Time in distributed system
18/5/2026

From Chapter 14 of Distributed Systems Concepts and Design,

By G. Coulouris, J. Dollimore, T. Kindberg and G. Blair

Published by Addison Wesley/Pearson Education

Time in distributed system

- Introduction
- Clocks, events and process states
- Synchronizing physical clocks
- Logical time and logical clocks

Time is an important issue in DS

- Need to measure accurately
  - Know at what time a particular event occurred, synchronize clock with external source of time, E.g. auditing in e-commerce
- Algorithms depending on clock synchronization
  - E.g. maintaining consistency of databases

- The problem is based on a limitation to timestamp events at different nodes sufficiently accurately to know the order in which any pair of events occurred.

Time in distributed system

- Introduction
- Clocks, events and process states
- Synchronizing physical clocks
- Logical time and logical clocks

Model of a distributed system

- **£**
  - A collection of N processes $\boldsymbol{p_i}$, i = 1,2, .. N
- s_i
  - The state of p_i
  - E.g. variables
- **Actions of p_i**
  - Operations that transform p_i's state
  - Send or receive message between p_j

Model of a distributed system

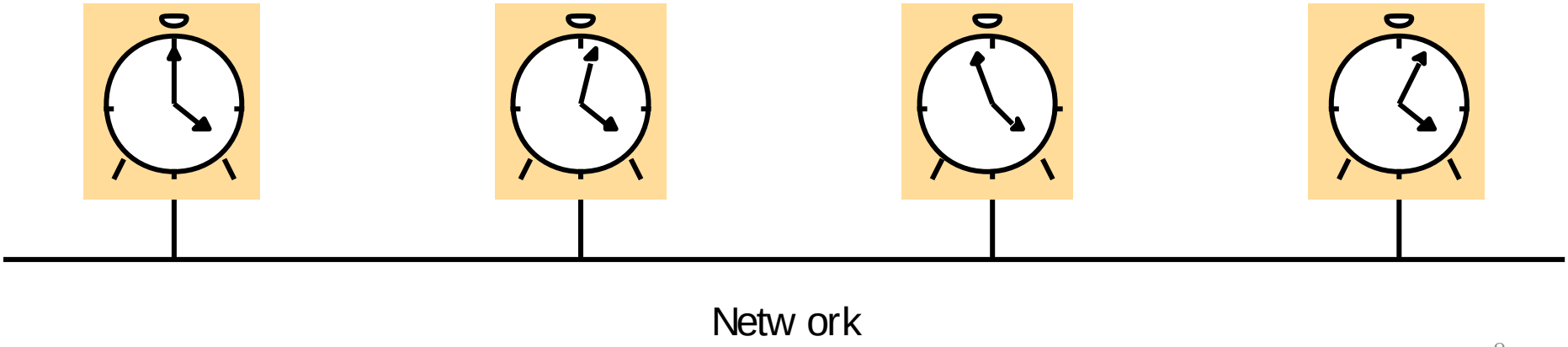
- ***e***
  - Event: occurrence of a single action
- $\rightarrow_i$
  - occur before in p_i , e.g. $e \rightarrow_i e`$
  - Total order of events in p_i
- ***history*(p_i) = h_i** the series of events that take place within p_i ordered by the relation $\rightarrow_I$
  - $h_i = <e_i^0, e_i^1, e_i^2, \ldots>$

Clock in computer

- How to timestamp the events at a process?
- We need **Clock**
- **Clock** is a device that count oscillations occurring in a crystal at a definite frequency
- The OS reads the node's Hardware time: *$H_i(t)$ to produce a* Software time *$C_i(t)$.*
- We use this value to *Timestamp* any event *at* ***p_i***
  - Assign a date and time to the event

Clock skew and clock drift

- Clock drift
  - Crystal oscillate at different rate
  - Clock drift can not be avoided
- Clock skew
  - The instantaneous difference between the readings of any two clocks



Time in distributed system

- Introduction
- Clocks,events and process states
- Synchronizing physical clocks
- Logical time and logical clocks

