Distributed Systems

a.y. 2023/2024

Distributed Systems:

Time

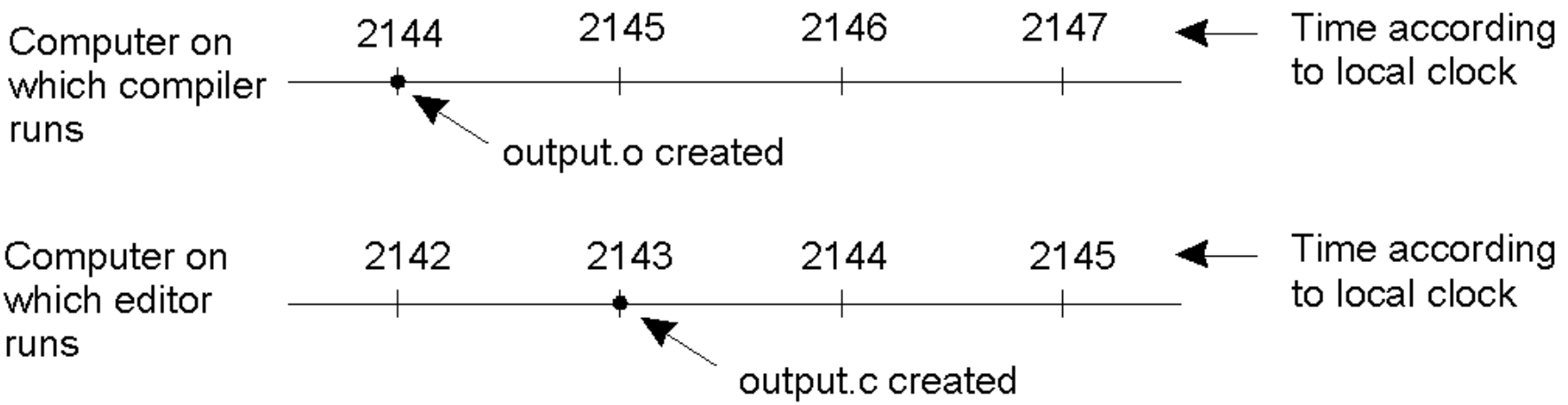
Time and Clocks

- We need to measure time accurately:
  - At what time an event occurred at a computer ?
- Algorithms for clock synchronization are useful for
  - concurrency control based on timestamp ordering
  - authenticity of requests
  - avoiding duplicate updates

- There is no global clock in a distributed system, hence no absolute global time
  - clock accuracy and synchronisation
- Process state and global state
  - Are there some states occuring “at the same time” ?

- What clock properties are required by the Unix *make* program when it uses local files?
- What clock properties are required by the Unix *make* program when it uses distributed files?

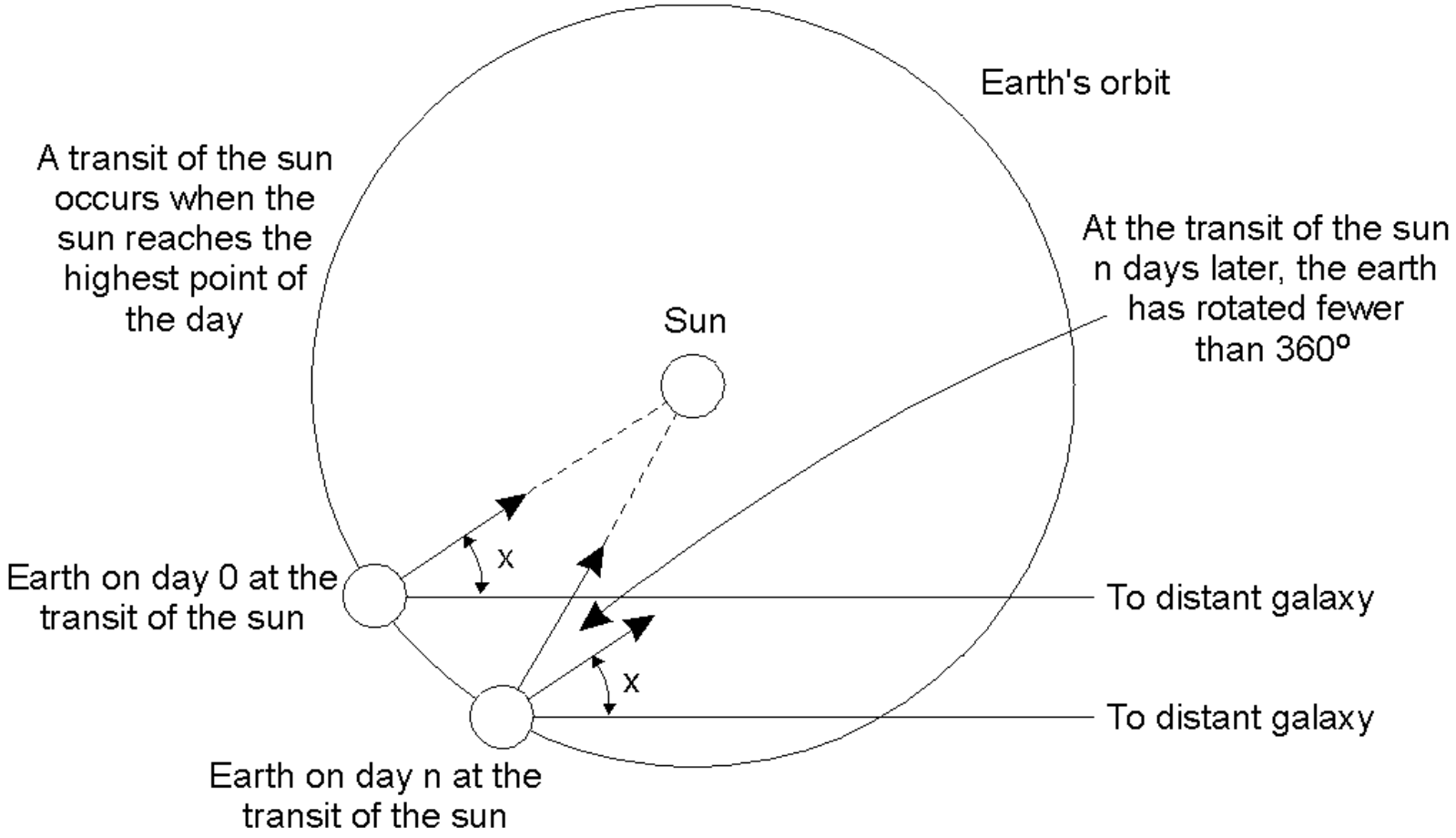
Clock Synchronization



- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

- Logical time is an alternative
  - It focuses on ordering of events

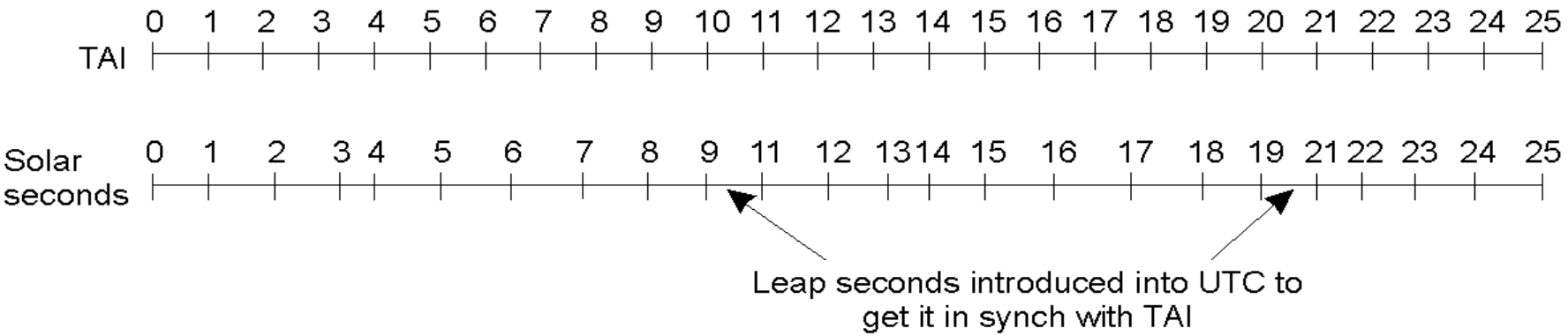
Computation of the mean solar day



Coordinated Universal Time (UTC)

- International Atomic Time (TAI) is based on very accurate physical clocks (drift rate 10^{-13})
- UTC is an international standard for time keeping
- It is based on atomic time, but occasionally adjusted to astronomical time
- It is broadcast from radio stations on land and satellite (e.g. GPS)

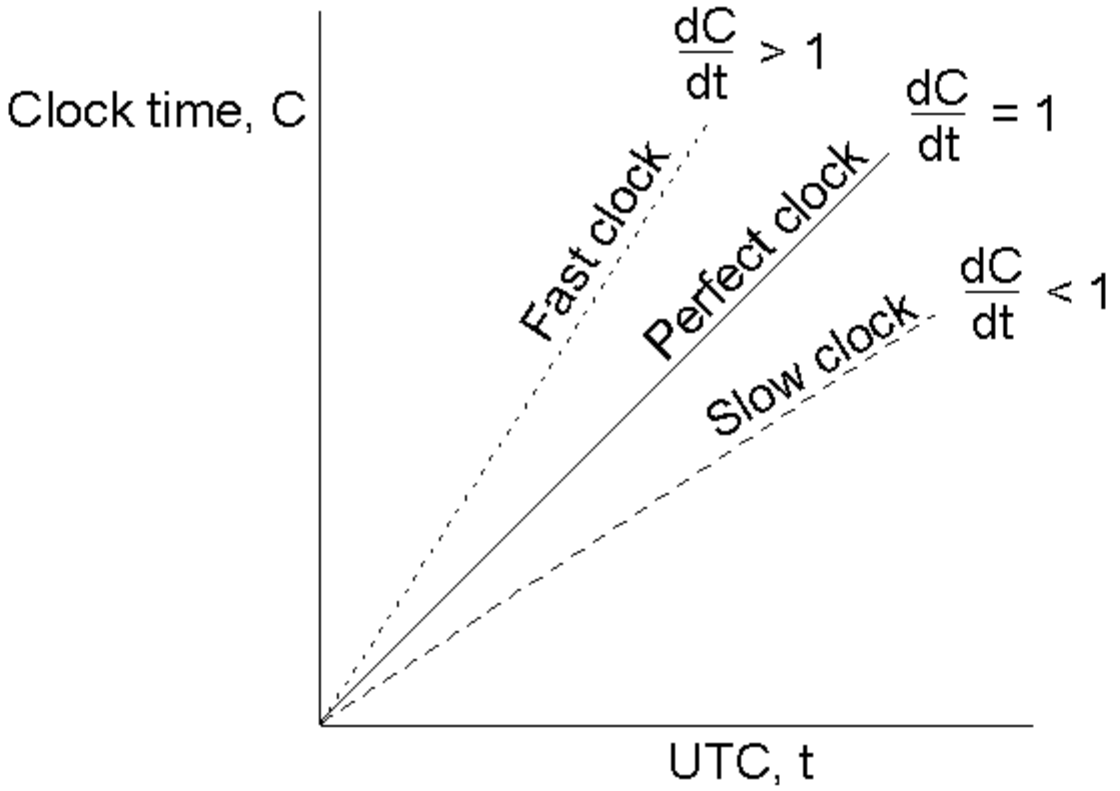
…adjusting physical clock…



- TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

- Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1-10 millisecond
- Signals from GPS are accurate to about 1 microsecond

…clock time and UTC…



- Each computer in a DS has its own internal clock used by local processes to obtain the value of the current time

- clocks on different computers may give different times
  - computer clocks drift from perfect time and their drift rates differ from one another.
  - ***clock drift rate***: the relative amount that a computer clock differs from a perfect clock

- Even if clocks on all computers in a DS are set to the same time, their clocks will eventually vary quite significantly unless corrections are applied

Remind ...

