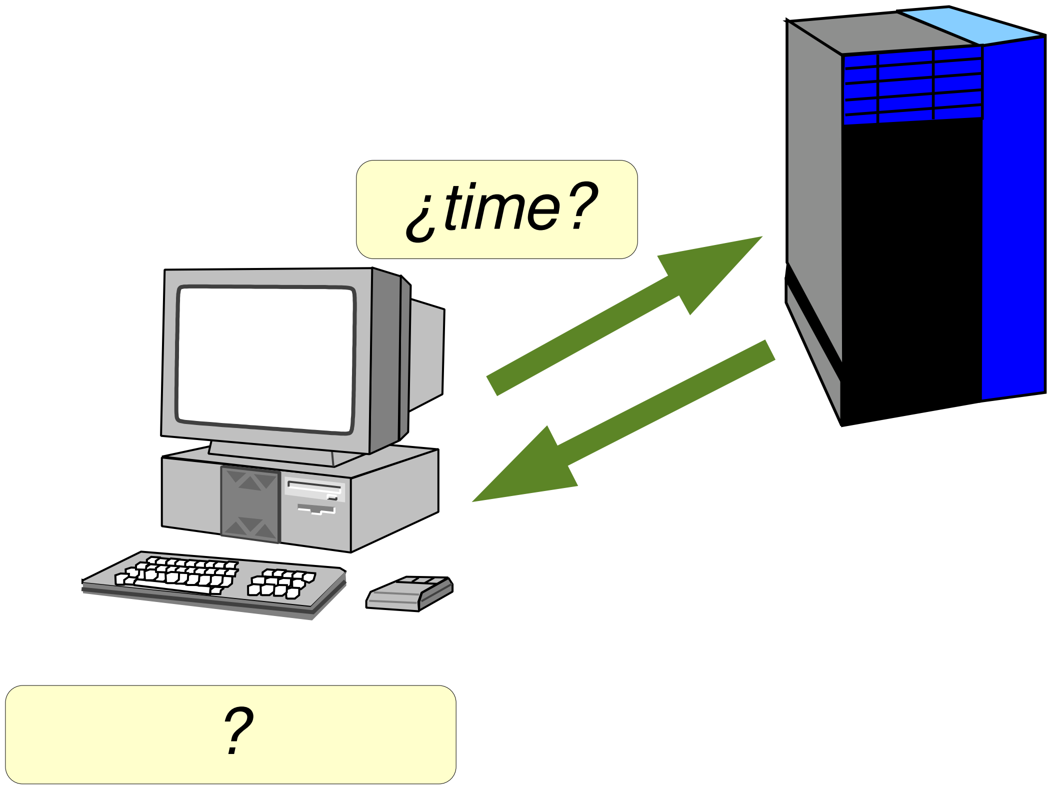


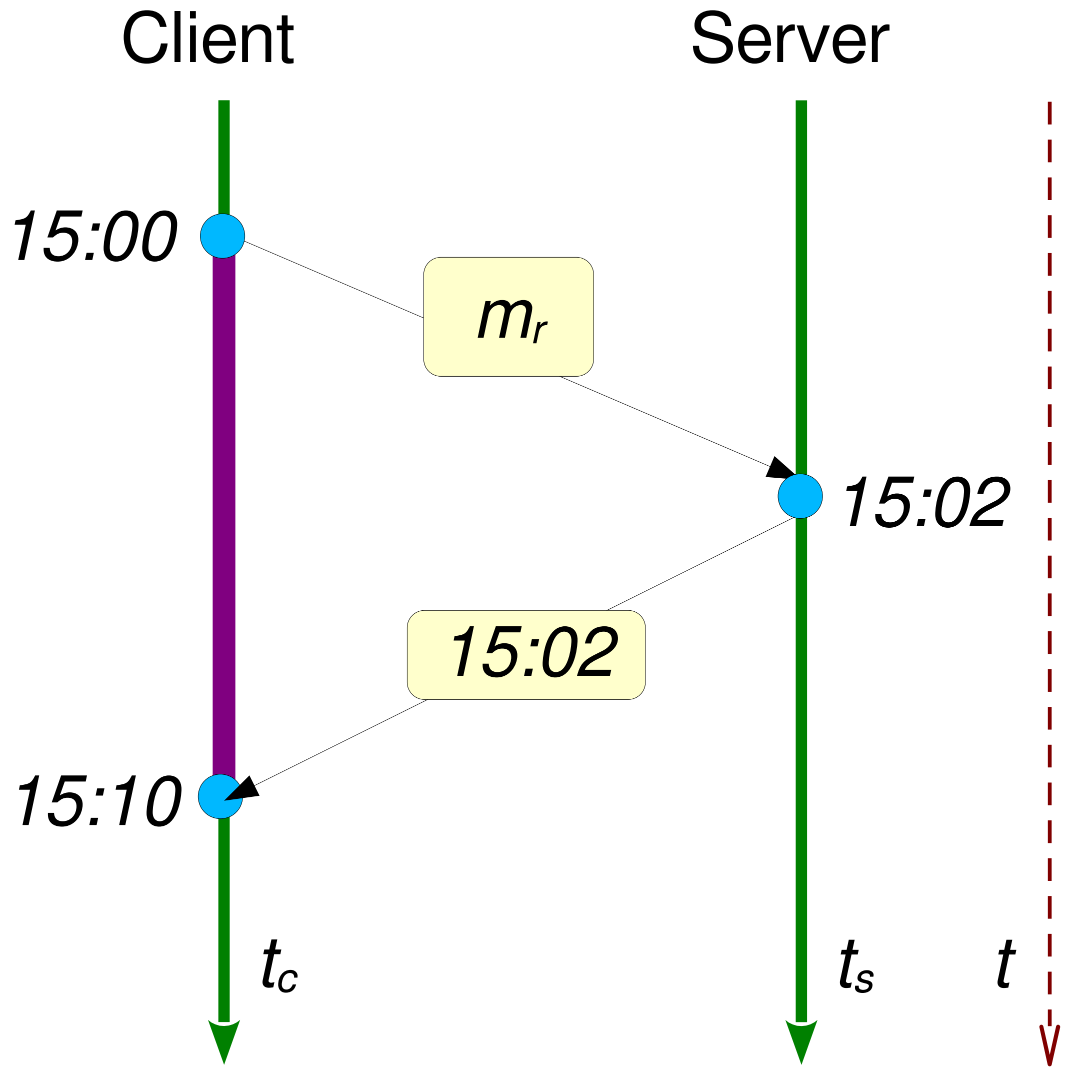
$$T = T_S + \frac{\Delta T_C}{2}$$

$$\varepsilon = \pm \frac{\Delta T_C}{2}$$

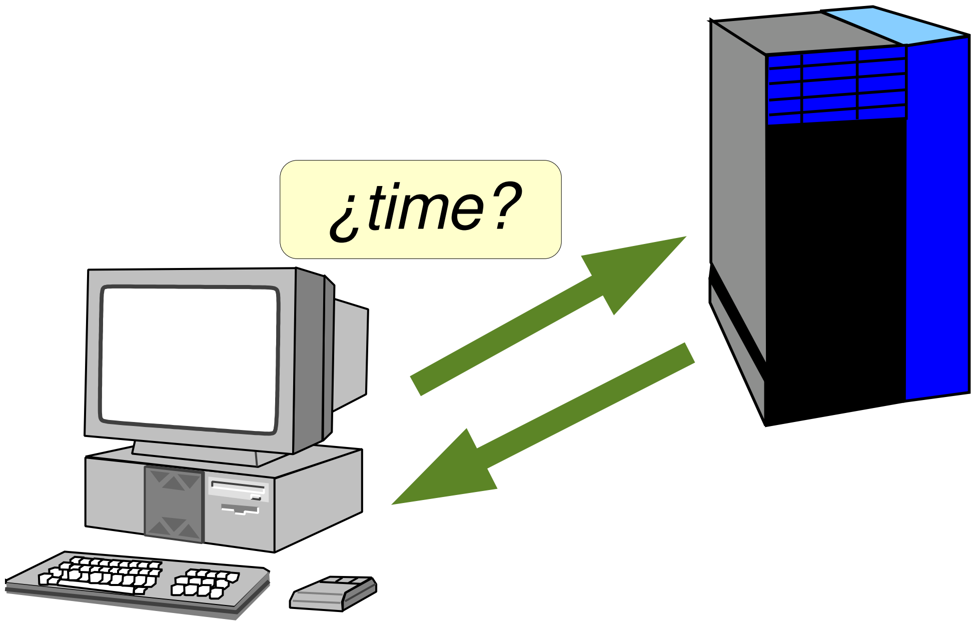


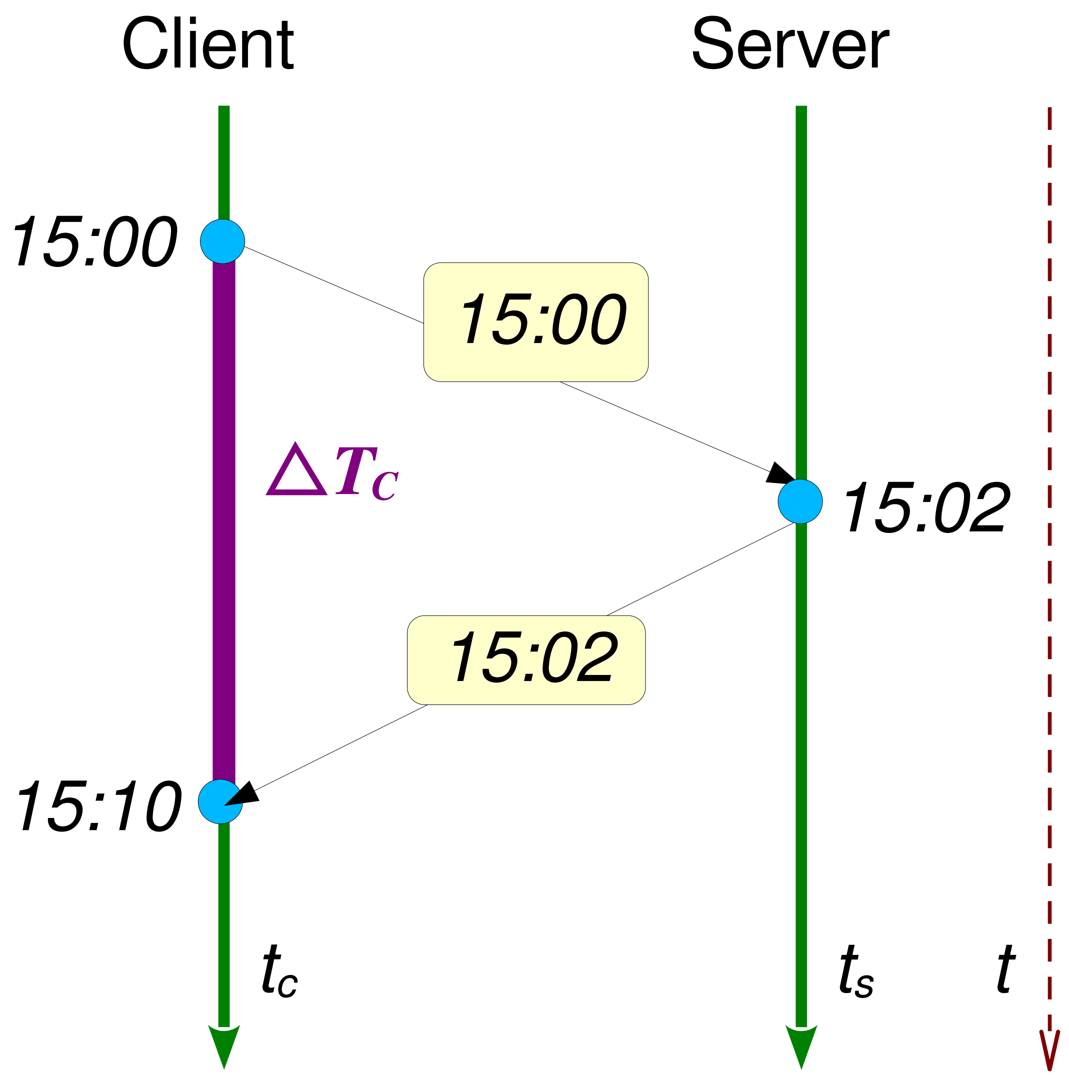
Cristian's algorithm exercise





Cristian's algorithm exercise





15:07

$$T = 15:02 + \frac{15:10 - 15:00}{2}$$

$$\varepsilon = \pm \frac{15:10 - 15:00}{2}$$

NTP
Network Time Protocol

DIY

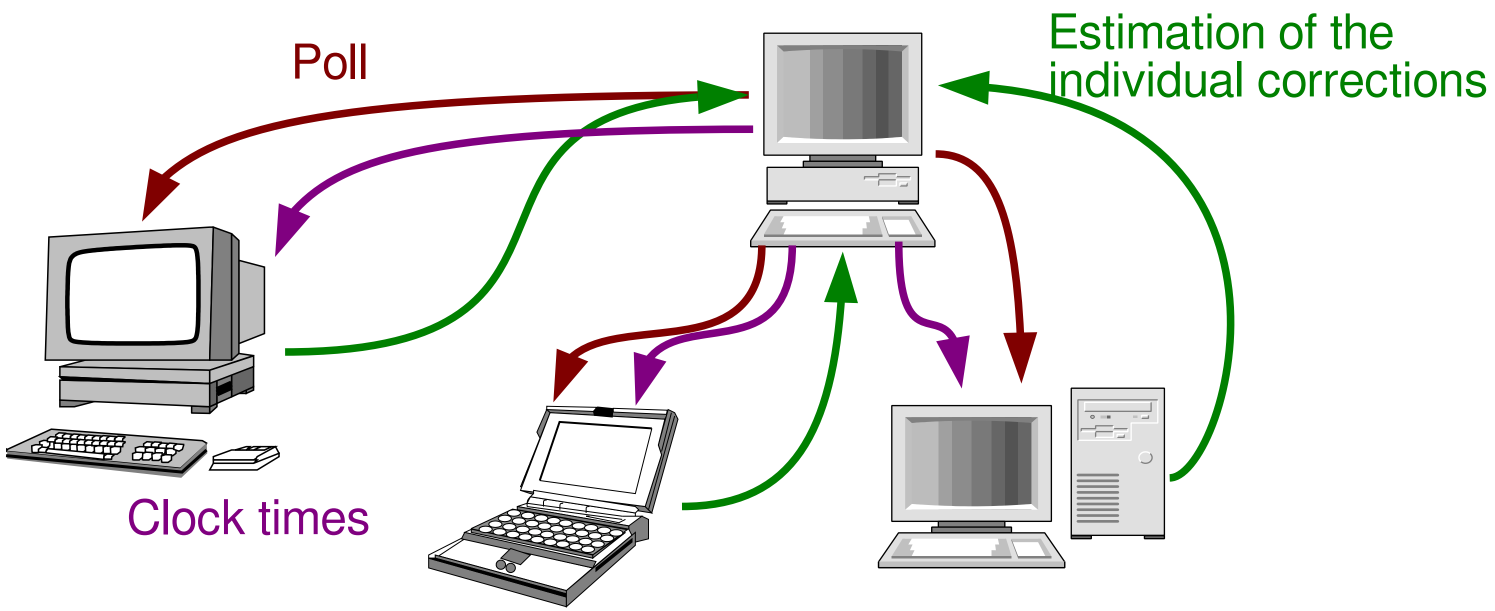
NTP applies the Cristian algorithm.

```
$ ntpstat
synchronised to NTP server (37.139.121.60) at stratum 3
   time correct to within 223 ms
   polling server every 64 s

$ ntpq -pn
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
 0.debian.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.000
 1.debian.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.000
 2.debian.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.000