External & Internal synchronization

- *In order to know at what time events occur at the processes in our DS it is necessary to synchronize the process' clocks*
- C_i : p_i's clock
- *I*: an interval of real time
- External synchronization
  - For a synchronization bound $D > 0$, and for a source S of time, $|S(t)-C_i(t)| < D$, for $i = 1, 2, \ldots N$ and for all real times t in I
  - Clocks C_i are *accurate* to within the bound D

External & Internal synchronization (2)

- Internal synchronization
  - For a synchronization bound $D > 0$,

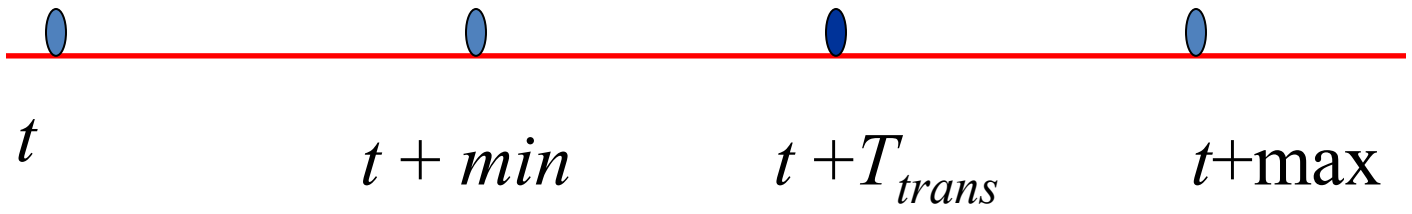
    $|C_i(t)-C_j(t)| < D$ for $i, j = 1,2, \ldots N$, and for all real times t in I
  - Clocks C_i *agree* within the bound D
- If *accurate* to within D, then *agree* within $2D$

Synchronization in a synchronous system

- Internal synchronization between two processes in a synchronous distributed system.
- Protocol
  - Sender: send $M(t)$
  - Receiver: set time to $t + T_{trans}$
- Bounds are known in synchronous system
  - ❖ min < T_{trans} < max (constant)
- So, set T_{trans} = (min+max) / 2
  - Receiver's clock = t + (min+max) / 2

Synchronization in a synchronous system(2)

- Clock skew between sender and receiver
  - ❖ (max – min) / 2

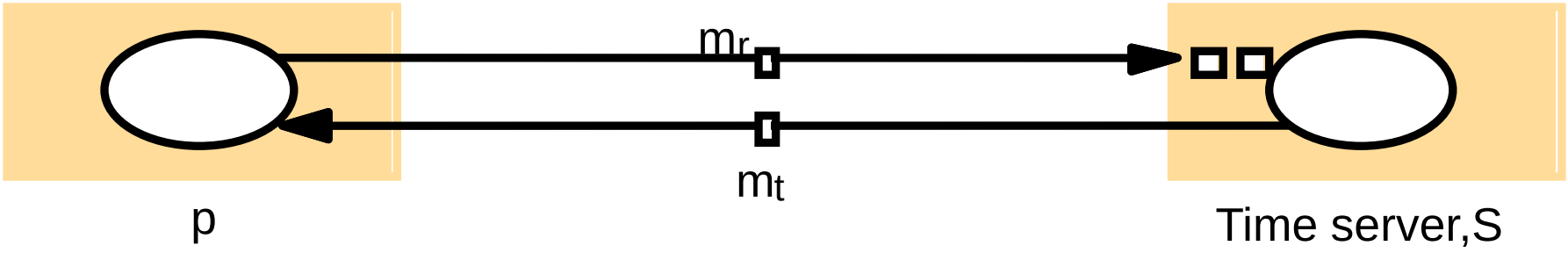


Cristian's method of synchronizing clocks

- Cristian use a time server to synchronize computers externally
- Applying circumstance
  - C/S
  - Round-trip time is short compared with the required accuracy

Cristian's method of synchronizing clocks

- Protocol
  - A process *p requests the time in a message* m_r*, and receives the time value t in a* message m_t *(t is inserted in* m_t *at the last possible point before transmission from S'*).
  - Process *p records the total round-trip time* T_{round} *taken to send the request* m_r *and receive the reply* m_t .
  - m_r, $m_t(t)$, T_{round}
  - Estimated time: t in $m_t + T_{round}/2$



The Berkeley algorithms

- Internal synchronization
- A coordinator computer is chosen to act as the master.