- A distributed system is defined as a collection P of N processes p_i , $i = 1,2,\ldots N$
- Processors do not share memory
- Each process p_i has a state s_i consisting of its variables (which it transforms as it executes)
- Processes communicate only by messages (via a network)
- Actions of processes:
  - *Send, Receive,* change their own (internal) state
- *Event*: the occurrence of a single action that a process carries out as it executes

- Events at a single process p_i can be placed in a total ordering denoted by the relation $\rightarrow_i$ between the events. i.e.

  $e \rightarrow_i e'$ if and only if e occurs before e' at p_i

- A history of process p_i is a series of events ordered by $\rightarrow_i$

  $history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

Clocks

The computer's clock (for timestamping events)

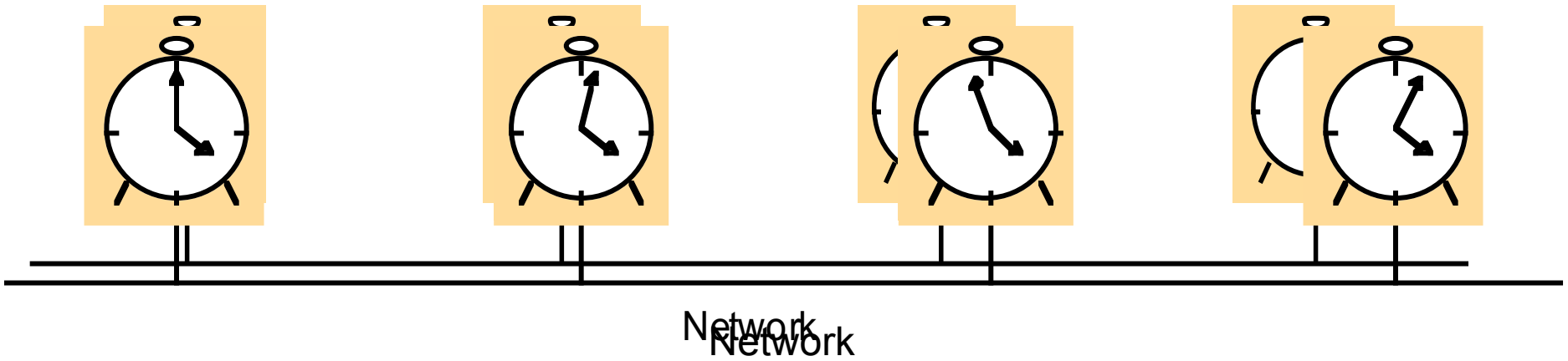
- the time on the computer's hardware clock $H_i(t)$
- The software clock:

  $C_i(t) = \alpha H_i(t) + \beta$

- $C_i(t)$ is the reading of the software clock

Clock resolution < time interval between successive events

Skew between computer clocks



Skew: the difference between the times on two clocks (at any instant)

- Computer clocks are subject to *clock drift* (they count time at different rates)
- Clock *drift rate*: the difference per unit of time from some ideal reference clock

- Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).
- High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

Synchronizing (physical) clocks

- External synchronization
  - A computer's clock C_i is synchronized with an external authoritative time source S, if:
  - $|S(t) - C_i(t)| < D$ for $i = 1, 2, \ldots N$ over an interval I
  - The clocks C_i are accurate to within the bound D.
- Internal synchronization
  - The clocks of a pair of computers are synchronized with one another so that:
  - $| C_i(t) - C_j(t)| < D$ for $i = 1, 2, \ldots N$ over an interval I
  - The clocks C_i and C_j agree within the bound D.

- Internally synchronized clocks are not necessarily externally synchronized, as they may drift collectively

- If the set of processes P is synchronized externally within a bound D, it is also internally synchronized within bound $2D$

Clock correctness

- A hardware clock, H is said to be correct if its drift rate is within a bound $\rho > 0$. (e.g. 10^{-6} secs/ sec)

- This means that the error in measuring the interval between real times t and t' is bounded:
  - $(1 - \rho)\ (t' - t) \leq H(t') - H(t) \leq (1 + \rho)\ (t' - t)$

    (where $t' > t$)
  - Which forbids jumps in time readings of hardware clocks

- Weaker condition of monotonicity
  - $t' > t \Rightarrow C(t') > C(t)$
  - e.g. required by Unix *make*
  - can achieve monotonicity with a hardware clock that runs fast by adjusting the values of α ans β ($C_i(t) = \alpha H_i(t) + \beta$)

- *a faulty* clock is one that does not obey its correctness condition
- *crash failure* - a clock stops ticking
- *arbitrary* failure - any other failure e.g. jumps in time

the 'Y2K bug' …

Clock synchronization in a synchronous system

- a synchronous distributed system is one in which :
  - the time to execute each step of a process has known lower and upper bounds
  - each message transmitted over a channel is received within a known bounded time
  - each process has a local clock whose drift rate from real time has a known bound

Internal synchronization

- One process p_1 sends its local time t to process p_2 in a message m,

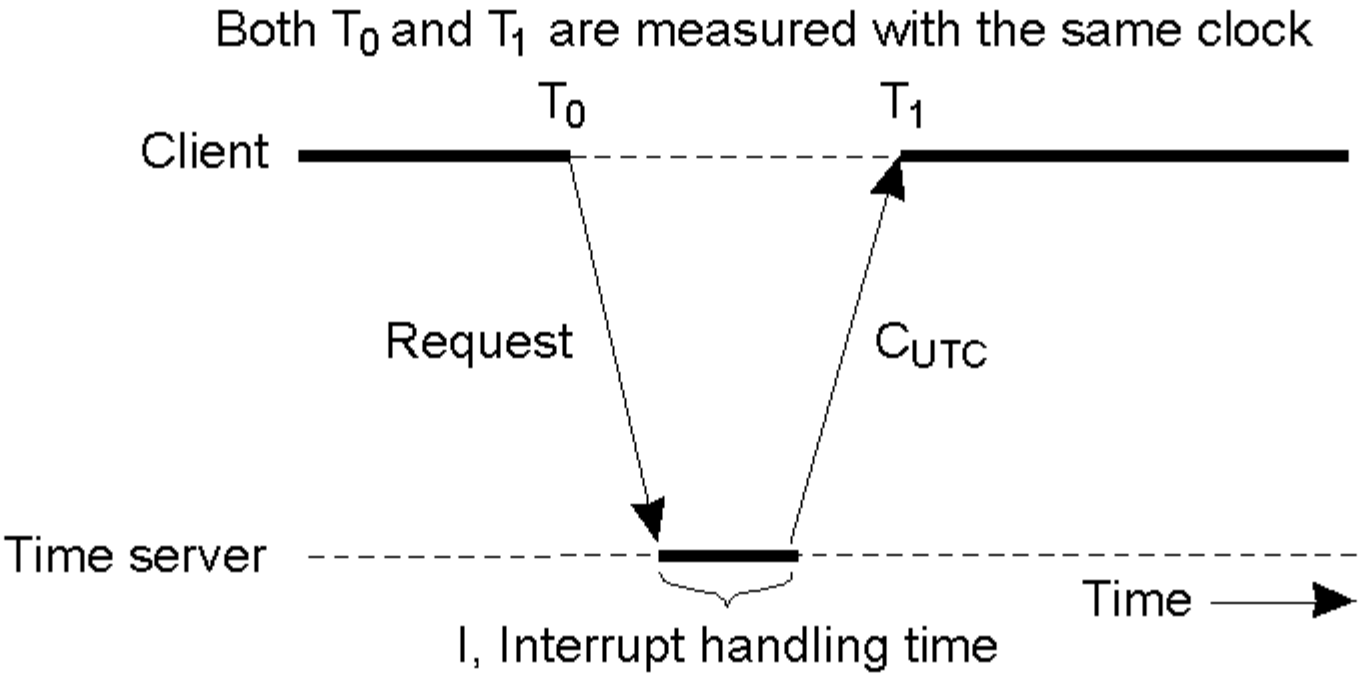
- p_2 could set its clock to $t + T_{trans}$ where T_{trans} is the time to transmit m

- T_{trans} is unknown but $min \leq T_{trans} \leq max$
- uncertainty $u = max\text{-}min$. Set clock to $t + (max + min)/2$ then skew $\leq u/2$

In the Internet, we can only say $T_{trans} = min + x$ where $x >= 0$

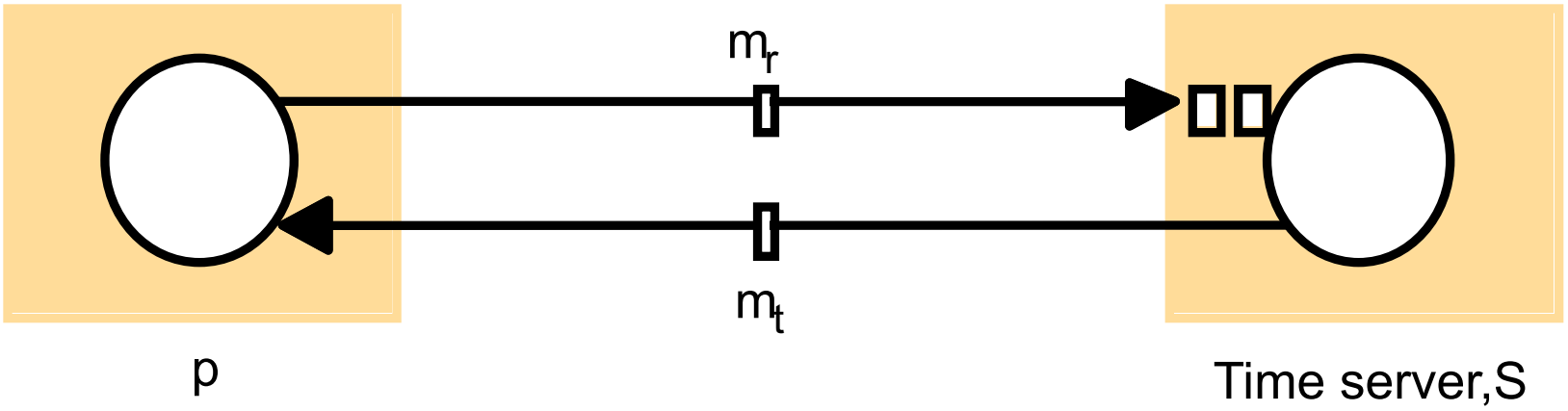
- Cristian’s algorithm -
  - a single time server might fail, so they suggest the use of a group of synchronized servers
  - it does not deal with faulty servers

Cristian's Algorithm



- Getting the current time from a time server.