 3.debian.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.000
-90.165.120.190  150.214.94.10    2 u  136  512  175   23.519   -3.308   2.168
*147.156.7.18    147.156.1.135    2 u  198  512  377   15.879    1.594   0.487
+162.159.200.1   10.40.9.80       3 u  483  512  377    6.570    1.928   0.682
+185.132.136.116 130.206.3.166    2 u  482  512  377   22.703    0.197   0.613
-213.251.52.234  195.66.241.10    2 u  348  512  377    8.192    0.373   0.527
```

Berkeley's algorithm

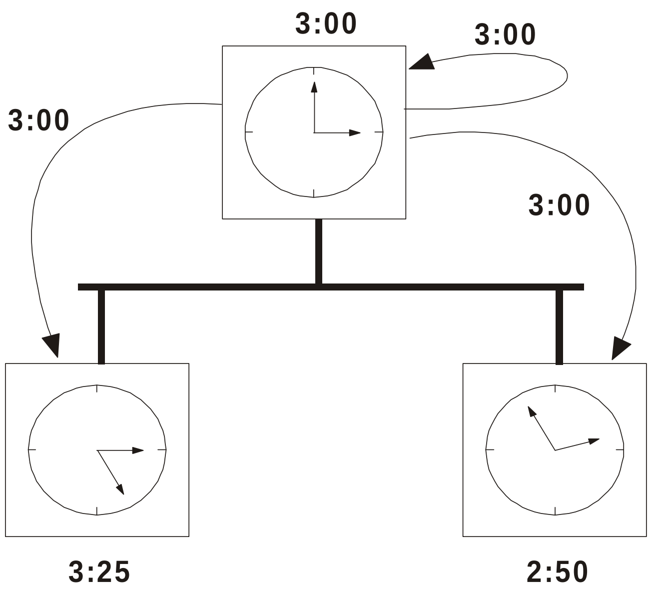
- Gusella and Zatti (1989), internal synchronization
- Algorithm:
  - A computer in the distributed system is elected as master
  - The master periodically polls the other computers (slaves)
  - The slaves response to the master with their clock skews
  - The master estimates the time correction for each slave taking into account the average of the values received and the round-trip times