1. The master periodically polls the slaves' clocks (computers whose clocks are to be synchronized)
2. The slaves send back their clock values to it.
3. The master estimates the slaves' clocks by round-trip time
   - Similar to Christian's algorithm
4. The master averages the slaves' clock values (including its own clock's reading).
   - Cancel out the individual clock's tendencies to run fast or slow

The Berkeley algorithms (2)

5. The master sends back to the slaves the amount that the slaves' clocks should adjust by
   - Positive or negative value
6. Slave adjust its clock

- The Berkeley algorithm eliminates readings from faulty clocks.
  - the master takes a fault-tolerant average (a subset is chosen of clocks that do not differ from one another by more than a specified amount, and the average is taken of readings from only these clocks.
- If the master fail, then another can be elected to take over and function exactly as its predecessor.

The Network Time Protocol

- NTP defines an architecture for a time service and a protocol to distribute time information over the Internet.

Design aims:

- External synchronization
  - Enable clients across the Internet to be synchronized accurately to UTC (an international standard for timekeeping)
- Reliability
  - Can survive lengthy losses of connectivity
    - Redundant server & redundant path between servers

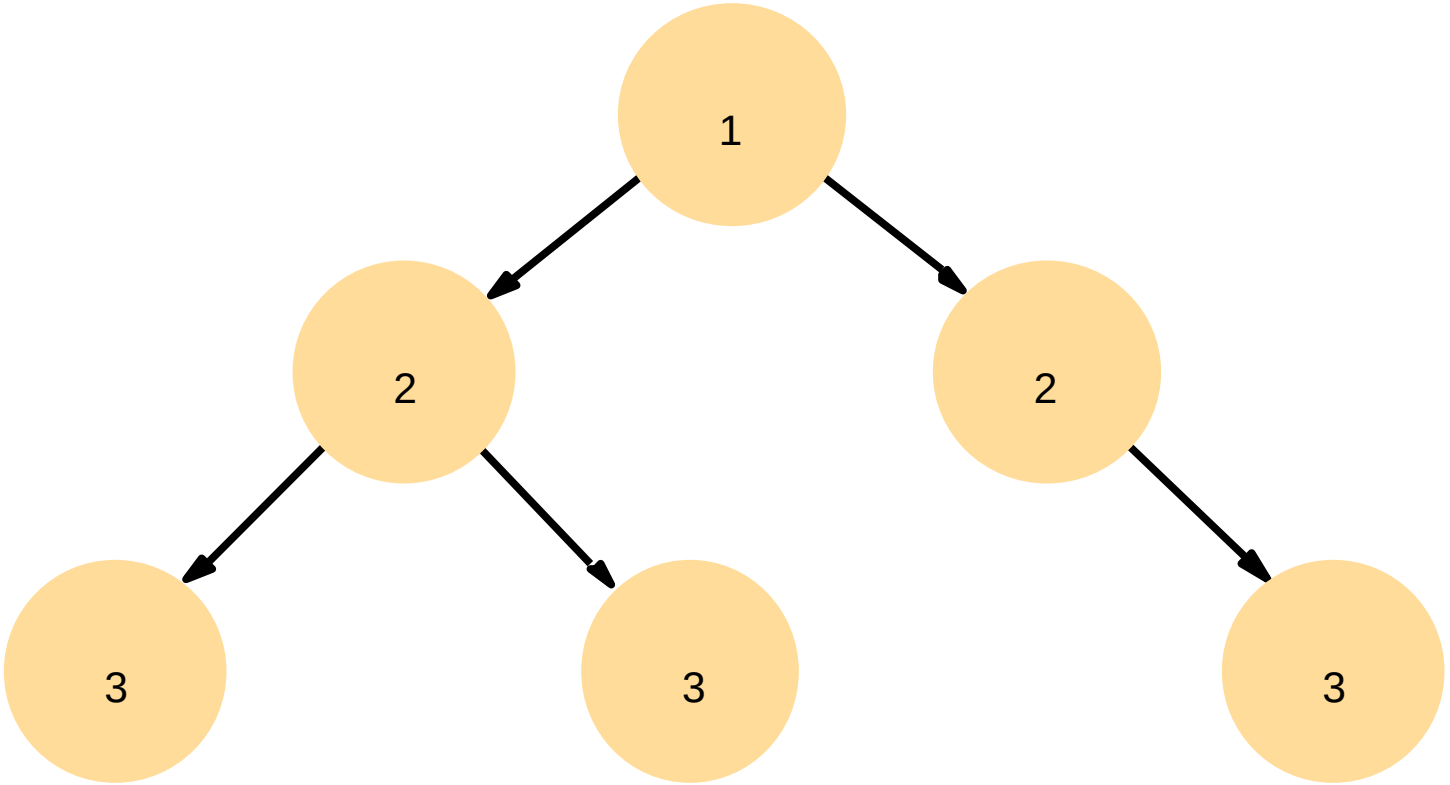
Design aims of Network Time Protocol (2)