Cristian's method for an asynchronous system



- A time server *S* receives signals from a UTC source
- Process *p* requests time at m_r and receives *t* at m_t
- *p* sets its clock to $t + T_{round}/2$

T_{round} is the round trip time recorded by *p*

Cristian's method (1989) for an asynchronous system

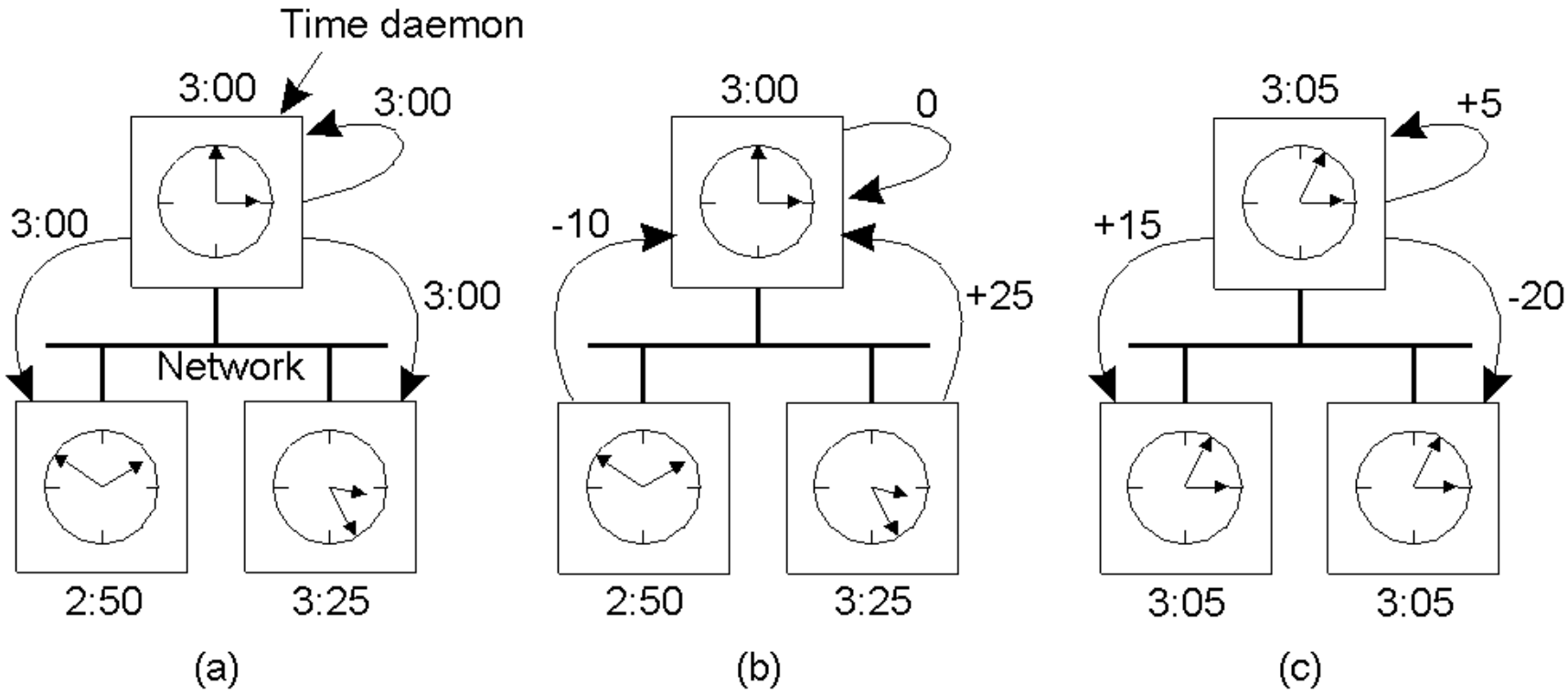
- Accuracy $\pm$ ($T_{round}/2$ - *min*) :
  - because the earliest time *S* puts *t* in message m_t is *min* after *p* sent m_r.
  - the latest time was *min* before m_t arrived at *p*
  - the time by *S*'s clock when m_t arrives is in the range [t+*min*, $t + T_{round}$ - *min*]

min is an estimated minimum round trip time

Berkeley algorithm

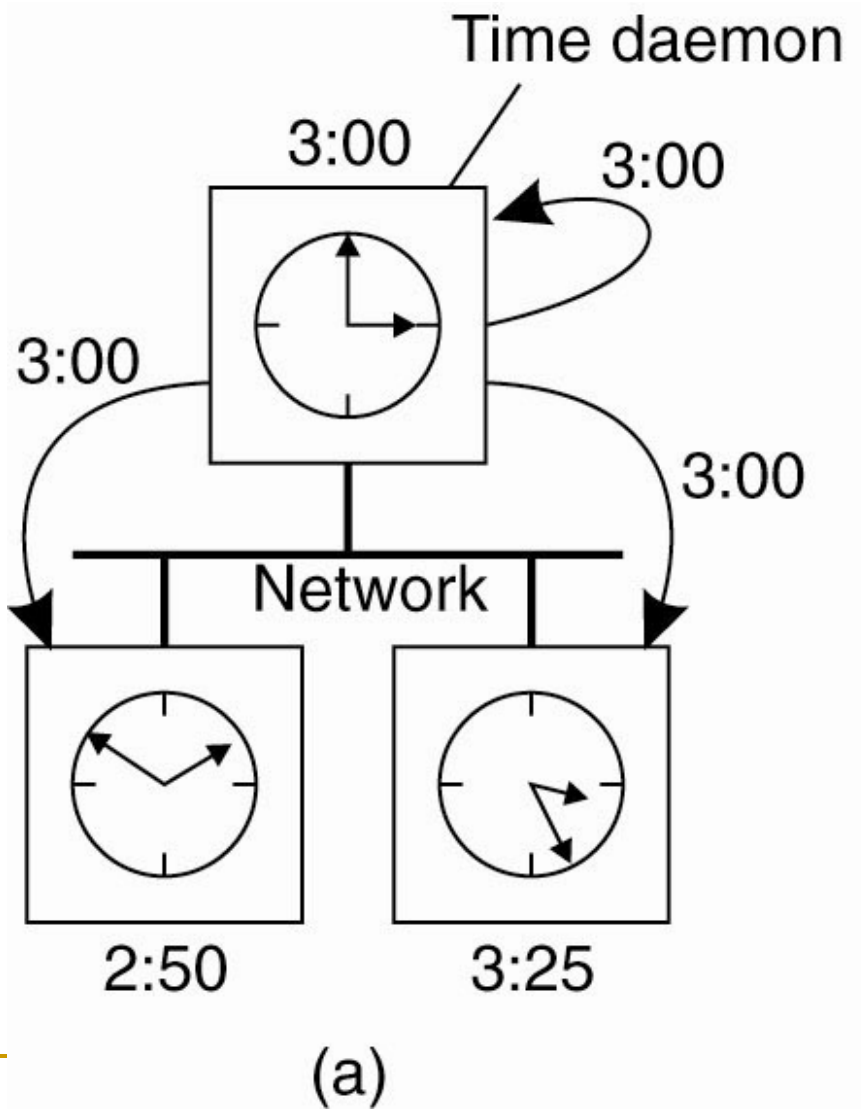
- Berkeley algorithm
  - An algorithm for internal synchronization of a group of computers
  - A *master* polls to collect clock values from the others (*slaves*)
  - The master uses round trip times to estimate the slaves' clock values

