Time in Distributed Systems

Brian Nielsen
bnielsen@cs.aau.dk

Needs for precision time

- Stock market buy and sell orders
- Secure document timestamps (with cryptographic certification)
- Distributed network gaming and training
- Aviation traffic control and position reporting
- Multimedia synchronization for real-time teleconferencing
- Event synchronization and ordering
- Network monitoring, measurement and control

Physical Time

1. The sun

Today: 1 sec ~ 1 day / 86400
but rotation of earth slows down

2. An Atom

State transitions in atoms (defined by BIH in Paris)
1 sec = time a cesium atom needs for 9 192631 770 state transitions*

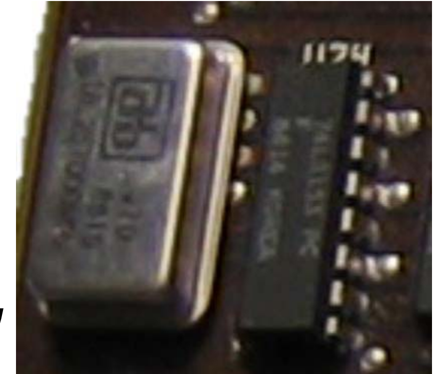
* TAI (International Atomic Time)

- A TAI-day is about 3 msec shorter than a day
- BIH (Bureau International de l'Heure) inserts 1 sec, if the difference between a day and a TAI-day is more than 800 msec
- Definition of **UTC = universal time coordinated**, being the base of any international time measure.

UTC broadcasts

- UTC-signals come from **shortwave radio** broadcasting stations or from **satellites** (GEOS, GPS) with an accuracy of:
    - 1.0 msec (broadcasting station)
    - 1.0 μsec (GPS)
    - >> 1ms (UTC available via phone line)
- Receivers are available commercially and can be connected to PCs

Computer Clocks



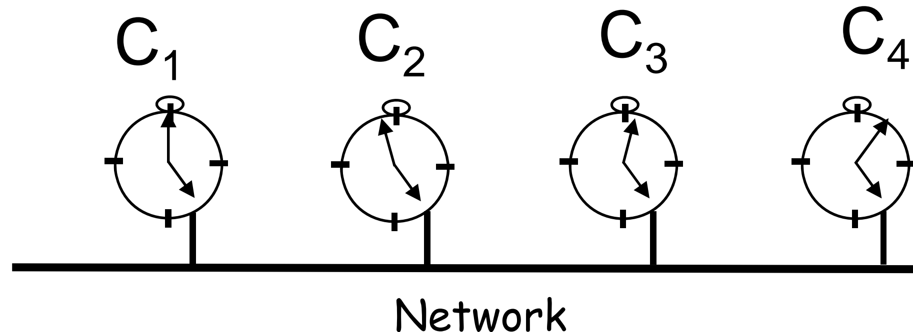
- Each node has its own private *physical clock !*
- Physical clocks are HW devices that count oscillations of a quartz.
- After a specified number of oscillations, the clock increments a register, thereby adding one *clock-tick* to a counter the represents the passing of time: **$H_i(t)$.**
- **Resolution:** Period between clock updates
- The OS maintains SW Clock by scaling and adding an offset to it:

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

- $C_i(t)$ approximates the physical time t at process p_i. $C_i(t)$ may be implemented by a 64-bit word, representing nanoseconds that have elapsed at time t.
- Successive events can be distinguished if the *clock resolutions* is smaller that the time interval between the two events.

Drift and Skew

Clock Skew



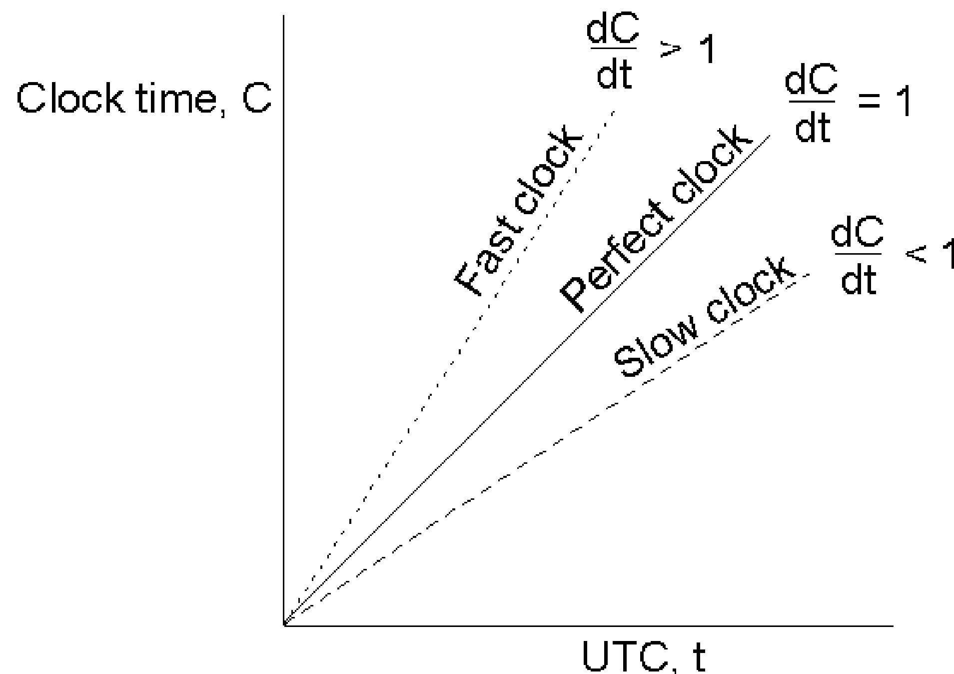
- Computer clocks, like any other clocks tend not to be in perfect agreement !!
- ***Clock skew (offset)*: the difference between the times on two clocks $|C_i(t) - C_j(t)|$**
- ***Clock drift* : they count time at different rates**
  - Ordinary quartz clocks drift by ~ 1sec in 11-12 days. (10^{-6} secs/sec).
  - High precision quartz clocks drift rate is ~ 10^{-7} or 10^{-8} secs/sec
  - Differences in material, *Temperature variation.*