- Scalability
  - Enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers (large numbers of clients and servers)
- Security
  - Protect against interference with the time service, whether malicious or accidental

Network Time Protocol Architecture

- The NTP service is provided by a network of servers located across the Internet.
- Primary servers are connected directly to a time source such as a radio clock receiving UTC;
- secondary servers are synchronized, ultimately, with primary servers.
- The servers are connected in a logical hierarchy called a synchronization subnet (see Figure), whose levels are called strata.
- Primary servers occupy stratum 1: they are at the root.
- Stratum 2 servers are secondary servers that are synchronized directly with the primary servers;
- Stratum 3 servers are synchronized with stratum 2 servers,and ...
- The lowest-level (leaf) servers execute in users' workstations.

Network Time Protocol Architecture



Note: Arrows denote synchronization control, numbers denote strata.

- Reconfigure when servers become unreachable

NTP servers synchronization modes

1. Multicast mode
   - Intend for use on a high speed LAN
     - One or more servers periodically multicasts the time to the servers running in other computers connected by the LAN, which set their clocks assuming a small delay.
   - Low accuracy
   - Sufficient for many purposes
2. Procedure-call mode
   - Similar to Christian's algorithm
     - In this mode, one server accepts requests from other computers, which it processes by replying with its timestamp (current clock reading).
   - Higher accuracy than multicast

NTP servers synchronization modes

3. Symmetric mode
   - is intended for use by the servers that supply time information in LANs and by the higher levels (lower strata) of the synchronization subnet
   - A pair of servers operating in symmetric mode exchange messages bearing timing information.
   - The highest accuracy