The Berkeley Algorithm



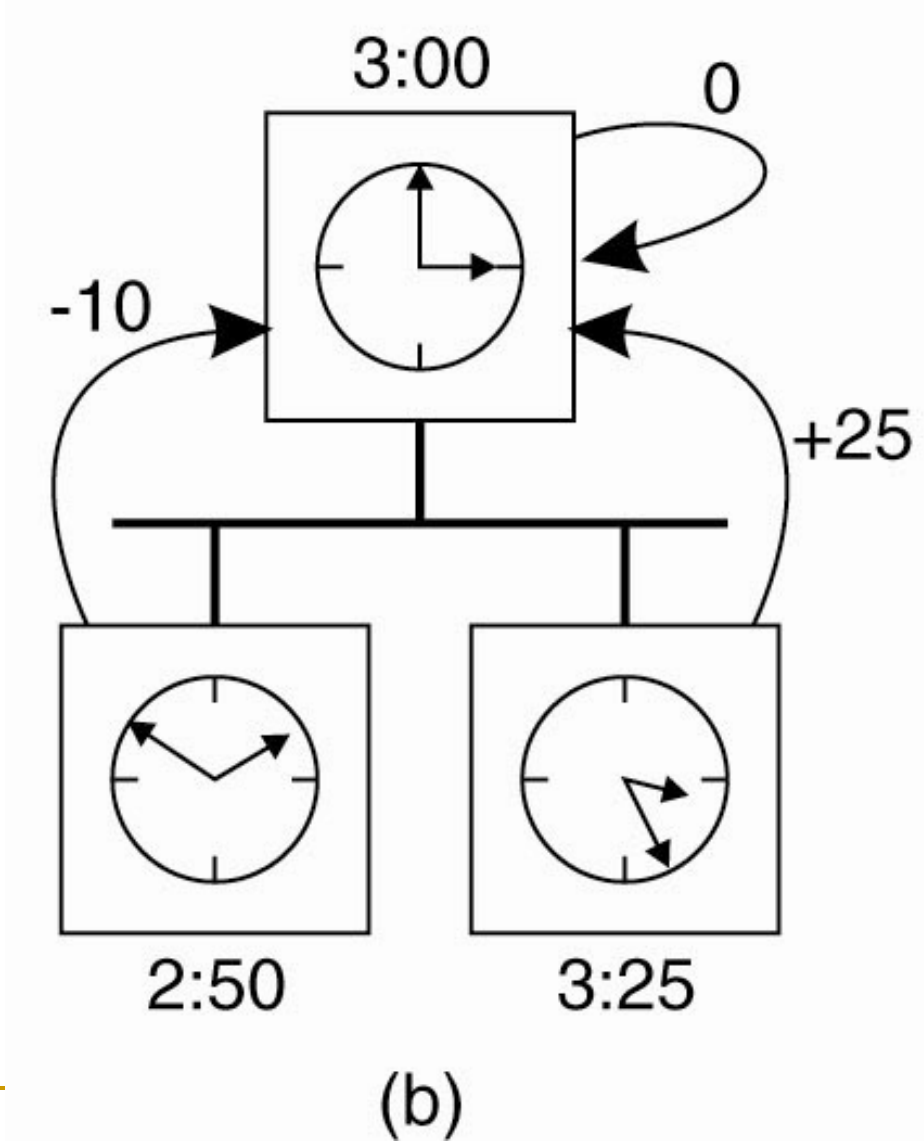
The Berkeley Algorithm

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



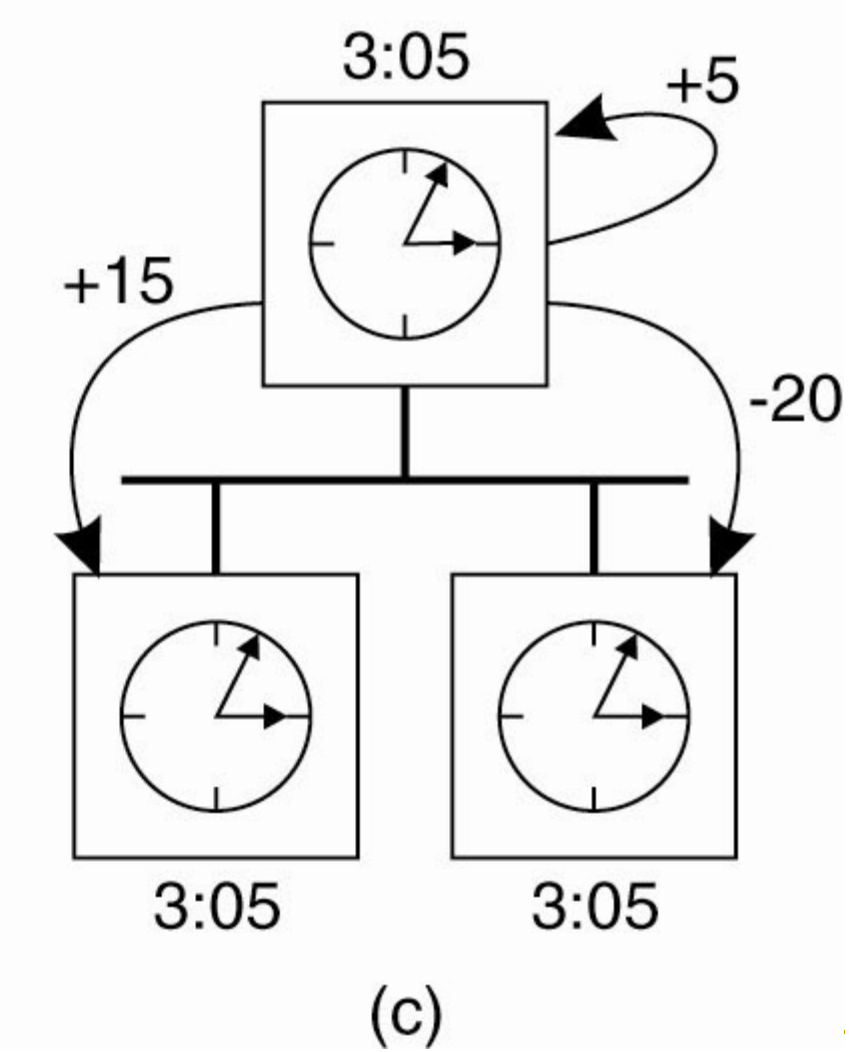
The Berkeley Algorithm

- The machines answer.



The Berkeley Algorithm

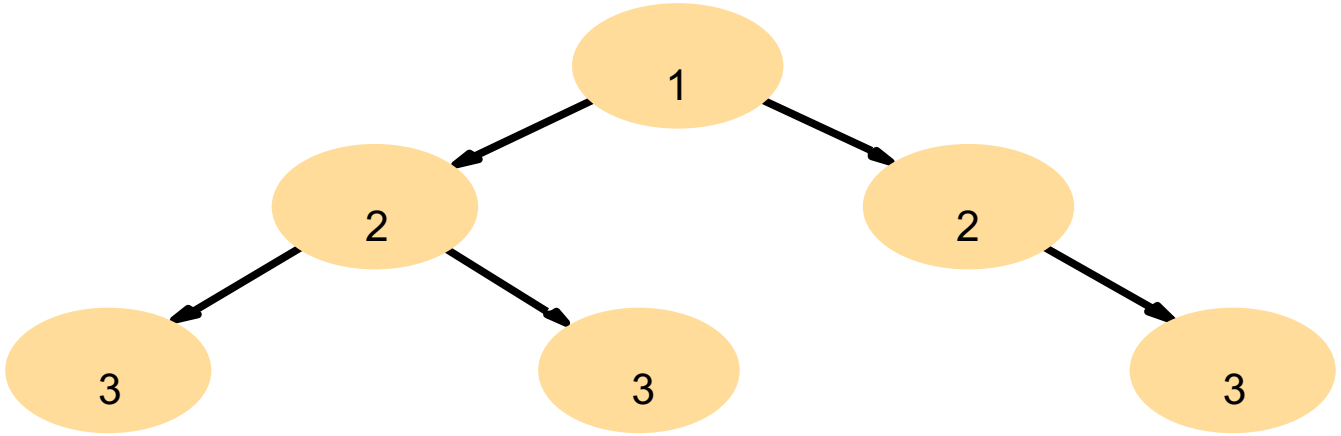
- The time daemon tells everyone how to adjust their clock.



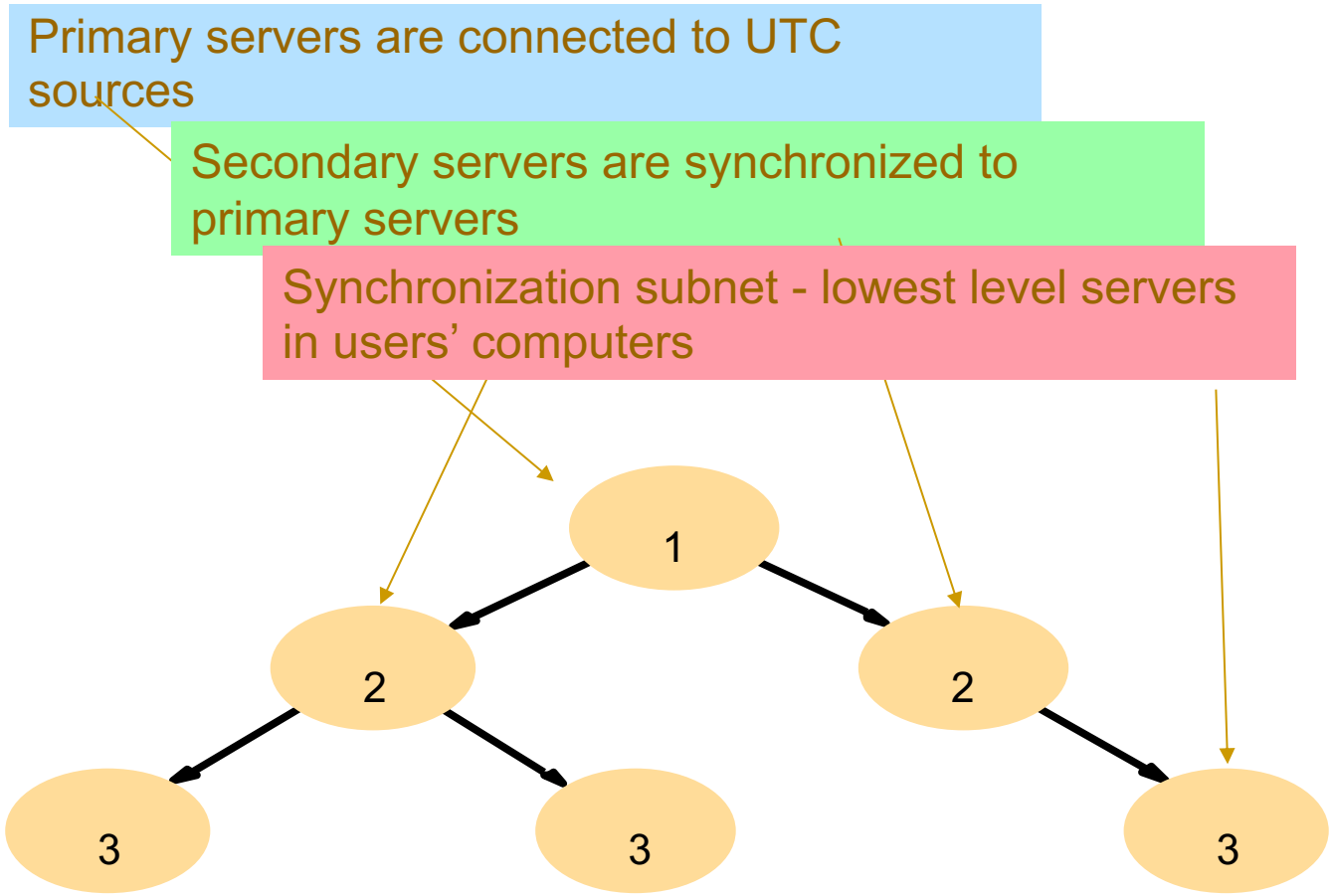
- It takes an average (eliminating any above some average round trip time or with faulty clocks)
- It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)
- Measurements
  - 15 computers, clock synchronization 20-25 millisecs drift rate < $2x10^{-5}$
  - If master fails, can elect a new master to take over (not in bounded time)

Network Time Protocol (NTP)

- It synchronizes clients to UTC
  - Reliability from redundant paths,
  - scalable,
  - authenticates time sources



Network Time Protocol (NTP)



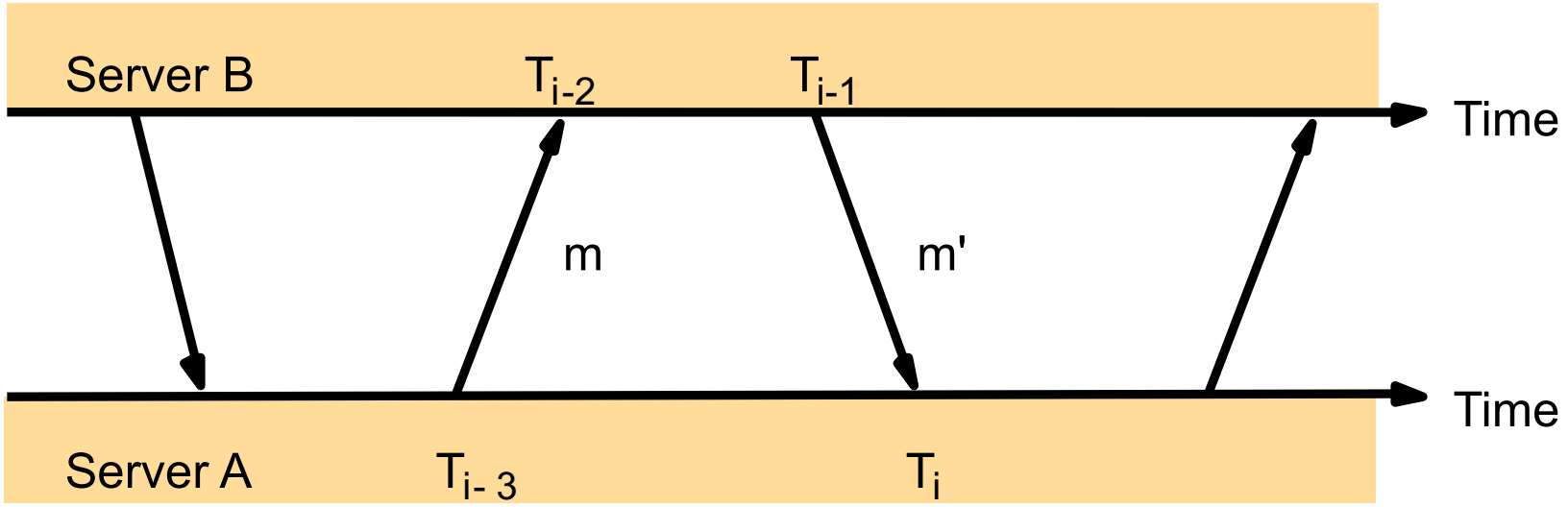
NTP - synchronisation of servers

- The synchronization subnet can reconfigure if failures occur, e.g.
  - a primary that loses its UTC source can become a secondary
  - a secondary that loses its primary can use another primary