Clock Drift

- Clock makers specify a maximum drift rate ρ (rho) sec/sec.
- By definition

  $1-\rho \leq dC/dt \leq 1+\rho$

  where C(t) is the clock's time as a function of the real time



- Max skew δ:Resynchronize at least every $\delta/2\rho$ seconds

Internal/External Synchronization

External synchronization

- synchronization of process' clocks C_i with an authoritative external source S.
- Let $\delta > 0$ be the synchronization bound and S be the source of UTC.
- Then $|S(t) - C_i(t)| < \delta$ *for* $i=1,2,\ldots,N$ and for all real times t.
- We say that clocks C_i are ***accurate*** within the bound of δ

Internal synchronization

- synchronization of process' clocks C_i with each other.
- Let $\delta > 0$ be the synchronization bound and C_i and C_j are clocks at processes p_i and p_j, respectively.
- Then $|C_i(t) - C_j(t)| < \delta$ *for* $i,j=1,2,\ldots,N$ and for all real times t.
- We say that clocks C_i, C_j ***agree*** within the bound of δ

Note that clocks that are internally synchronized are not necessarily externally synchronized. i.e., even though they agree with each other, the drift collectively from the external source of time.

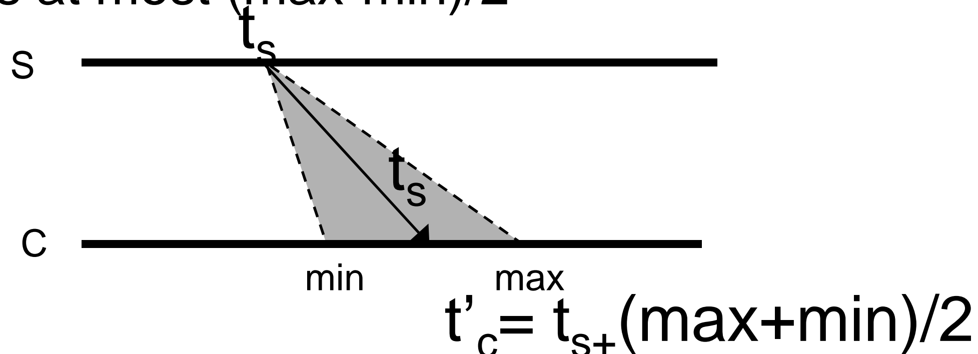
Synchronization in a ***synchronous*** system

In a ***synchronous system*** we have:

- known upper *(max)* and lower *(min)* bound for communication delay,
- known maximum clock drift,
- known maximum time taken for each computational step.

We synchronize by:

- time server sends its local time *t* to a client,
- Ideally, client sets clock to $t_s + T_{tran}$ (Unknown!)
- the client sets its local clock to $t_s + (max+min)/2$.
- *Skew* is at most (max-min)/2



Synchronization in an *asynchronous* system

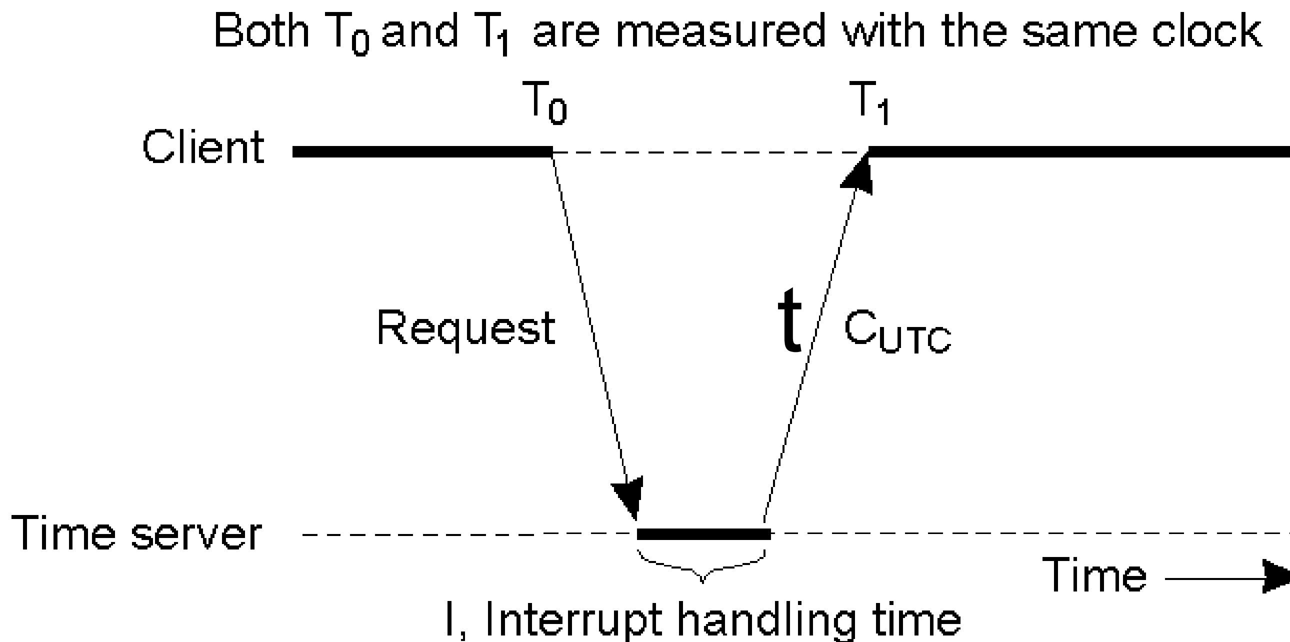
- Christians algorithm.
- The Berkeley algorithm.
- Network time protocol (NTP).

Cristian’s Algorithm

- A node is a time server TS (presumably with access to UTC). How can the other nodes be sync’ed?
- Periodically, at least every $\delta/2\rho$ seconds, each machine sends a message to the TS asking for the current time and the TS responds.