Time in distributed system

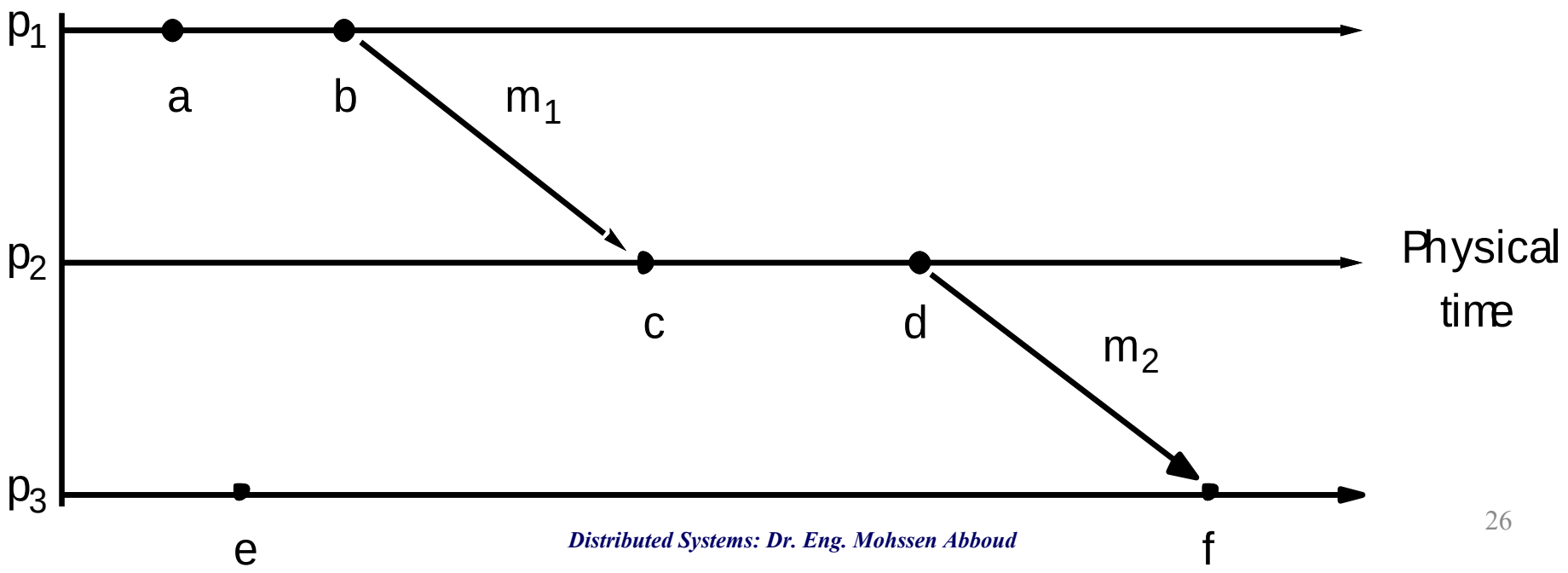
- Introduction
- Clocks, events and process states
- Synchronizing physical clocks
- Logical time and logical clocks

Happen-before relation

- From the point of view of any single process, events are ordered uniquely by times shown on the local clock
- since we cannot synchronize clocks perfectly across a distributed system, we cannot use physical time to find out the order of any arbitrary pair of events occurring within it.
- $\rightarrow$
  - HB1: If $\exists$process p_i: $e \rightarrow_i e`$, then $e \rightarrow e`$
  - HB2: For any message m, send(m) $\rightarrow$receive(m)
  - HB3: IF e, e`and e`` are events such that
    $e \rightarrow e`$ and $e` \rightarrow e``$, then $e \rightarrow e``$
  - *Causal ordering*

Happen-before relation

- The relation $\rightarrow$ is illustrated for the case of three processes, p1 , p2 and p3 , in the Figure
- It can be seen that a $\rightarrow$ b, c $\rightarrow$ d, b $\rightarrow$ c d $\rightarrow$ f.
- we may also say that, a $\rightarrow$ f.