Berkeley's algorithm

DIY

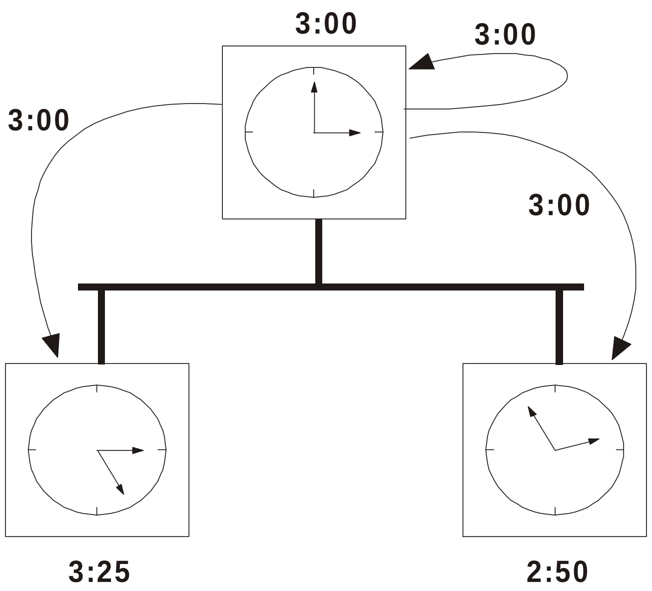
What is the correction for the clocks (ε)?

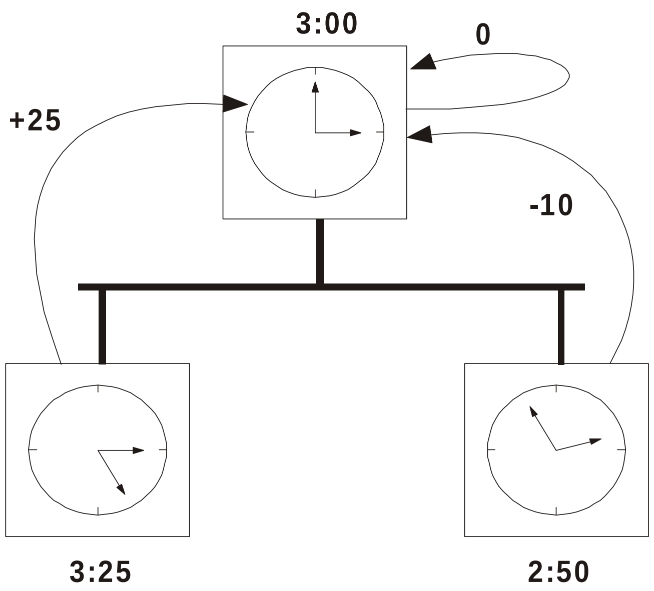


Berkeley's algorithm

DIY

What is the correction for the clocks (ε)?

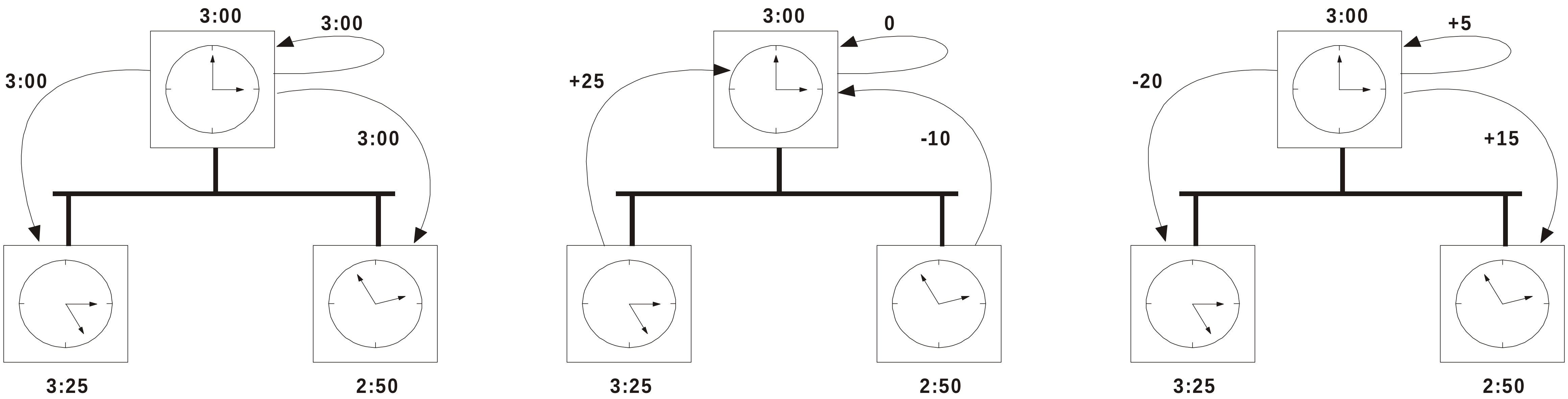




Berkeley's algorithm

DIY

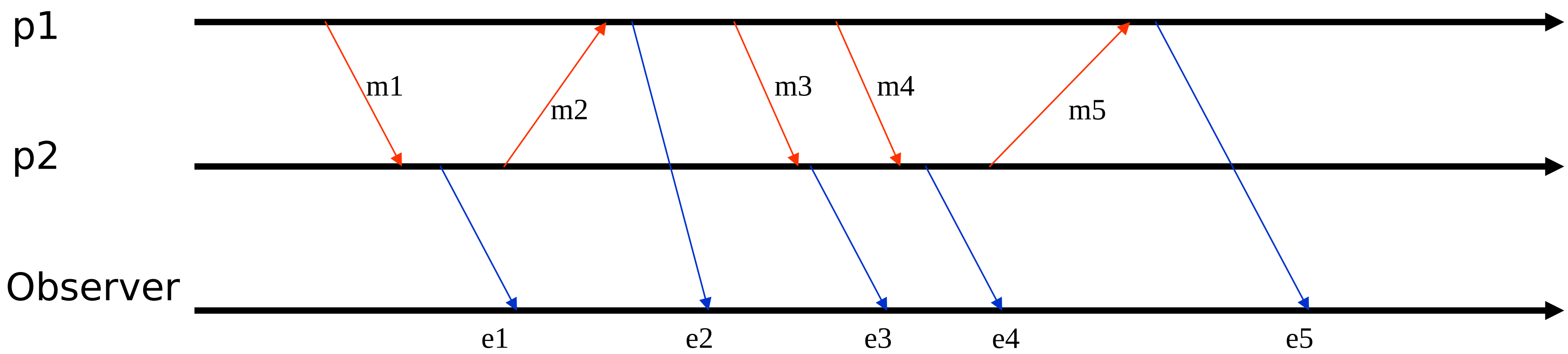
What is the correction for the clocks (ε)?



Physical clocks problems

- Physical clocks are useful to order events that happen on a single process
- However, in a distributed system the events proceed from different processes:
  - ε could be larger than the difference of time of two events from different process (same timestamp for these events)
  - There exists no perfect synchronization

We cannot use physical clocks to order events in a distributed system



Logical time and logical clocks

- Lamport (1978) demonstrates that logical time can be used to order events in a distributed system
- Events are ordered based on two simple points:
  - If two events occurred at the same process p_i, then they occurred in the order in which p_i observes them (order $\rightarrow_i$)
  - If a message is sent between processes, the event of sending the message occurred before the event of receiving that message
- The happened-before relationship (aka *causal ordering* or *potential causal ordering*), denoted as $\rightarrow$, is defined as:
  - If exists a process p_i: $e \rightarrow_i e'$ then $e \rightarrow e'$
  - For any message m, send (m) $\rightarrow$ receive(m)
  - If e, e' and e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$