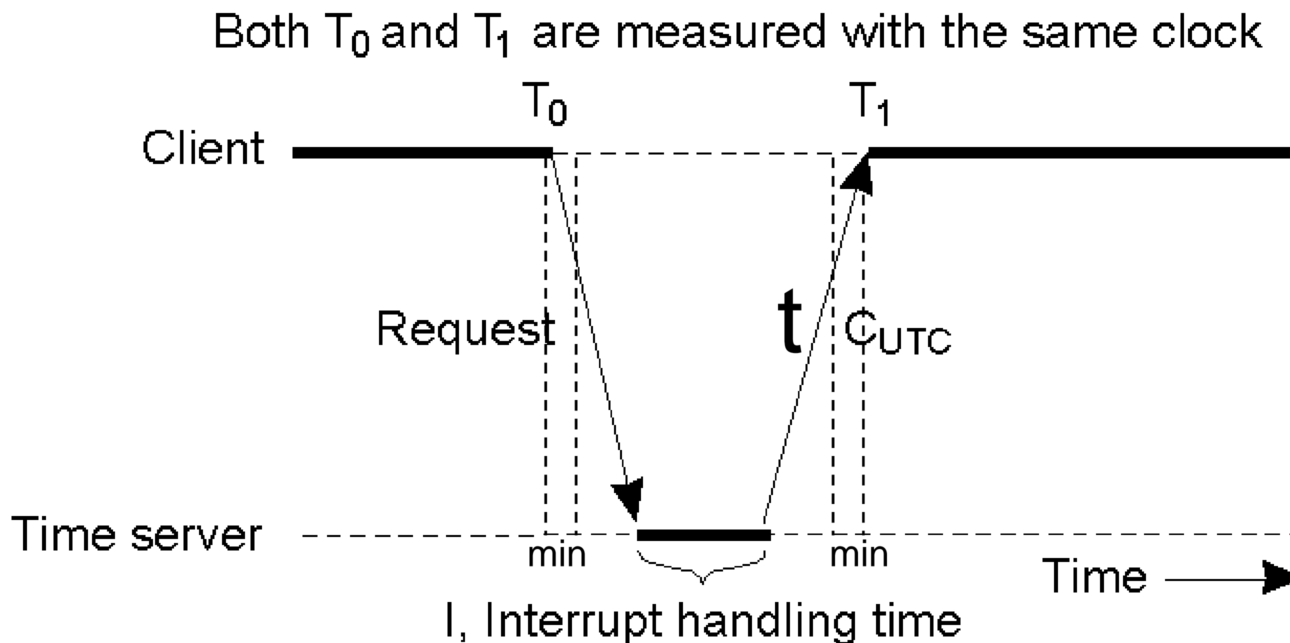
Christians algorithm

- Client *p* sends request (m_r) to time server *S*,
- *S* inserts its time *t* immediately before reply (m_t) is returned,
- *p* measures how long it takes $(T_{round} = T_1 - T_0)$ from m_r is send to m_t is received
- *p* sets its local clock to $t + T_{round}/2$.



Accuracy of Christians algorithm

- Assume min = minimal message delay
- t is in the interval [T_0+min, T_1-min]
- Uncertainty on t = T_{round}-2min
- Estimated Accuracy= T_{round}/2-min

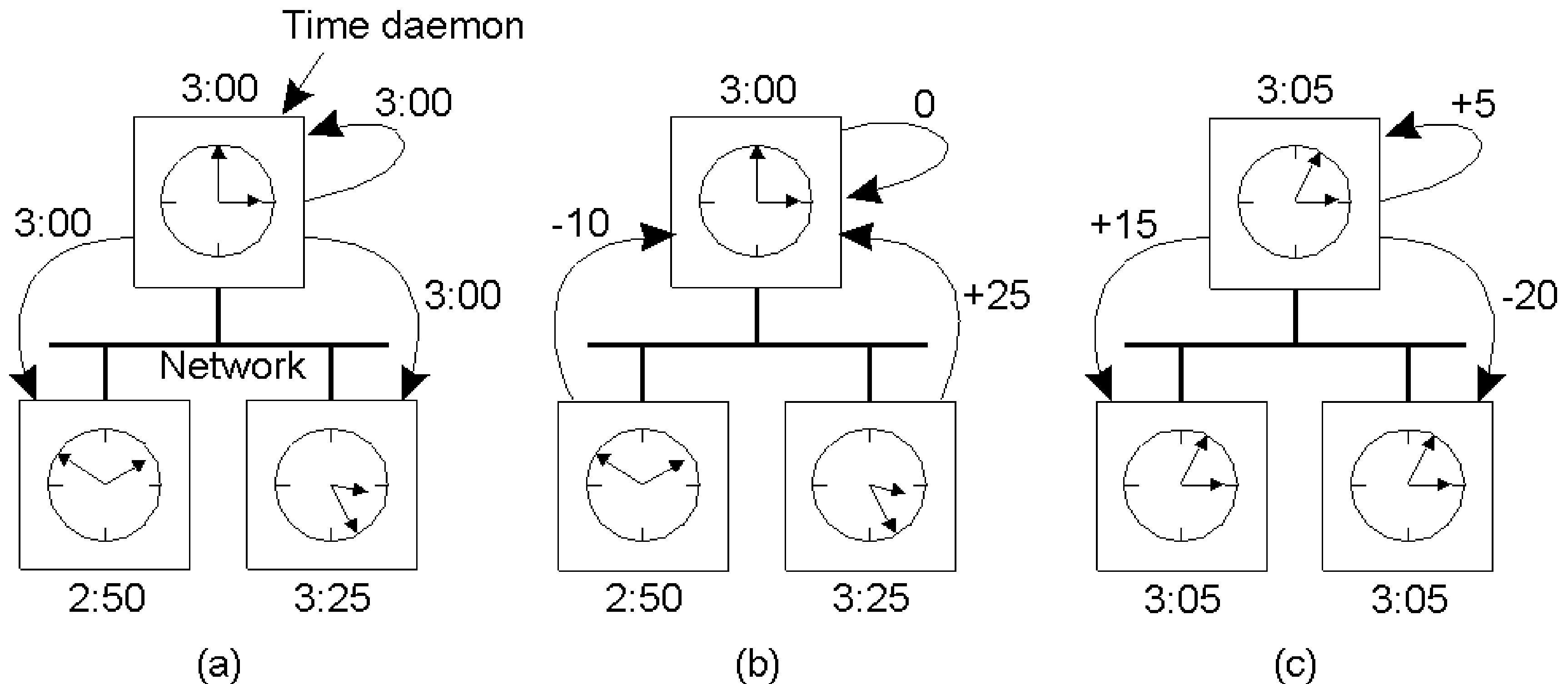


Cristian's Algorithm

- Monotonicity:
  - Jumps in time backward not permitted
  - Jumps forward may be confusing
  - Receiver adjusts clock rate α: $C_i(t) = \alpha H_i(t) + \beta$.
- Improve precision
  - by taking several measurements and taking the smallest round trip
  - or use an average after throwing out the large values

The Berkeley Algorithm

Designed for internal synchronization.