Happen-before relation

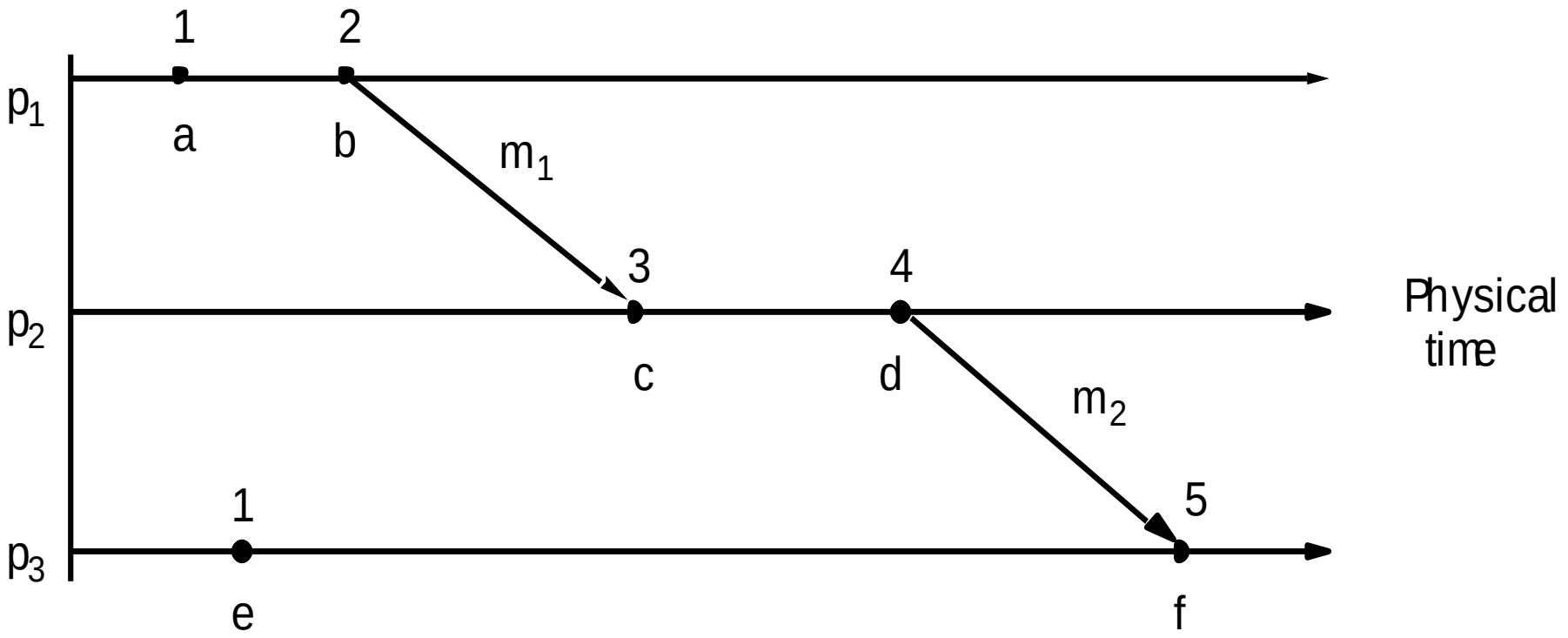
- It can also be seen from the Figure that not all events are related by the relation $\rightarrow$
- For example, a and e, since they occur at different processes, and there is no chain of messages intervening between them.
- We say that events such as a and e that are not ordered by $\rightarrow$ are concurrent and write a || e

Lamport timestamps algorithm

- Lamport invented a simple mechanism by which the happened before ordering can be captured numerically, called a logical clock
- logical clock is a monotonically increasing software counter L_i , whose value need bear no relationship to any physical clock.
- 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)

Lamport timestamps algorithm (2)

- $e \rightarrow e` \Rightarrow L(e) < L(e`)$
- $L(e) < L(e`) \Rightarrow e \rightarrow e`$ or $e \| e`$



Totally ordered logical clocks

Assumption

T_i : local timestamp of e that is an event occurring at p_i

T_j : local timestamp of e` that is an event occurring at p_j

Define the timestamps of e, e` are (T_i, i), (T_j, j)