Logical clocks

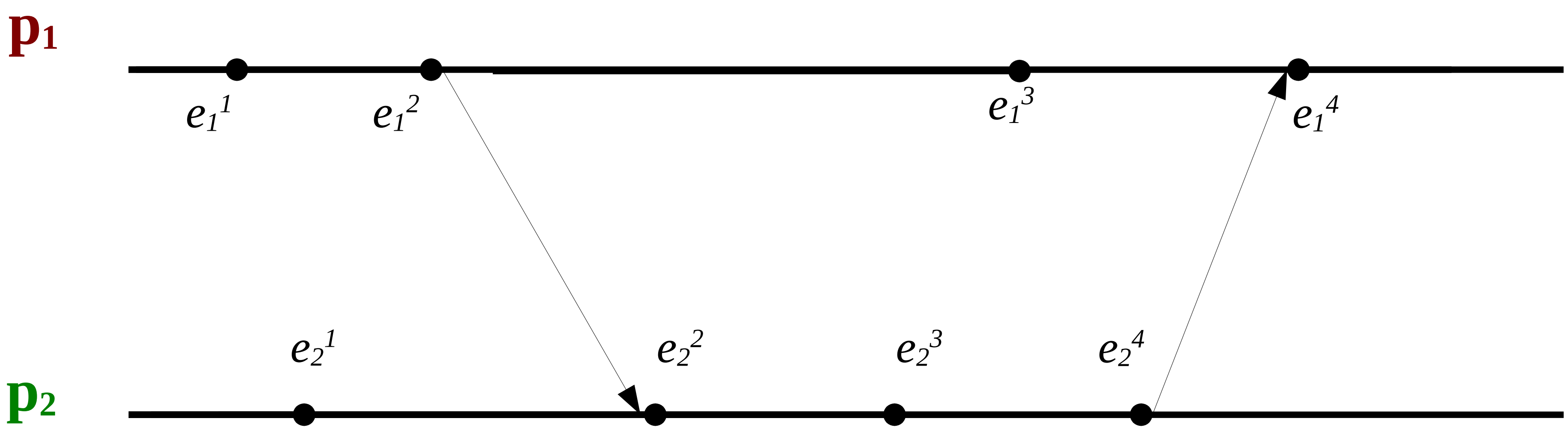
- A logical clock is a a monotonically increasing software counter.
- Timestamps for ordering events.
- Algorithm for computing logical clocks according to the causal ordering (Lamport, 1978):
  - Each process p_i keeps its logical clock value L_i, $L_i(e)$ is the timestamp of event *e* at p_i and $L(e)$ is the timestamp of event *e*
  - Rules to capture the happended-before relation (→):

    1. L_i is increased before each event is issued at p_i : $L_i = L_i + 1$

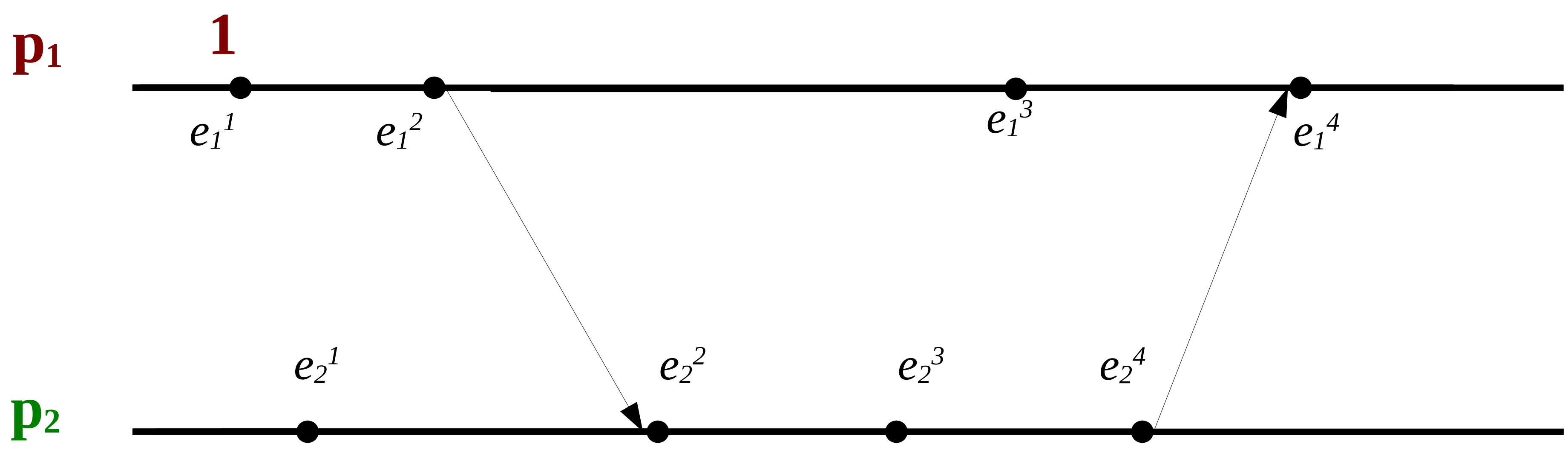
    a) When a process p_i sends a message *m*, it piggybacks on *m* the value of $t = L_i$

    b) On receiving (m,t) a process p_j computes $L_j = \max(L_j, t)$ and then applies rule (1)