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

b) The machines answers their *offset*

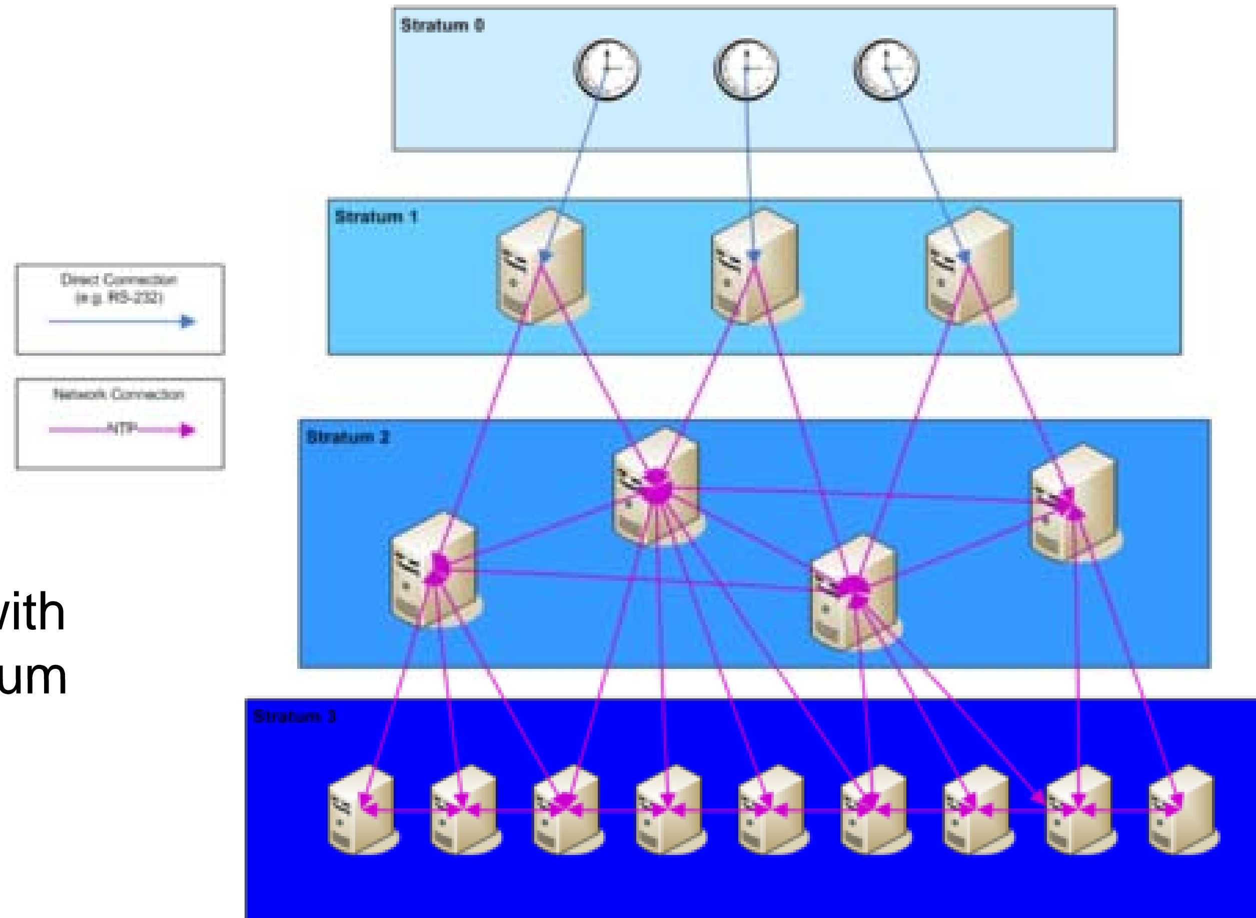
c) The time daemon tells each how to ***adjust*** its clocks

Network time protocol

- Synchronization of clients relative to UTC on an internet-wide scale
- Reliable, even in the presence of extensive loss of connectivity
- Allow frequent synchronization (relative to clock drift)
- Tolerant against disturbance
- <1ms within LAN
- 1-10 ms internet scale

read more about NTP at http://www.ntp.org also, check out RFCs 1305 & 2030.

NTP Stratum



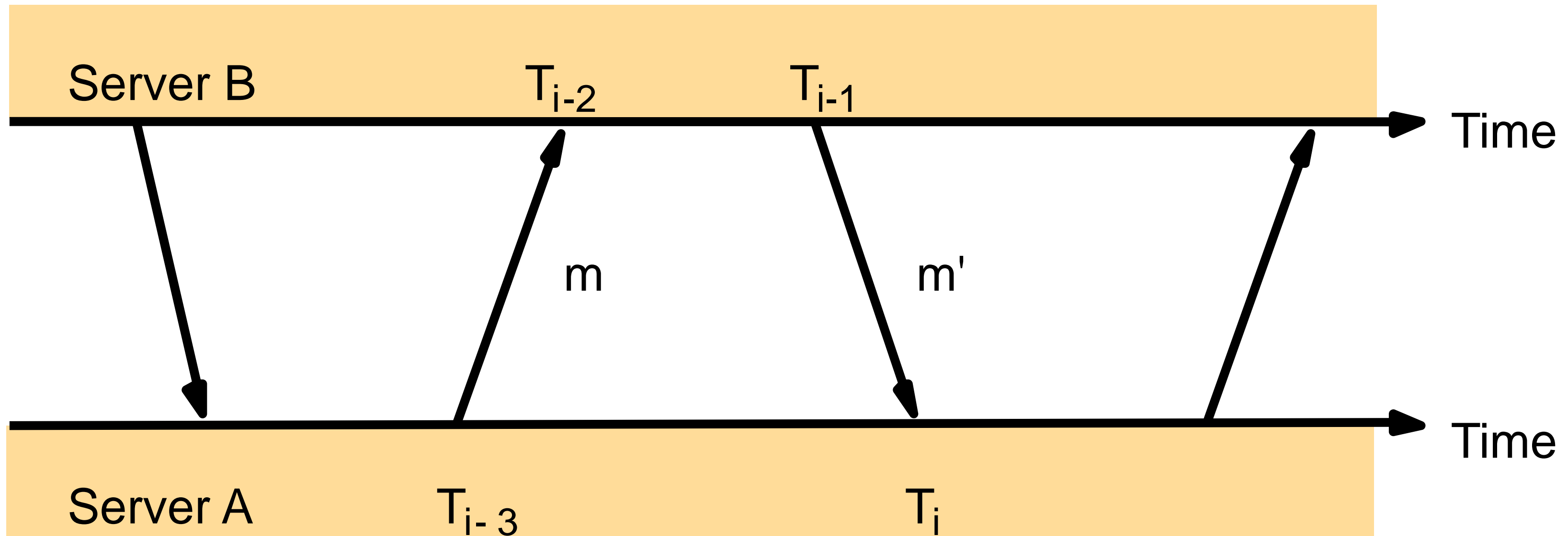
- Never synchronizewith servers at lover stratum