Define $<$

$(T_i, i) < (T_j, j)$ if $T_i < T_j$, or $T_i = T_j$ *and* $i < j$

- Useful in some applications

Vector Clocks - algorithm

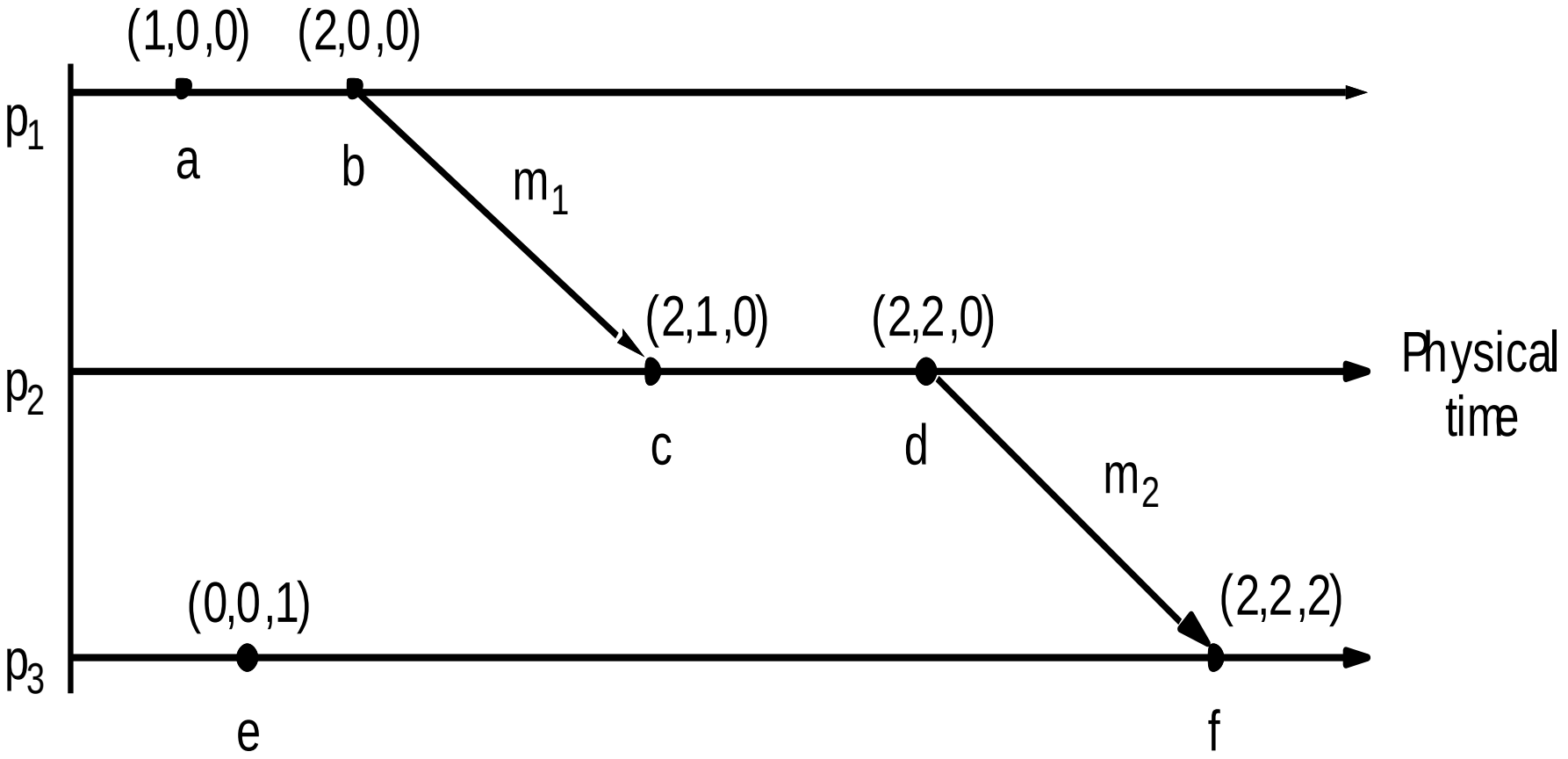
- Mattern and Fidge developed vector clocks to overcome the fact that from L(e) < L(e`)

  we cannot conclude that e $\rightarrow$ e`
- Each process p_i keeps a vector clock V_i
- 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 $j=1,2\ldots,N$

Vector Clocks - significance

- Compare vector timestamps
  - *V = V` iff V [j] = V`[j]* for *j = 1,2…, N*
  - *V <= V` iff V [j] <= V`[j]* for *j = 1,2…, N*
  - *V < V` iff V [j] < V`[j]* for *j = 1,2…, N*
  - *Otherwise V<>V'*
- Let V(e) be the vector timestamp applied by the process at which e occurs
  - V(e) < V(e`) $\Leftrightarrow$ e$\rightarrow$e`,
  - V(e) <> V(e`) $\Leftrightarrow$ e||e`

Vector Clocks - example



Vector Clocks - example

- The figure shows the vector timestamps of the events.
- It can be seen, for example, that V(a) < V(f), which reflects the fact that a$\rightarrow$f.
- Similarly, we can tell when two events are concurrent by comparing their timestamps.
- For example, that c||e can be seen from the facts that neither V(c) ≤ V(e), nor V(e) ≤ V(c),

Vector Clocks - example