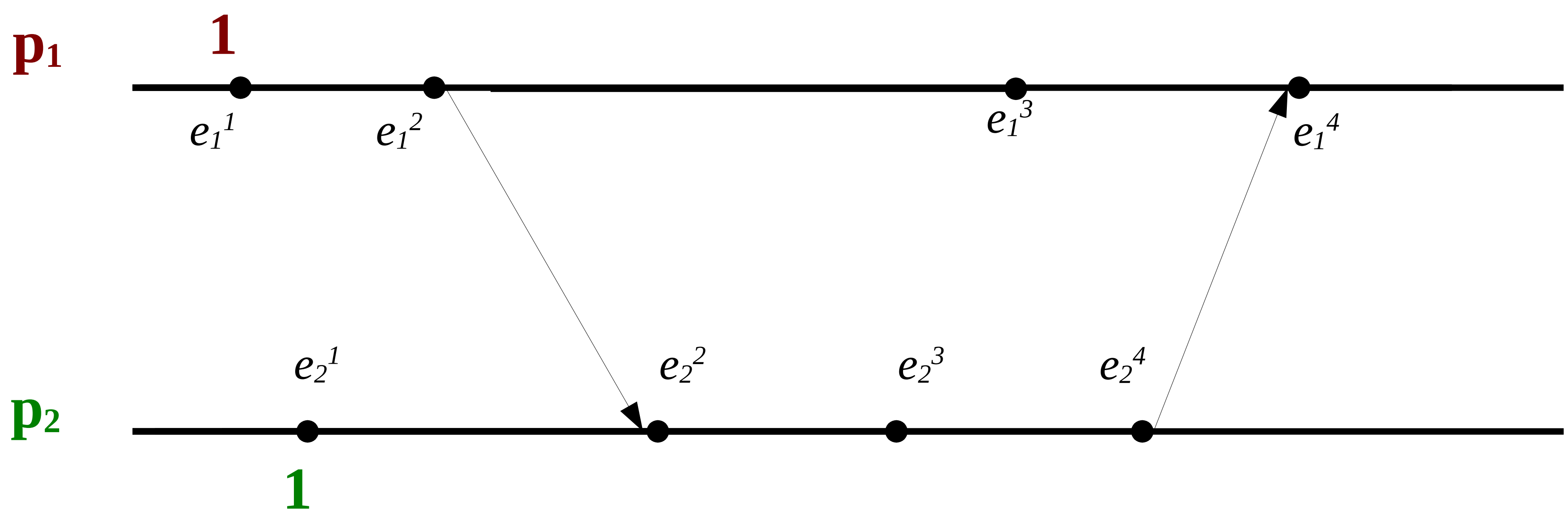
Logical clocks



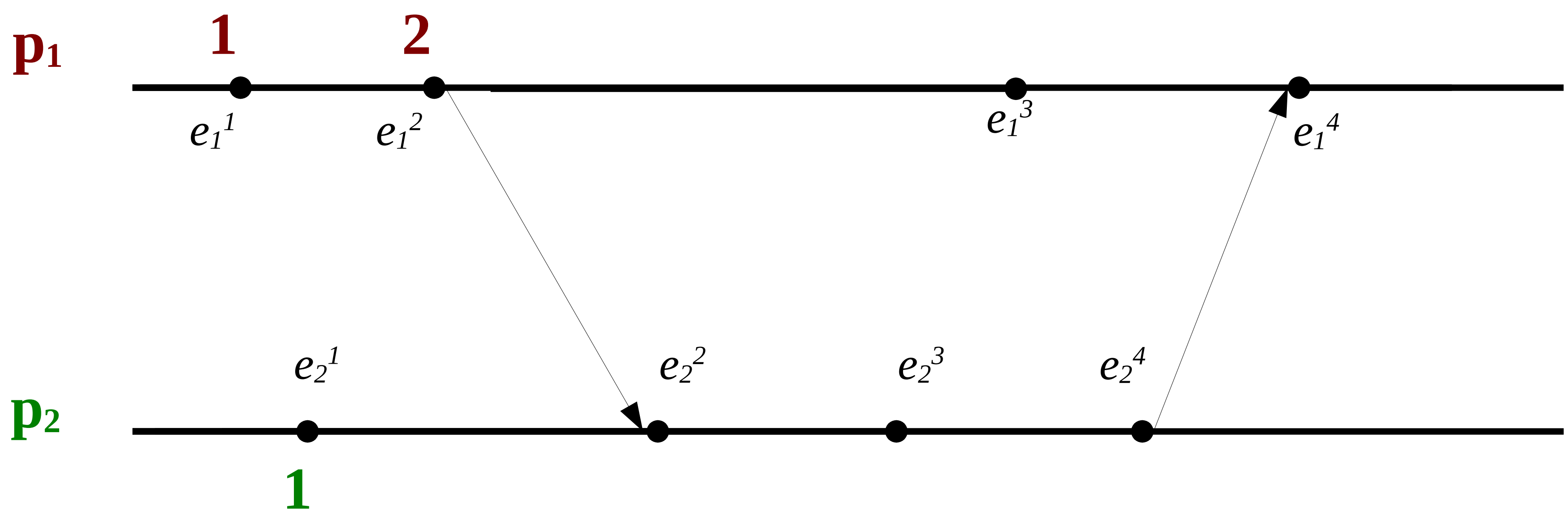
Logical clocks



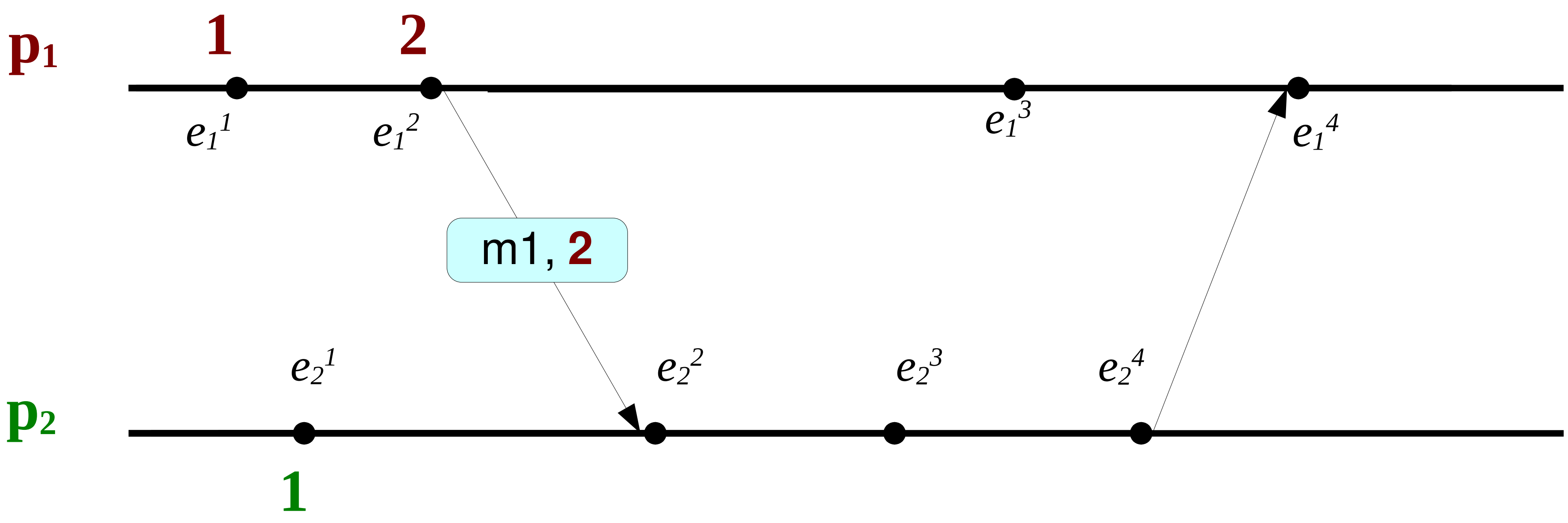
Logical clocks



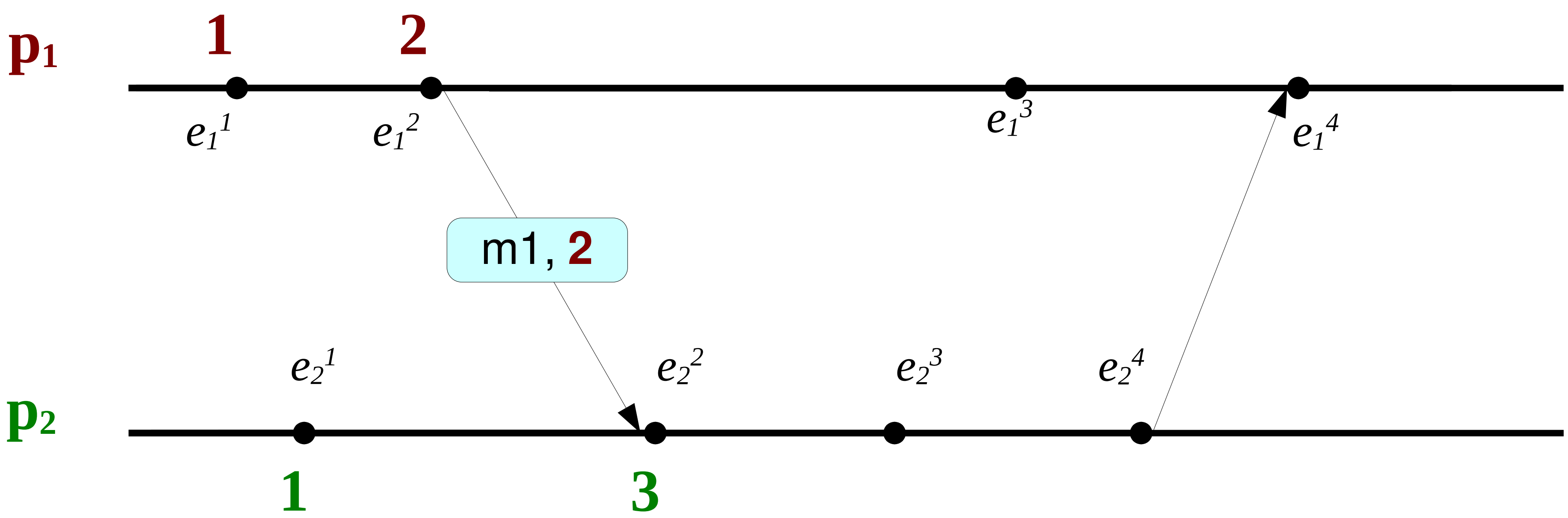
Logical clocks



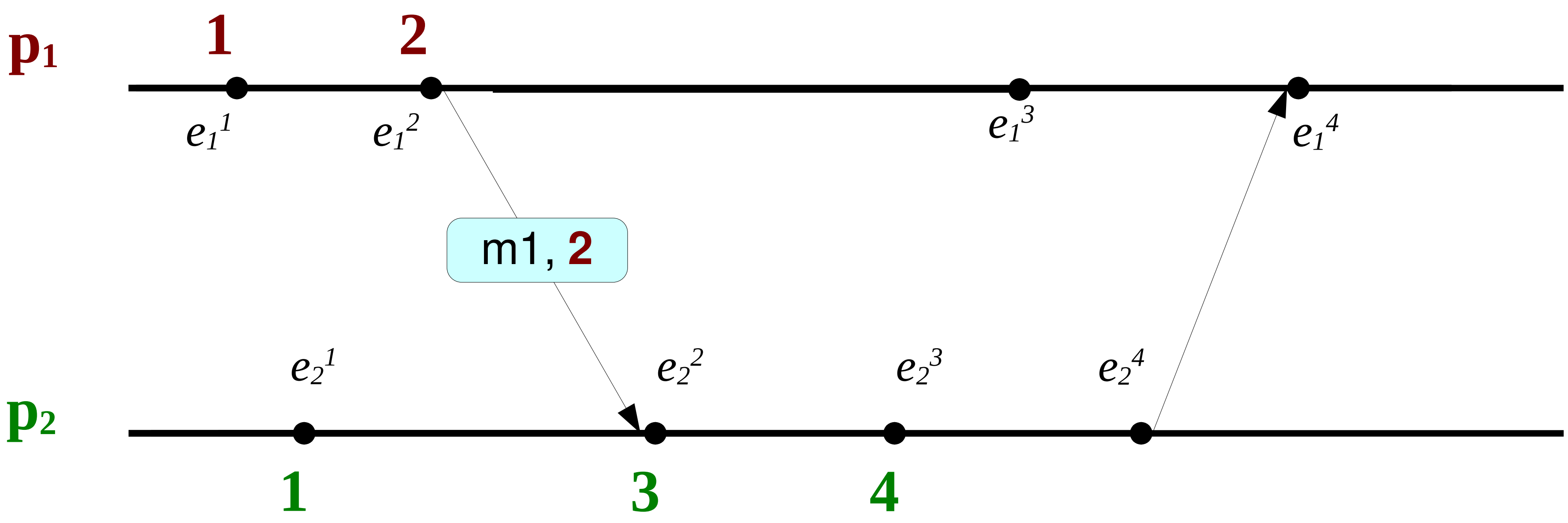
Logical clocks



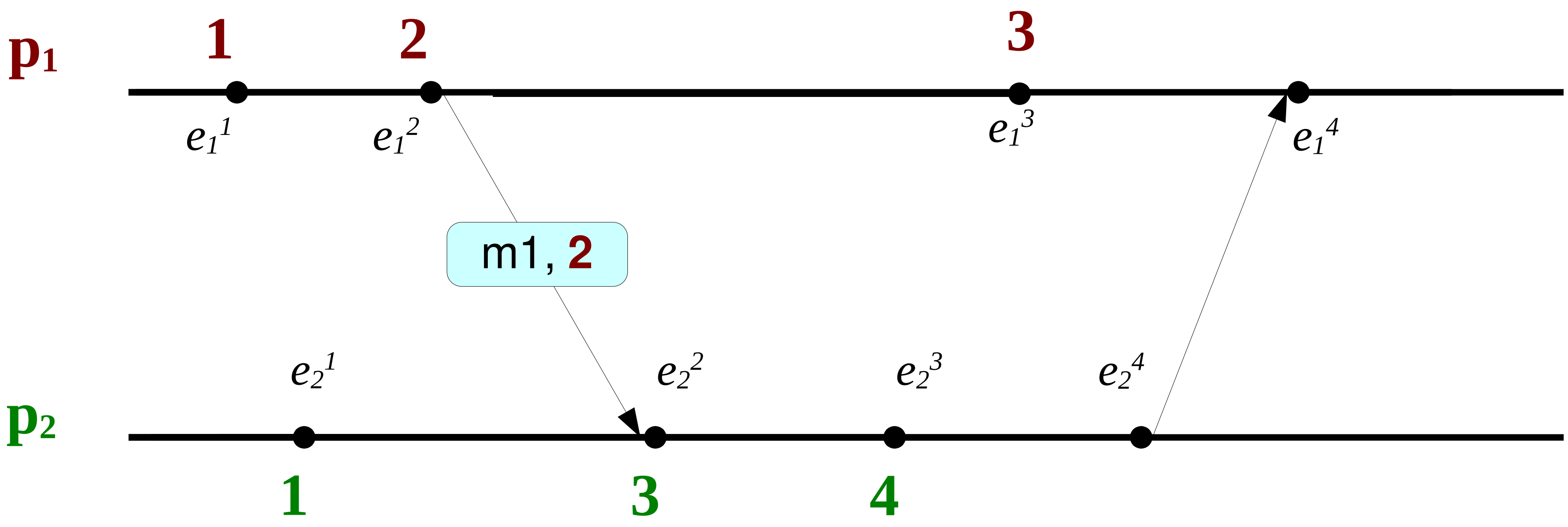
Logical clocks



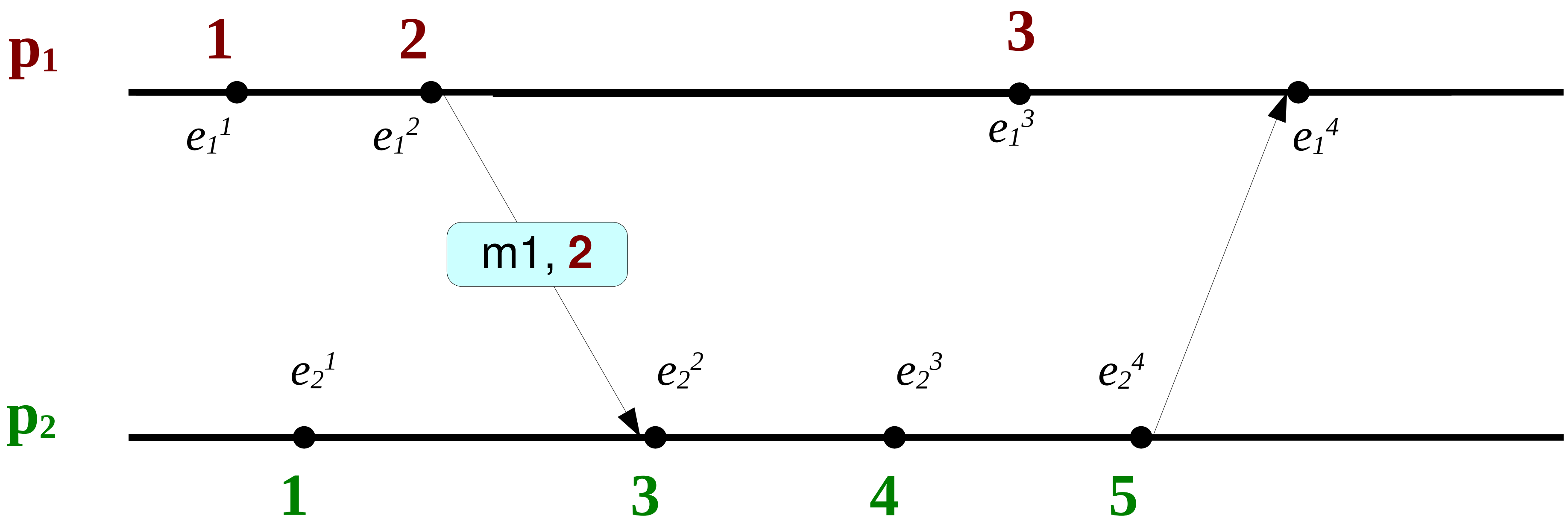
Logical clocks



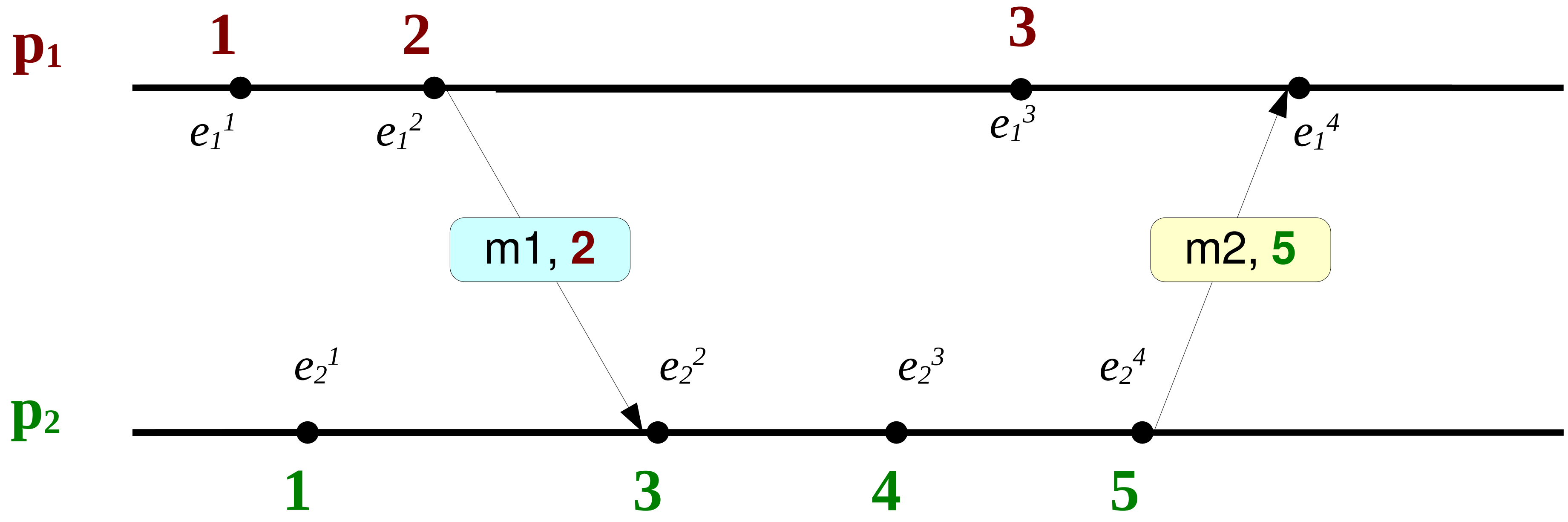