NTP-Modes

- Multicast (for quick LANs, low accuracy)
  - server periodically sends its actual time to its leaves in the LAN
- Procedure-call (medium accuracy)
  - server responds to requests with its actual timestamp
  - like Cristian's algorithm
- Symmetric mode (high accuracy)
  - used to synchronize between pairs of time servers with resp. high and low stratum

- In all cases, the UDP is used

Messages exchanged between a pair of NTP peers



$o_i=(T_{i-2}-T_{i-3}+T_{i-1}-T_i)/2$

$d_i=T_{i-2}-T_{i-3}+T_i-T_{i-1}$

- Exchanges local timestamps to estimate **offset o_i** and **delay d_i**
- NTP servere filters pairs <o_i,d_i>, saves 8 latest
- Use the o_i, with smallest d_i.(the smaller delay the better accuracy)

Theoretical Foundations

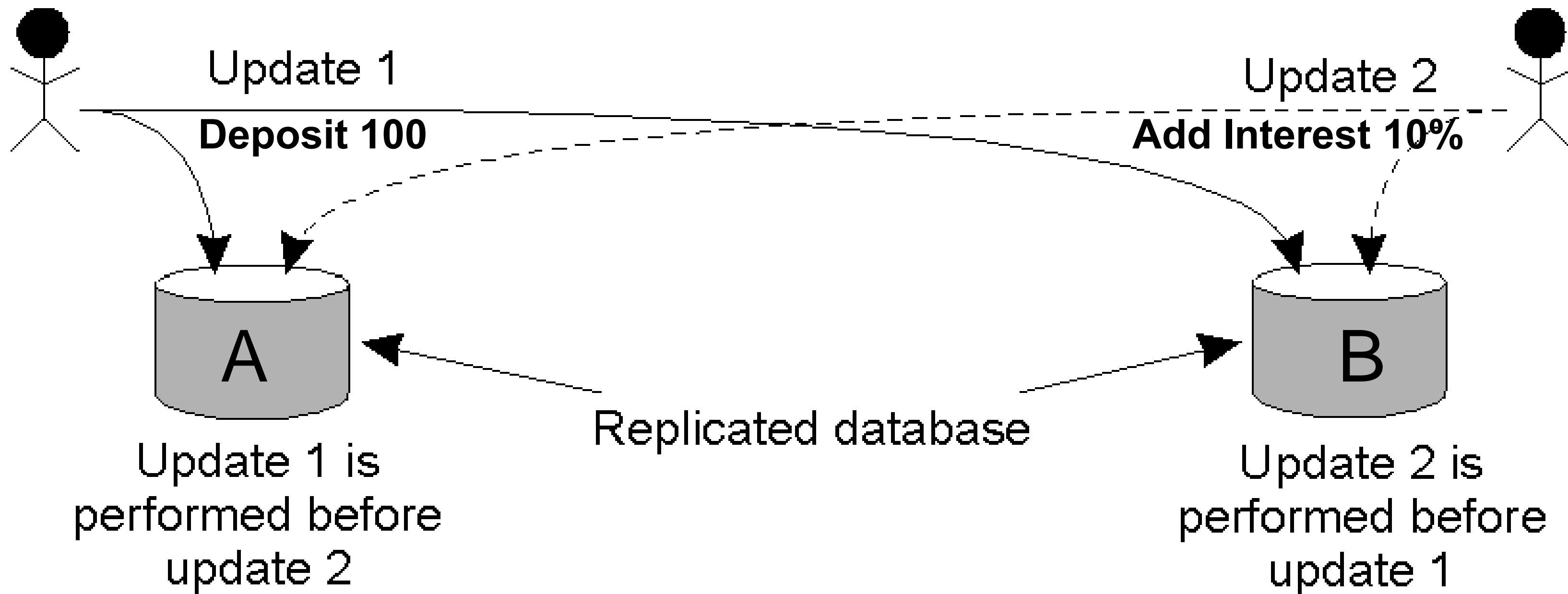
- Inherent characterstic of a distributed system:
  - Absence of a global clock:
  - Absence of 100% accurately synchronized clocks

  - Impact: Due to the absence of global clock, it is difficult to reason about the temporal order of events in distributed system, e.g. scheduling events is more difficult.

Logical Clocks in a DS

- What is important is usually not when things happened but in what order they happened so the integer counter works well in a centralized system.
- However, in a DS, each system has its own logical clock, and you can run into problems if one "clock" gets ahead of others. (like with physical clocks)
- **RELIABLE WAY OF ORDERING EVENTS IS REQUIRED**
- We need a rule to synchronize the logical clocks.

Event Ordering



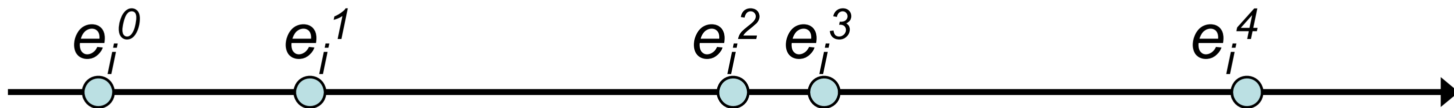
- A=100; B=100
- A'=(100+100)+10%=220
- B'=(100+10%)+100=210
- Updates need to be performed in the ***same order*** at all sites of a replicated database.