NTP - synchronisation of servers

- Modes of synchronization:
  - Multicast
    - A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
  - Procedure call
    - A server accepts requests from other computers (like Cristian's algorithm). Higher accuracy.
  - Symmetric
    - Pairs of servers exchange messages containing time information
    - Used where very high accuracies are needed (e.g. for higher levels)

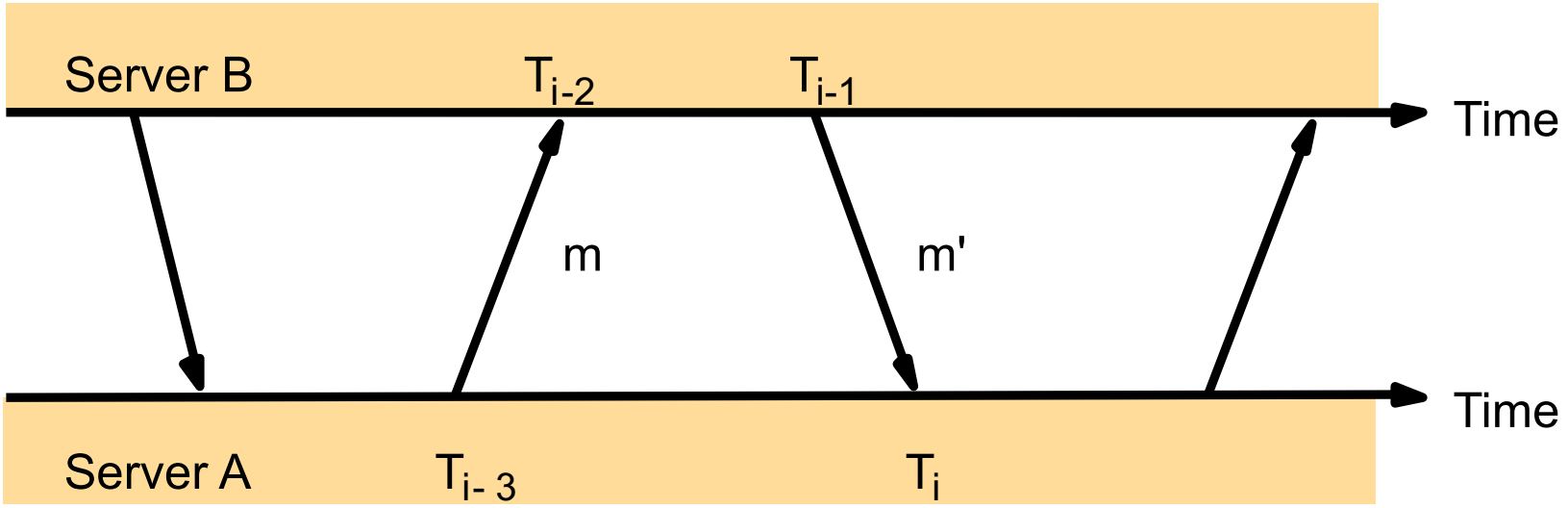
Messages exchanged between a pair of NTP peers



Each message bears timestamps of recent events:

- Local times of *Send* and *Receive* of previous message
- Local times of *Send* of current message

Messages exchanged between a pair of NTP peers



- Recipient notes the time of receipt T_i (we have T_{i-3}, T_{i-2}, T_{i-1}, T_i)
- In symmetric mode there can be a non-negligible delay between messages

Accuracy of NTP

- For each pair of messages between two servers, NTP estimates an offset o, between the two clocks and a delay d_i (total time for the two messages, which take t and t')

  $T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$
- This gives us (by adding the equations) :

  $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$
- Also (by subtracting the equations)

  $o = o_i + (t' - t)/2$ where $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$

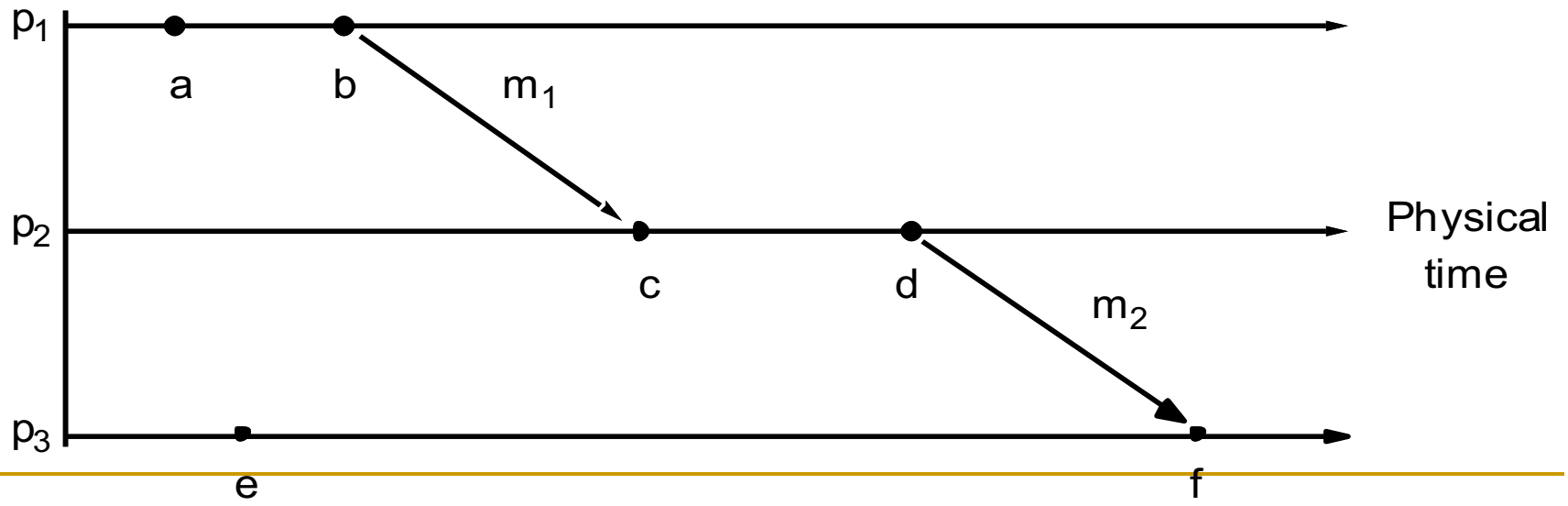
Accuracy of NTP

- Using the fact that t, t'>0 it can be shown that

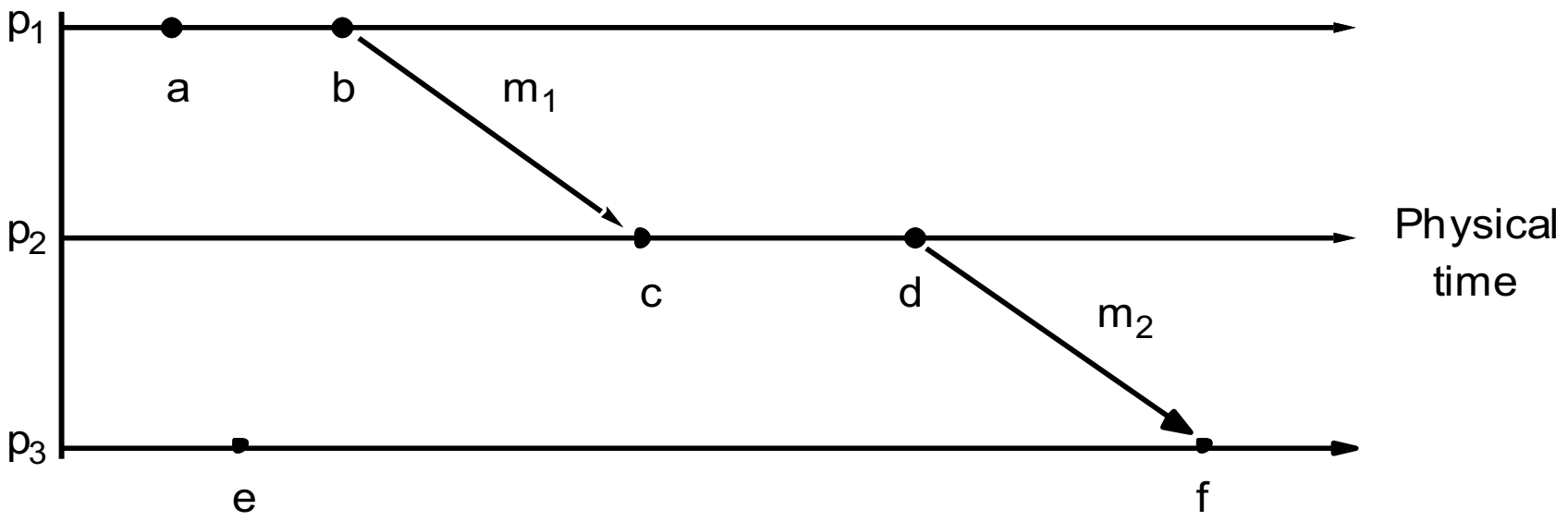
  $o_i - d_i/2 \leq o \leq o_i + d_i/2$.
  - Thus o_i is an estimate of the offset and d_i is a measure of the accuracy
- NTP servers filter pairs $<o_i, d_i>$, estimating reliability from variation, allowing them to select peers
- Accuracy of 10s of millisecs over Internet paths (1 on LANs)

Logical time and logical clocks

- Instead of synchronizing clocks, event ordering can be used
  1. For any two events occurred at the same process p_i, they occurred in the order observed by p_i, that is $\rightarrow_i$
  2. when a message, *m* is sent between two processes, *send(m)* $\rightarrow$ *receive(m)*
  3. The happened before relation is transitive
  4. the happened before relation is the relation of causal ordering



Logical time and logical clocks



$a \rightarrow b$ (at $p1$) $c \rightarrow d$ (at $p2$)