Logical clocks



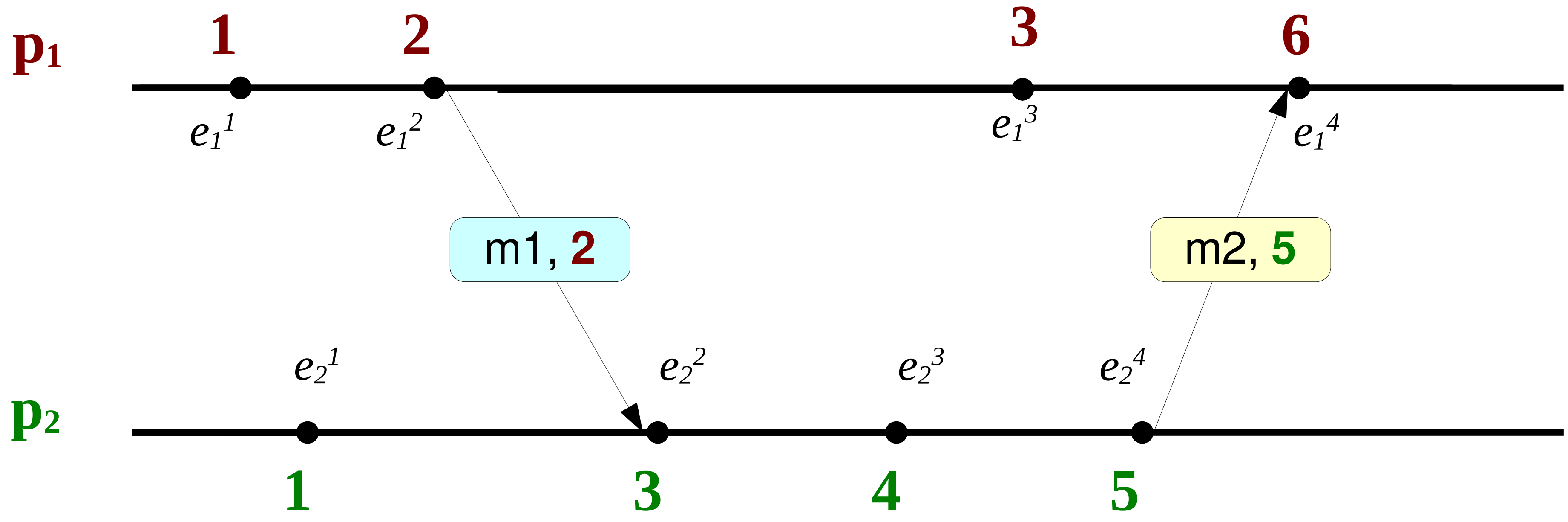
Logical clocks



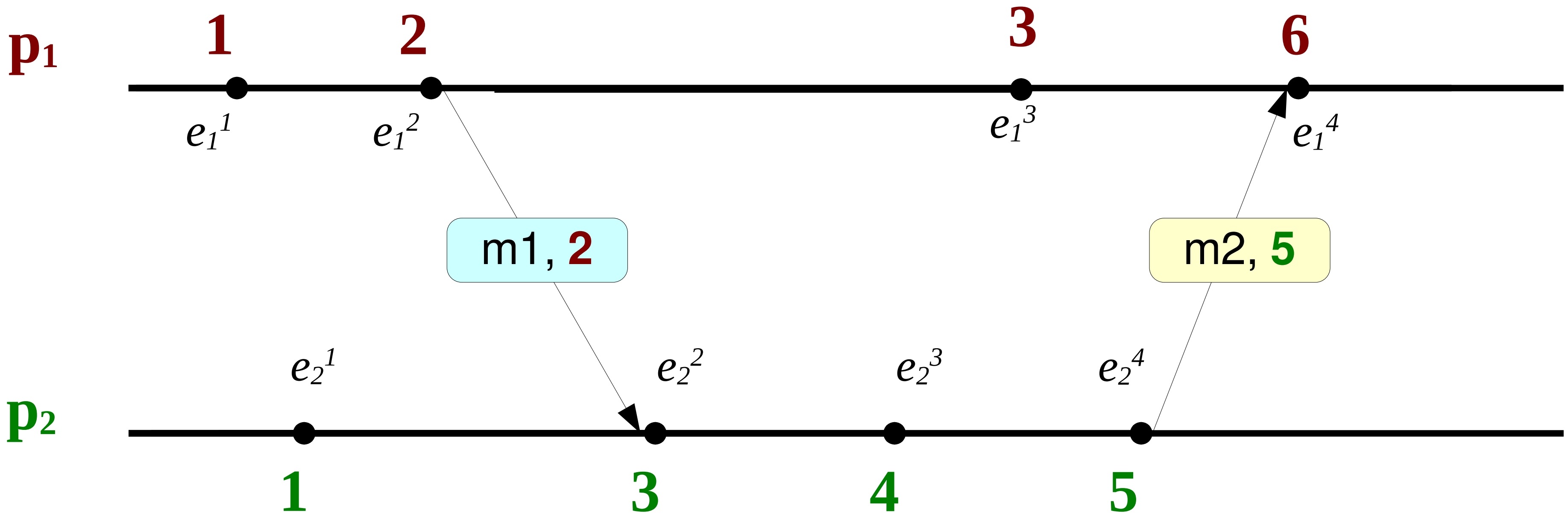
Logical clocks



Logical clocks



Logical clocks



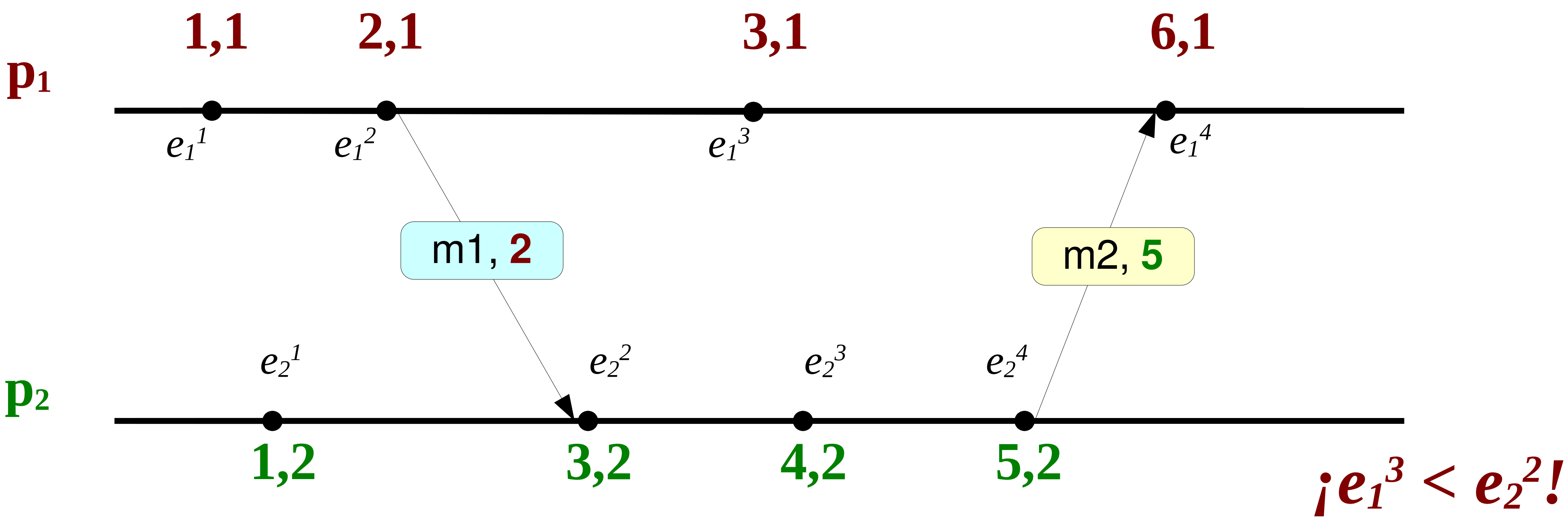
$$e \rightarrow e' \;\Rightarrow\; L(e) < L(e')$$

$$L(e) < L(e') \;\not\Rightarrow\; e \rightarrow e'$$

Totally ordered logical clocks

- Lamport's causal ordering is a partial order, so events at different process may have numerically identical timestamps.
- A total order may be obtained by adding the identifiers of the processes:
  - (T_i, i) is the global logical timestamp of the process i, where:
    - T_i is the local timestamp for event *e* in process p_i
    - i is the identifier of process p_i
- Let (T_i, i) and (T_j, j) be the global timestamps of process i and j, $(T_i, i) < (T_j, j)$ iff:
  - $(T_i < T_j)$ or $(T_i = T_j$ and $i < j)$