Events and Logical Clocks

- Leslie Lamport's 1978 paper: *Time, Clocks, and the Ordering of Events in Distributed Systems.*
  - Theoretical Foundation
  - Logical Clocks
  - Partial and Total Event Ordering
  - Towards distribute mutual exclusion
  - ***MUST KNOW FOR ANY COMPUTER SCIENTIST***

System Model

- A distributed system is a collection ***P*** of **sequential** processes p_i, i= 1,2,…N.
- A process p_i has state s_i
- Each process p_i executes a sequence of actions
  - Sending a message;
  - Receiving a message;
  - Performing an internal computation that alters its state s_i;
- The sequence of events within a single process p_i are totally ordered $e \rightarrow_i e'$



- The history of process p_i is the sequence of events that takes place therein
  $history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

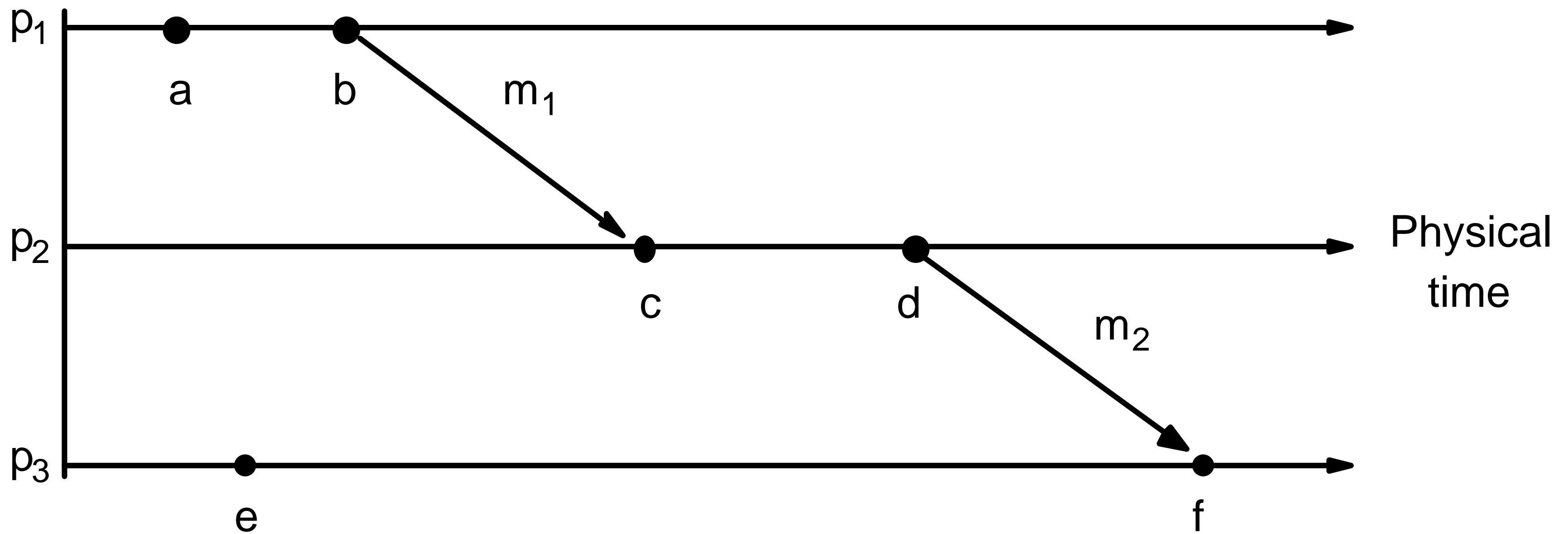
Happened Before Relation

- The **happened-before** relation captures the causal dependencies between events,
  1. a → b if a and b are events in the same process and a occurred before b.
  2. a → b if a is the event of sending a message m in a process and b is the event of receipt of the same message m by another process.
  3. If a → b and b → c, then a→c, i.e. happened before relation is transitive.
- That is, past events causal affects future events

Concurrent Events

- Two distinct events a and b are **concurrent** (a||b) if not (a→b or b→a).
- We cannot say whether one event happened-before
- For any two events a and b in a distributed system, either a→b, b→a or a||b.

Example



- a → b and a → c and b →f
- b→f and e→f Does e → b?

Logical Clocks

- There is a clock C_i at each process p_i
- The clock C_i can be thought of as a function that assigns a number $C_i(a)$ to any event a, called the **timestamp** of event a, at p_i
- These clocks can be implemented by counters and have no relation to physical time.

Conditions Satisfied by the System of Clocks

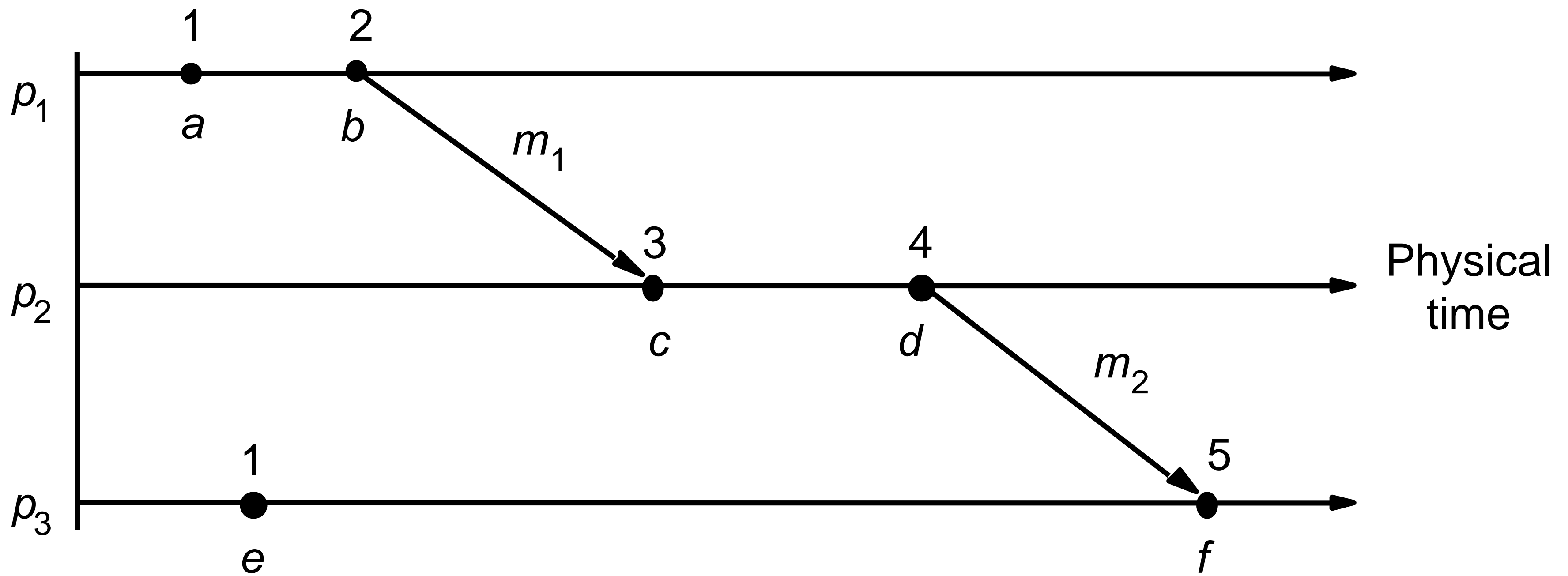
- For any events a and b: if $a \rightarrow b$, then $C(a) < C(b)$.