$b \rightarrow c$ because of $m1$

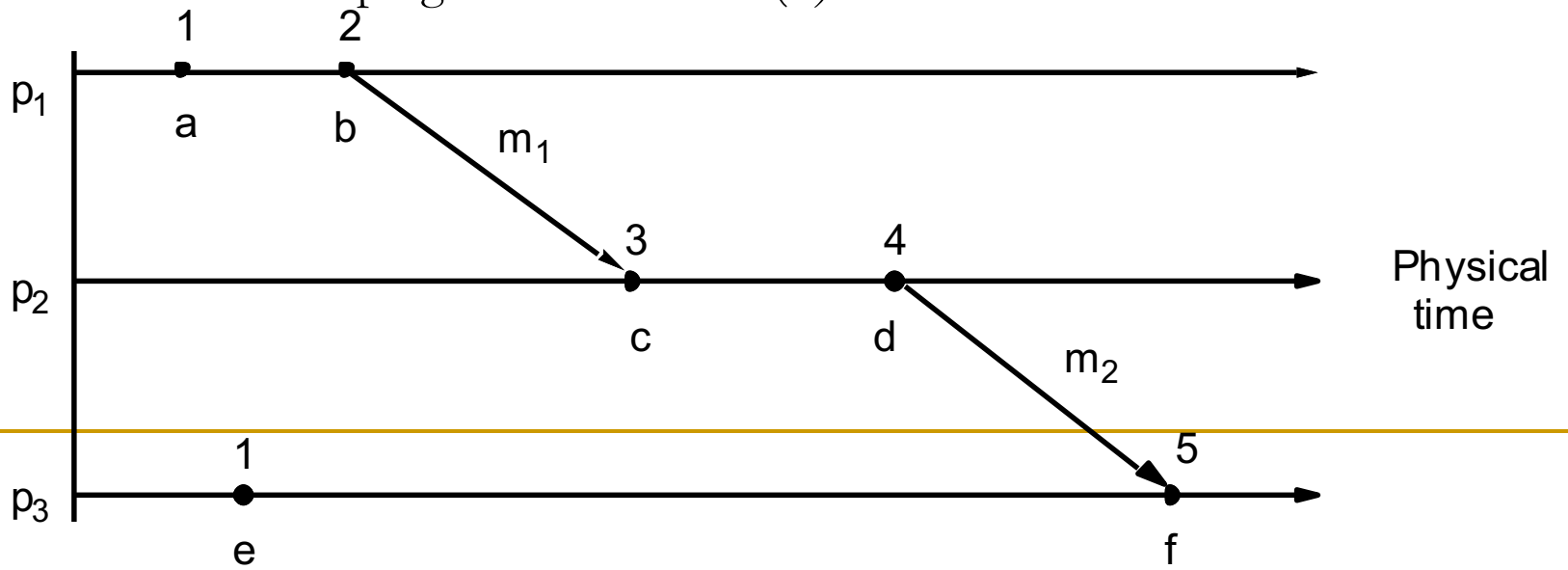
also $d \rightarrow f$ because of $m2$

Not all events are related by $\rightarrow$
consider a and e (different processes and no chain of messages to relate them)

they are not related by $\rightarrow$; they are said to be concurrent; write as $a \parallel e$

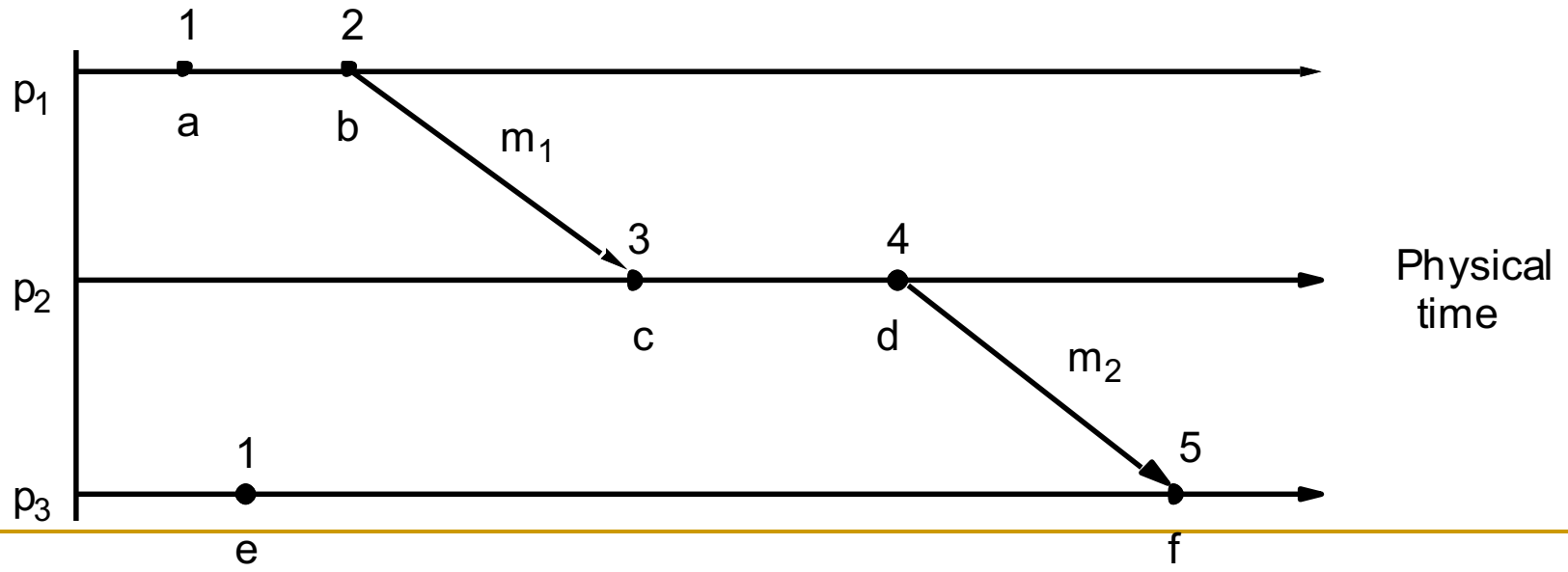
Lamport's logical clocks

- A logical clock is a monotonically increasing software counter. It need not relate to a physical clock.
- Each process p_i has a logical clock, L_i which can be used to apply logical (Lamport) timestamps to events
  - LC1: L_i is incremented by 1 before each event at process p_i
  - LC2:
    - (a) when process p_i sends message m, it piggybacks $t = L_i$
    - (b) when pj receives (m,t) it sets $L_j := max(L_j, t)$ and applies LC1 before timestamping the event *receive* (m)



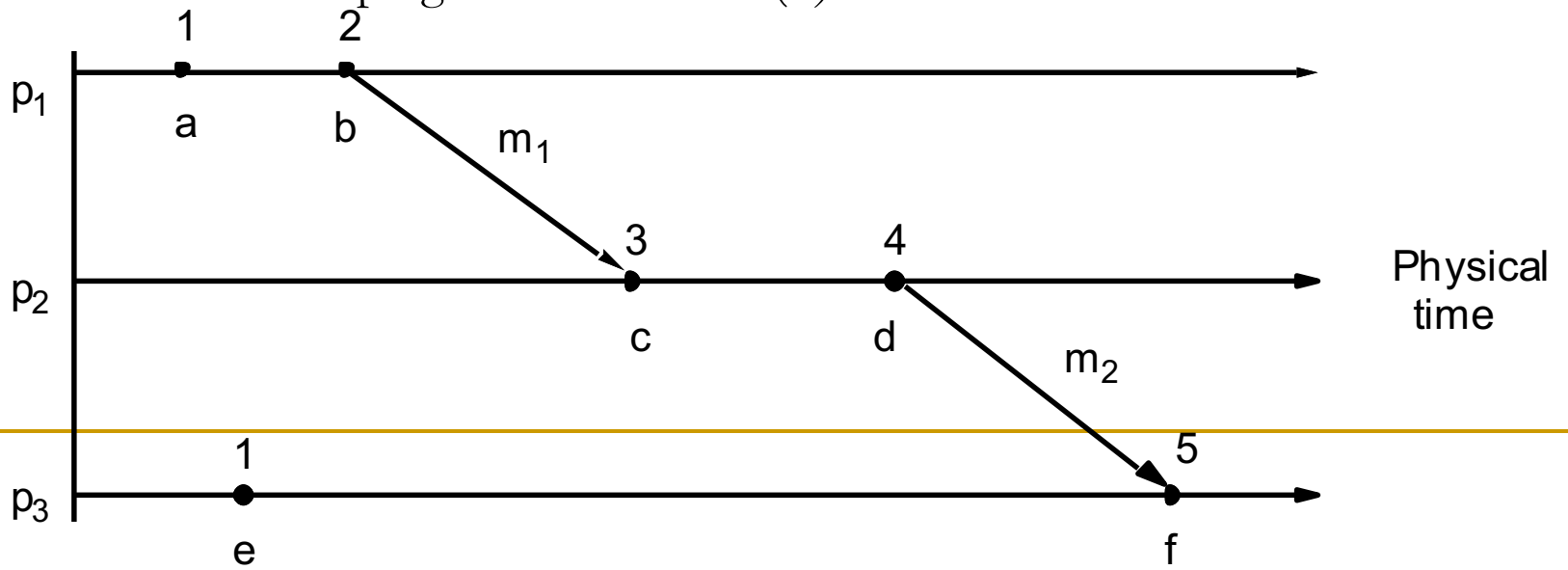
Lamport's logical clocks

- each of p1, p2, p3 has its logical clock initialised to zero,
- the clock values are those immediately after the event.
- for m_1, 2 is piggybacked and c gets $max(0,2)+1 = 3$
- $e \rightarrow e'$ implies $L(e)<L(e')$
- The converse is not true, that is $L(e)<L(e')$ does not imply $e \rightarrow e'$

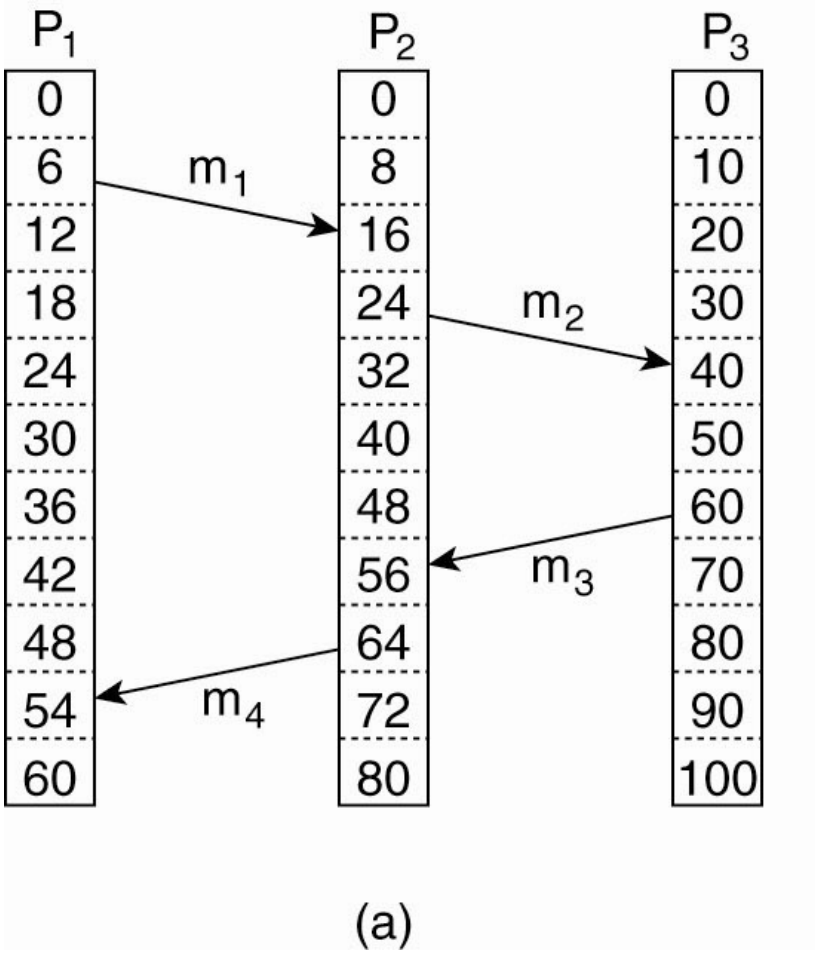


Lamport's logical clocks

- A logical clock is a monotonically increasing software counter. It need not relate to a physical clock.
- Each process p_i has a logical clock, L_i which can be used to apply logical (Lamport) timestamps to events
  - LC1: L_i is incremented by 1 before each event at process p_i
  - LC2:
    - (a) when process p_i sends message m, it piggybacks $t = L_i$
    - (b) when pj receives (m,t) it sets $L_j := max(L_j, t)$ and applies LC1 before timestamping the event *receive* (m)