Example



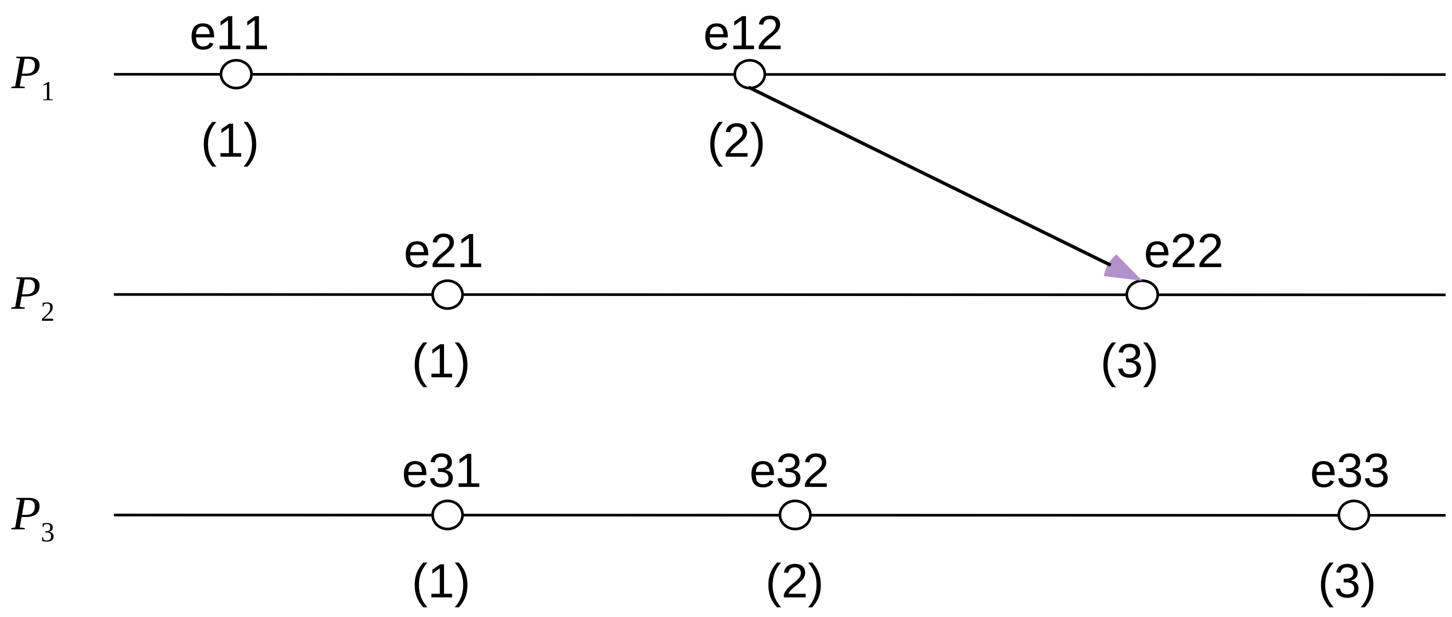
- $e < e' \Leftrightarrow L(e) < L(e') \lor (L(e) = L(e') \land P(e) < P(e'))$
- e_1^3 , e_2^2 cannot be globally ordered because there is not causal relationship between them

Logical clocks: problems

$L(e) < L(e') \not\Rightarrow e \rightarrow e'$

$L(e11) < L(e22) \Rightarrow e11 \rightarrow e22$

$L(e11) < L(e32) \not\Rightarrow e11 \rightarrow e32$



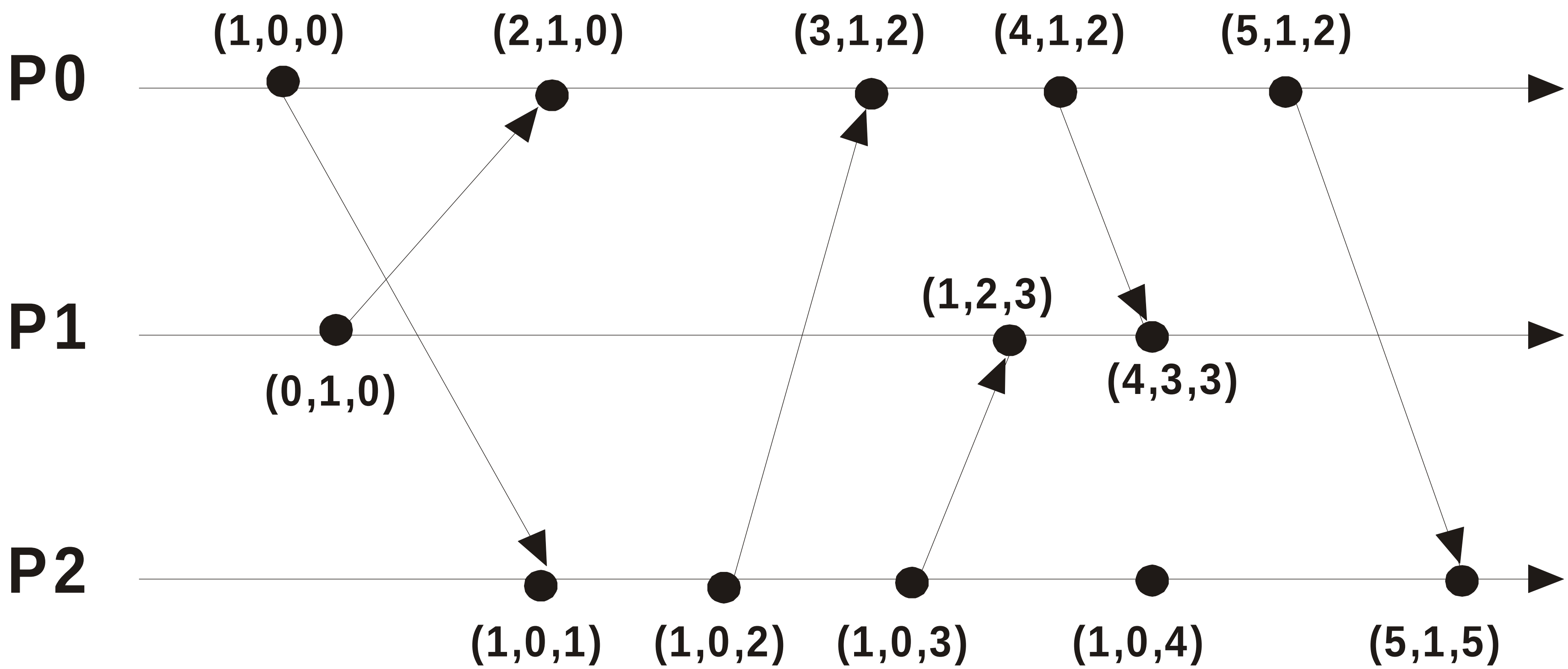
- We need a function F to order any events:

$$a \rightarrow b \Leftrightarrow F(a) < F(b)$$

Vector clocks

- Mattern (1989) and Fidge (1991)
- A vector clock is an array of *n* integers (*n* is the number of processes) used to timestamp local events
- Each process p_i keeps its own vector clock V_i
- $V_i[a]$ is the value of the vector clock at p_i when executes event *a*
- Rules for updating vector clocks:
  - Initially $V_i[j]= 0 \ \forall \ i,j=1..n$
  - Just **before** p_i issues an event: $V_i[i]= V_i[i]+1$
  - When a process p_i sends a message *m,* it piggybacks on *m* the value of $t=V_i$
- When a process p_i receives a message *m* it computes:
  - $V_i[j] =max(V_i[j],t[j]) \ \forall \ j=1..n$
  - $V_i[i] =V_i[i]+1$

Example



Vector clocks comparison

Let V_i and V_j be the vector clocks for processes p_i and p_j:

- $V_i \leq V_j$ iff $V_i[k] \leq V_j[k]$ $\forall$ $k=1..n$ $V_i = [3, \mathbf{x}, 2], V_j = [3, \mathbf{2}, 2]$
- $V_i = V_j$ iff $V_i[k] = V_j[k]$ $\forall$ $k=1..n$ $V_i = [3, \mathbf{2}, 2], V_j = [3, \mathbf{2}, 2]$
- $V_i < V_j$ iff $V_i \leq V_j$ and $V_i \neq V_j$ $V_i = [3, \mathbf{1}, 2], V_j = [3, \mathbf{2}, 2]$

Let *e* and *e'* be events occurring at different processes:

- $e \rightarrow e' \Rightarrow V(e) < V(e')$
- $V(e) < V(e') \Rightarrow e \rightarrow e'$
- Otherwise, e and e' are concurrent (e || e'):
  - Neither $V(e) \leq V(e')$ nor $V(e') \leq V(e)$

Example: causes and effects