- implies the following two conditions:
  - **[C1]** For any two events a and b in a process P_i, if a occurs before b, then $C_i(a) < C_i(b)$.
  - **[C2]** If a is the event of sending a message m in process P_i and b is the event of receiving the same message m at process P_j, then $C_i(a) < C_j(b)$.

Implementation Rules

- **[IR1]** Before P_i timestamps an event $C_i := C_i + 1$
- **[IR2a]** P_i sends m: [IR1] and piggy-back timestamp $t = C_i$: $m' = <m, t>$
- **[IR2b]** P_j receives $m' = <m, t>$: $C_j := \max(C_j, t)$, followed by [IR1]

Lamport clocks example

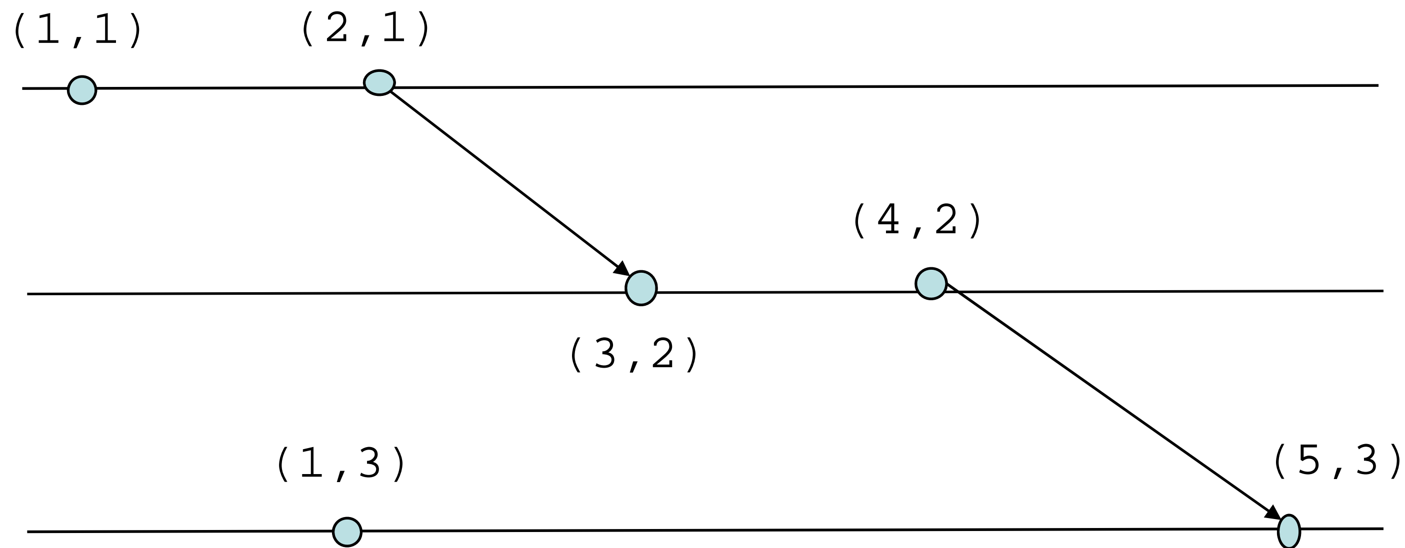


NOTE: C(e)<C(b), but not e → b

Total Ordering of Events

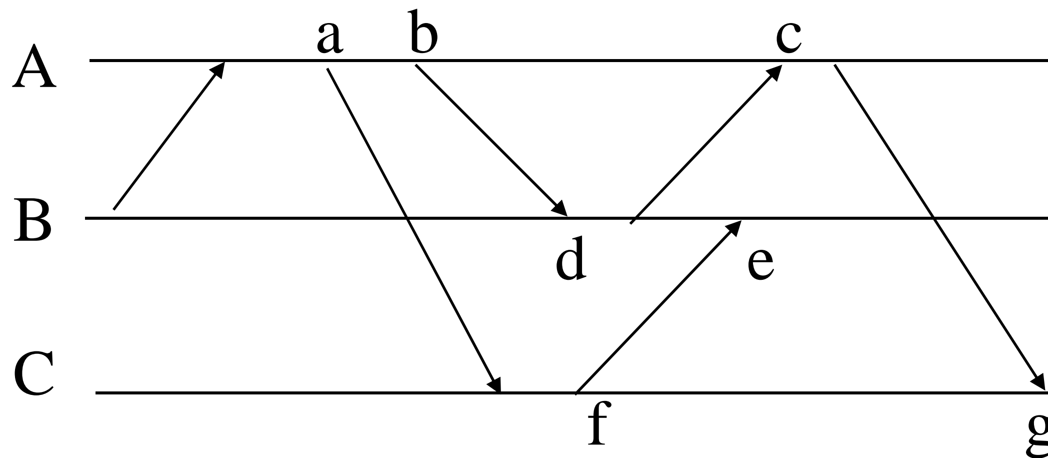
- Lamport's happened before relation defines an irreflexive partial order among the events.
- Total ordering (denoted by =>) can be obtained by using process-id to break tie if timestamps are equal:
- a => b iff
  1. $C_i(a) < C_j(b)$ or
  2. $C_i(a) = C_j(b)$ and $P_i < P_j$
- Allows processes to agree on order everywhere based on timestamp

Total Order Lamport Timestamps



- The order will be (1,1), (1,3), (2,1), (3,2) etc

Exercise: Lamport Clocks



- Assuming the only events are message send and receive, what are the clock values at events a-g?