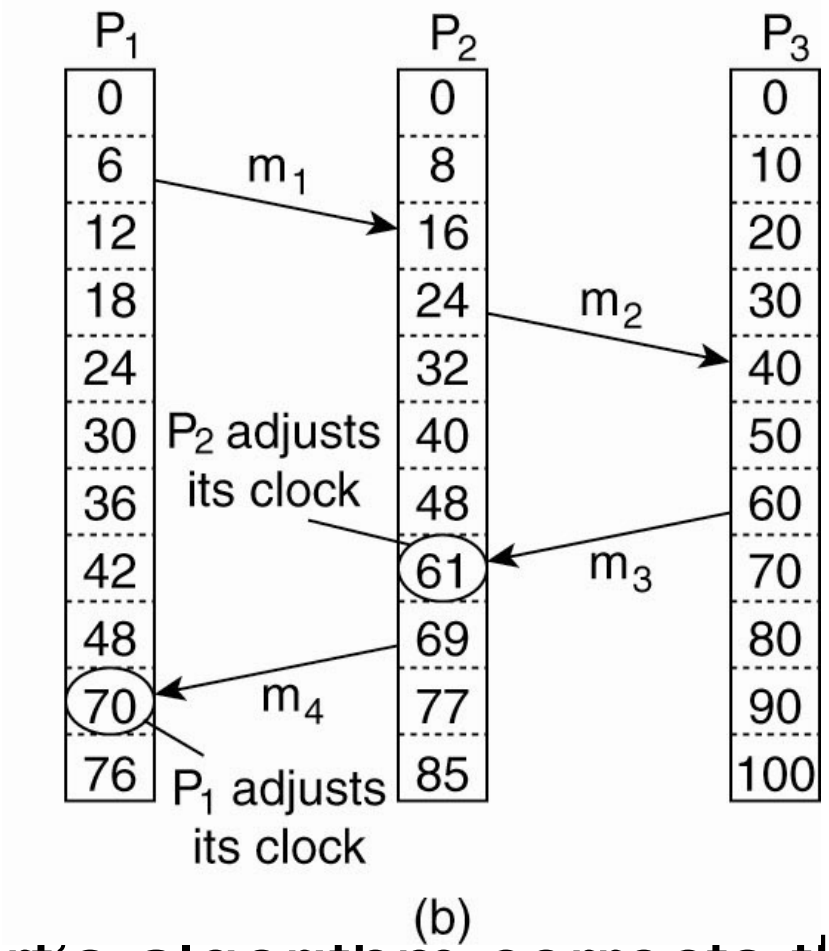


Lamport's Logical Clocks



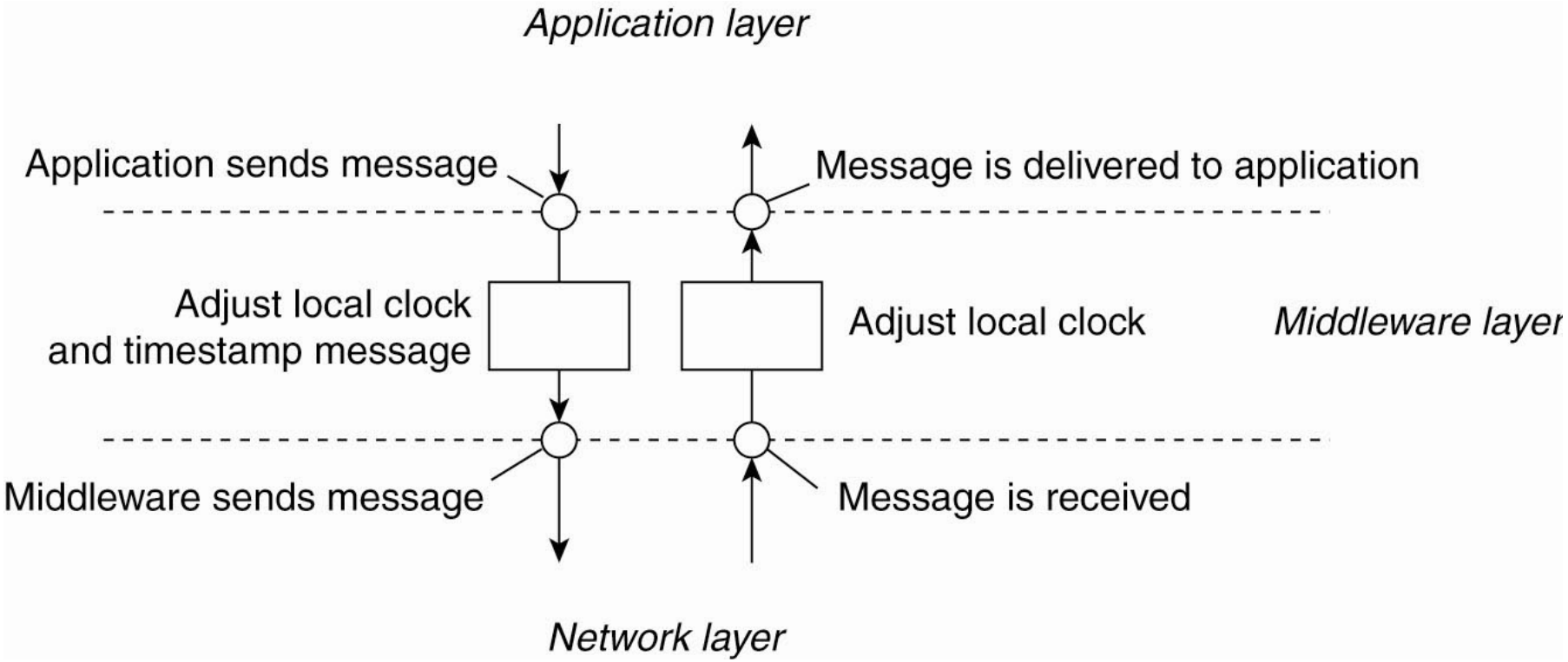
- (a) Three processes, each with its own clock. The clocks run at different rates.

Lamport's Logical Clocks



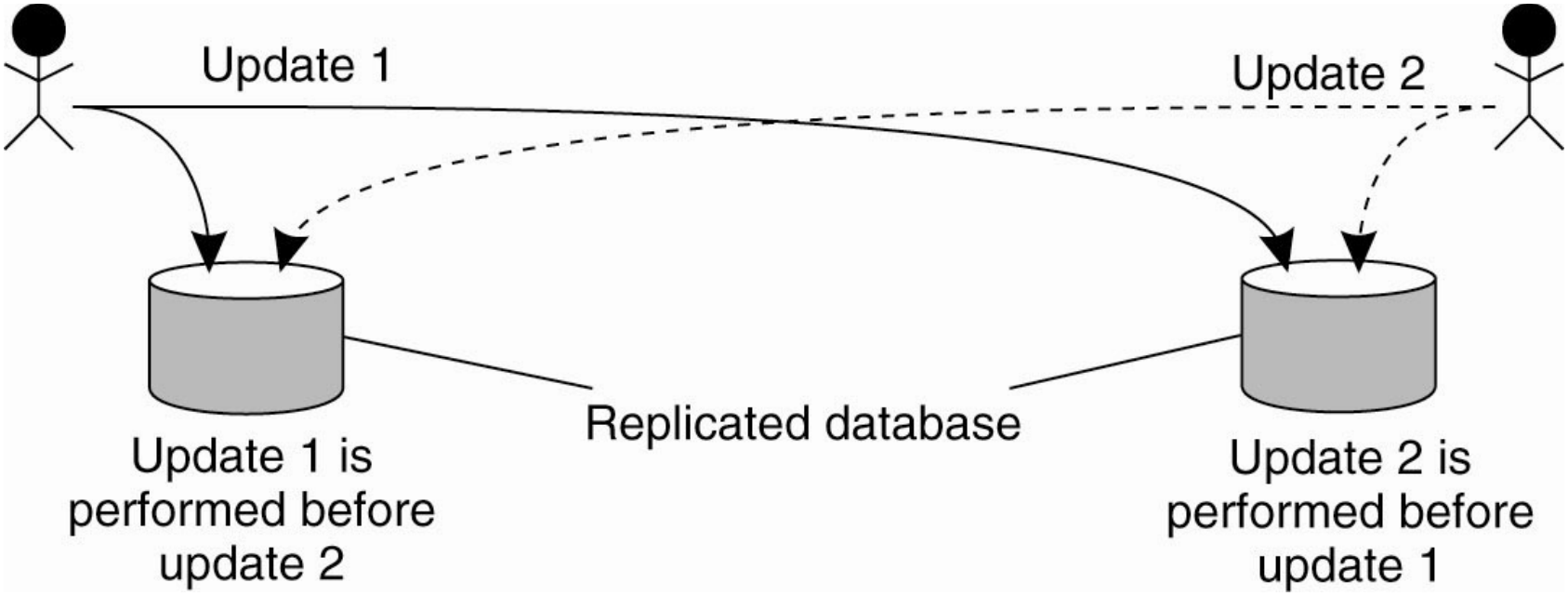
- (b) Lamport's algorithm corrects the clocks.

Lamport's Logical Clocks



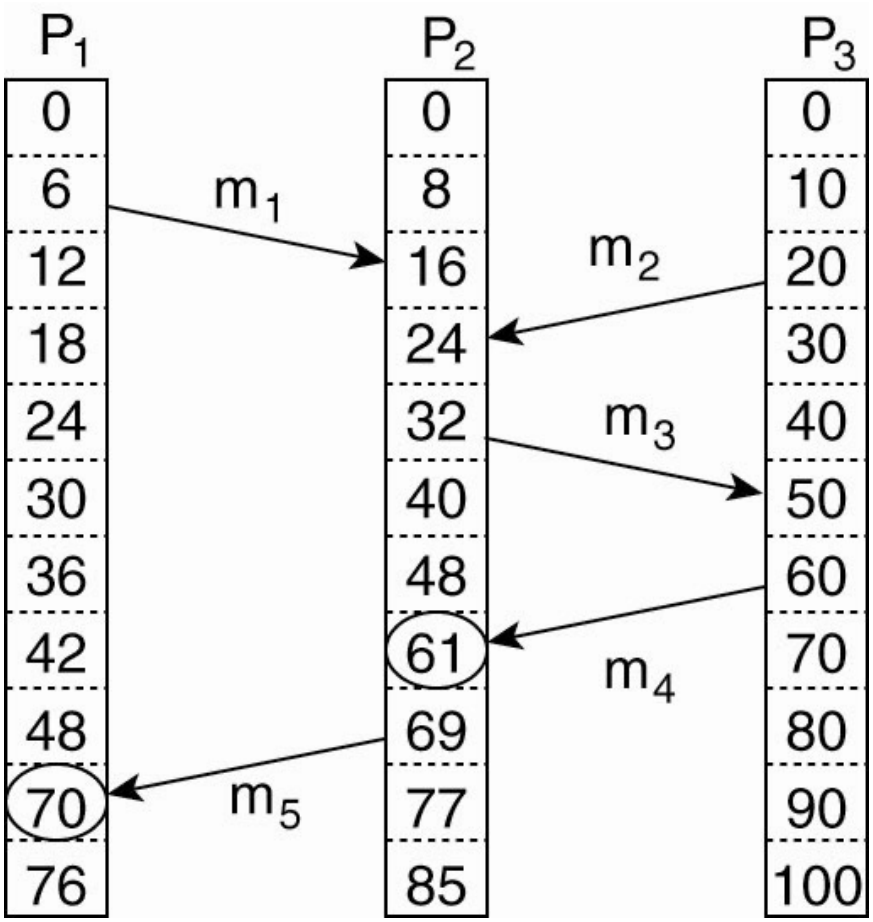
- The positioning of Lamport's logical clocks in distributed systems.

Example: Totally Ordered Multicasting



- Figure 6-11. Updating a replicated database and leaving it in an inconsistent state.

Vector Clocks



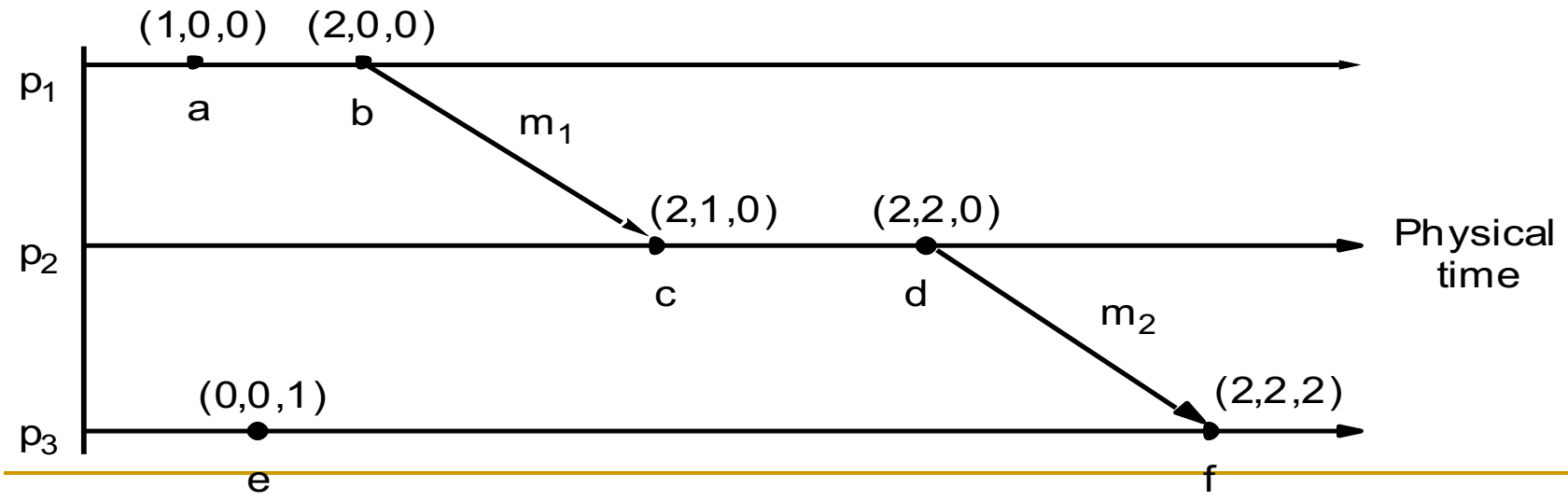
Concurrent message transmission using logical clocks.

Vector clocks

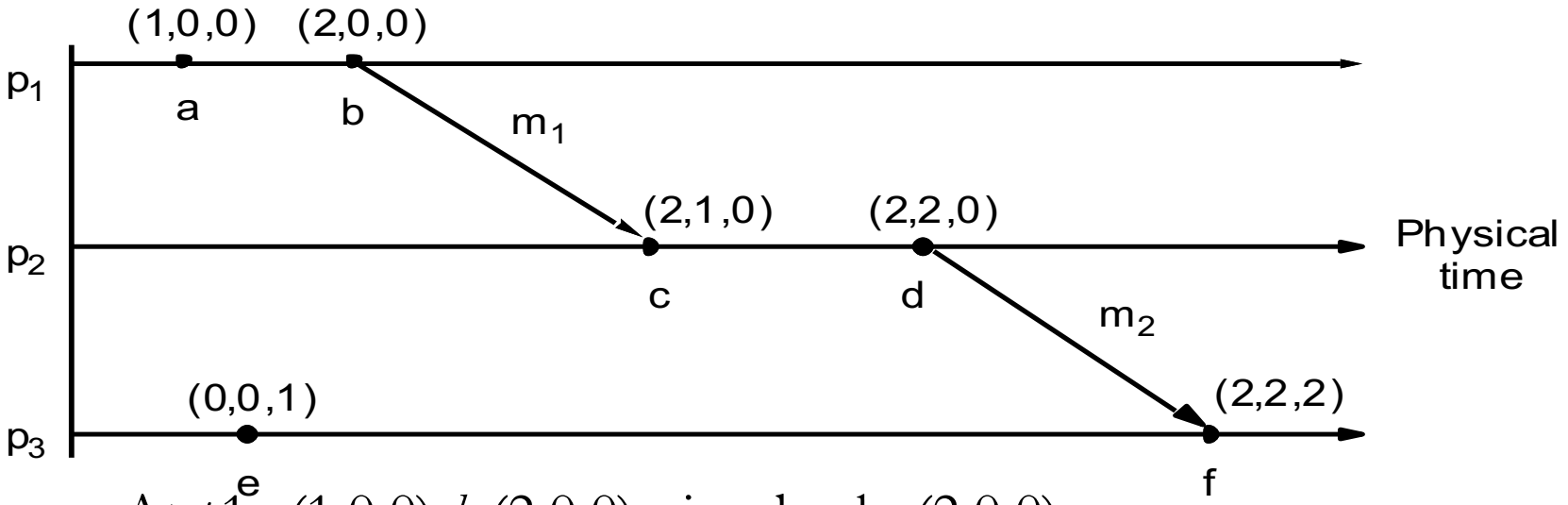
- Vector clocks overcome the shortcoming of Lamport logical clocks ($L(e) < L(e')$ does not imply $e \rightarrow e'$)
- Vector timestamps are used to timestamp local events
- $V_i[i]$ is the number of events that p_i has timestamped
- $V_i[j]$ ($j \neq i$) is the number of events at p_j that p_i has been affected by

Vector clocks

- Vector clock V_i at process p_i is an array of N integers
  - VC1: initially $V_i[j] = 0$ for $i, j = 1, 2, \ldots N$
  - VC2: before p_i timestamps an event it sets $V_i[i] := V_i[i] + 1$
  - VC3: p_i piggybacks $t = V_i$ on every message it sends
  - VC4: when p_i receives (m,t) it sets $V_i[j] := \max(V_i[j], t[j])$ (a *merge* operation)

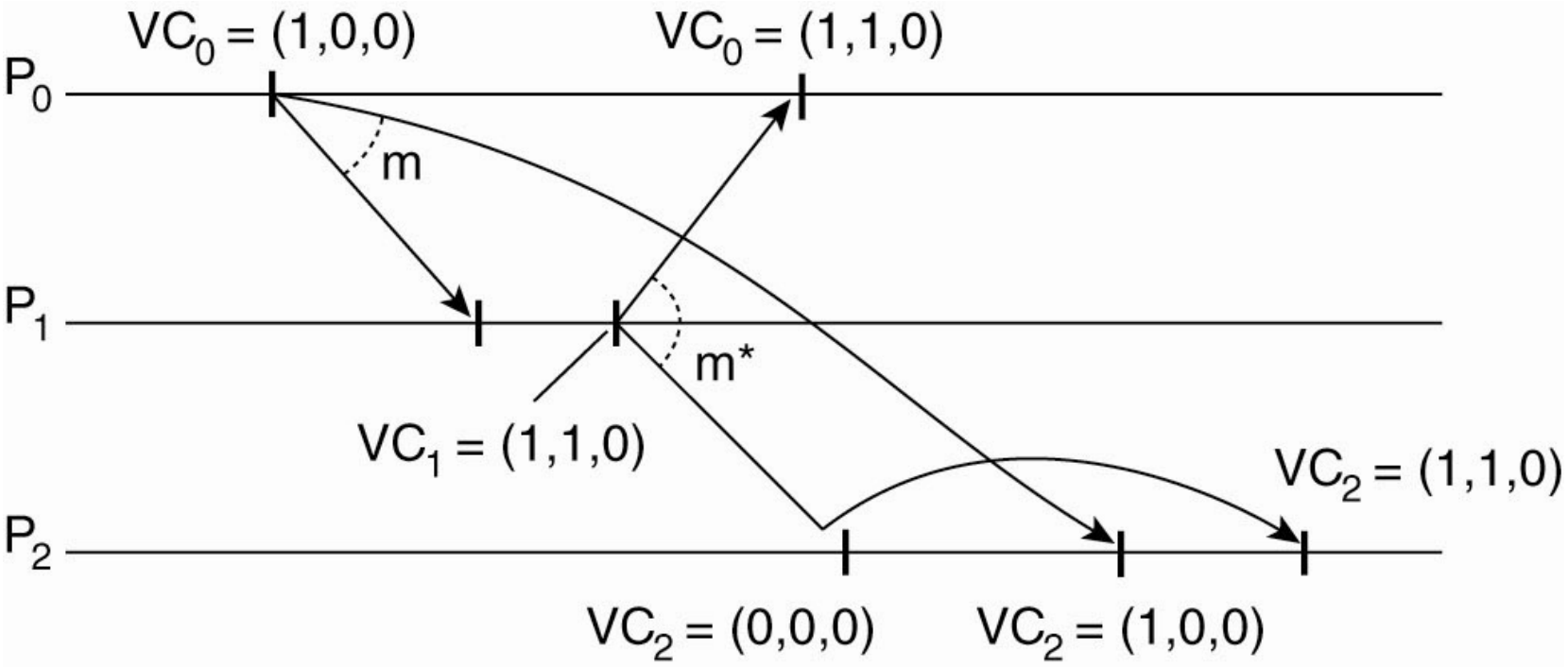


Vector clocks



- At *p*1 *a*(1,0,0) *b* (2,0,0) piggyback (2,0,0) on m_1
- At *p*2 on receipt of m_1 get *max* ((0,0,0), (2,0,0)) = (2, 0, 0), add 1 to own element = (2,1,0)
- Meaning of =, <=, max etc for vector timestamps - compare elements pairwise
- Note that $e \rightarrow e'$ implies $V(e)<V(e')$. The converse is also true.
- $c \mid\mid e$ (parallel) because neither $V(c) <= V(e)$ nor $V(e) <= V(c)$.

Enforcing Causal Communication



Summary on time and clocks in DS

- accurate timekeeping is important for distributed systems.
- algorithms (e.g. Cristian's and NTP) synchronize clocks in spite of their drift and the variability of message delays.
- for ordering of an arbitrary pair of events at different computers, clock synchronization is not always practical.
- the happened-before relation is a partial order on events that reflects a flow of information between them.
- Lamport clocks are counters that are updated according to the happened-before relationship between events.
- vector clocks are an improvement on Lamport clocks,
  - we can tell whether two events are ordered by happened-before or are concurrent by comparing their vector timestamps

…Distributed Systems…

End of lectures