Vector Clocks

- Lamport: e $\rightarrow$ f implies $C(e) < C(f)$
- Vector clocks: e $\rightarrow$ f **iff** $C(e) < C(f)$
- *Allows nodes to order events in happens-before order based on time-stamps*

- **Vector timestamps**: Each node maintains an array of N counters
- $\mathbf{V}_i[i]$ is the local clock for process p_i
- In general, $\mathbf{V}_i[j]$ is the latest info the node has on what p_j's local clock is.

Implementation Rules

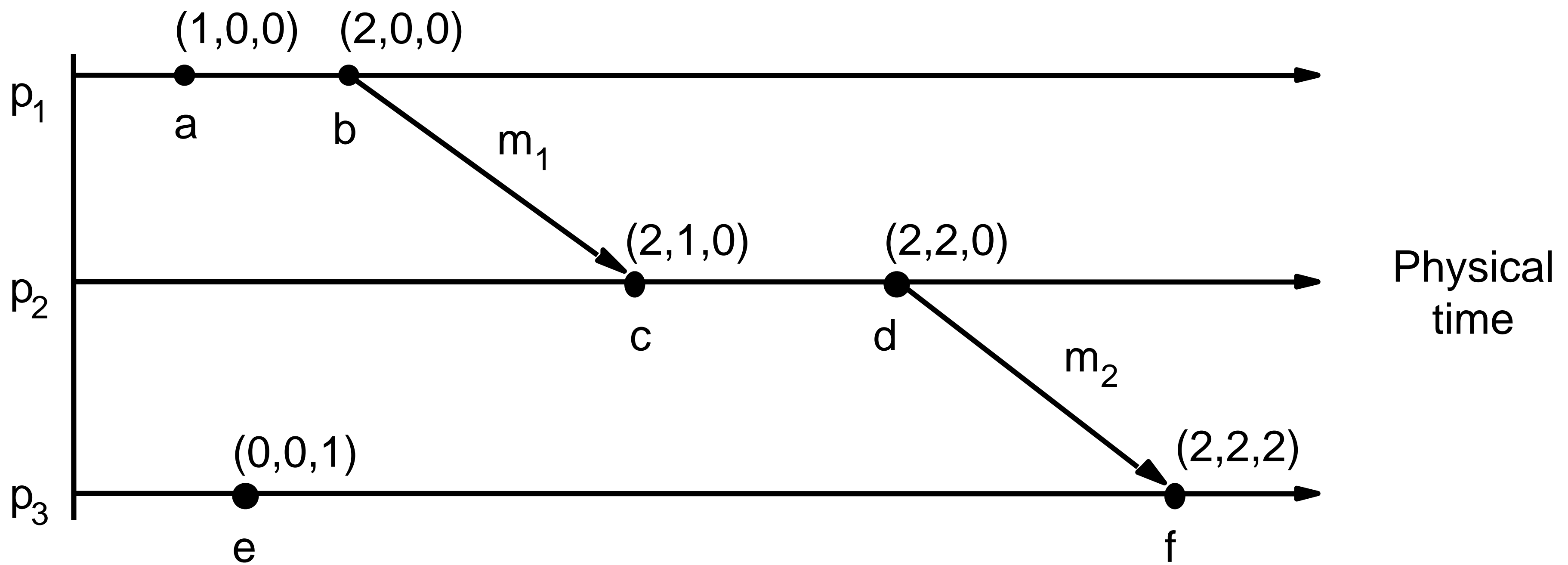
- **[VC1]** Initially $\mathbf{V}_i[j]=0$ for $i,j = 1\ldots N$
- **[VC2]** Before P_i timestamps an event: $\mathbf{V}_i[i] := \mathbf{V}_i\,[i] + 1$
- **[VC3]** P_i sends m: piggy-back vector timestamp $\mathbf{t}=\mathbf{V}_i$: $m' = <m, \mathbf{t}>$
- **[VC4]** P_j receives $m' = <m, \mathbf{t}>$ $\mathbf{V}_i[j] := \max(\mathbf{V}_i[j]\ ,\ \mathbf{t}_i[j]),\ l != j$

Comparison of Vector Clocks

Comparing vector clocks

- $V = V'$ iff $V[j] = V'[j]$ for all $j=1,2,\ldots,N$.
- $V \leq V'$ iff $V[j] \leq V'[j]$ for all $j=1,2,\ldots,N$.
- $V < V'$ iff $V \leq V'$ and $V \neq V'$.

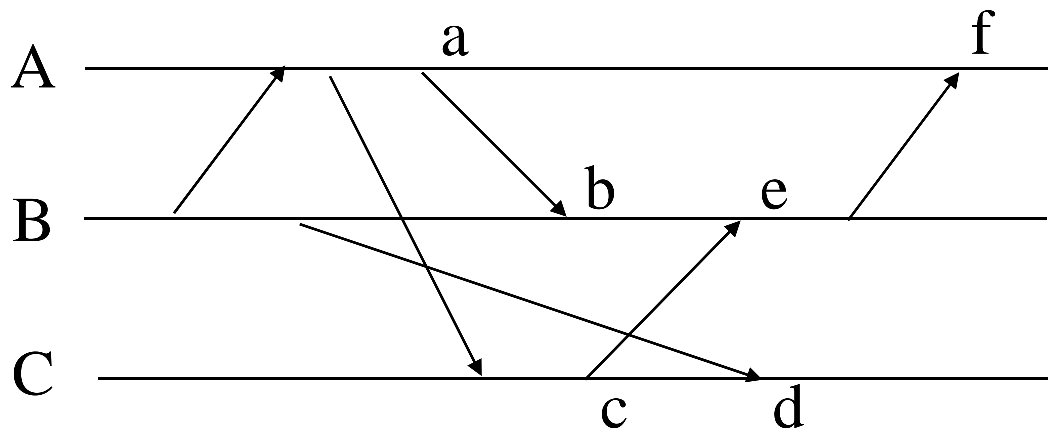
Vector clocks illustrated



NOTE e and b are not related

Vector Clock Exercise

- Assuming the only events are send and receive:
- What is the vector clock at events a-f?
- Which events are concurrent?



END