CS542: Topics in Distributed Systems

Diganta Goswami

Concept of Time in Distributed Systems

Time in DS

- **Time is an interesting and Important issue in DS**
  - **Ex. At what time of day a particular event occurred at a particular computer … Consistency (use of timestamp for serialization), e-commerce, authentication etc.**
- **Algorithms that depend upon clock synchronization have been developed for several problems.**
- **Due to loose synchrony, the notion of physical time is problematic in DS**
  - **There is no absolute physical "global time" in DS**

Clocks

- How time is really measured?
  - » Earlier: Solar day, solar second, mean solar second

- **Solar day: time between two consequtive transits of the sun**
- **Solar second: 1/86400 of a solar day**
- **Mean solar day: average length of a solar day**

- **Problem: solar day gets longer because of slowdown of earth rotation due to friction (300 million years ago there were 400 days per year)**

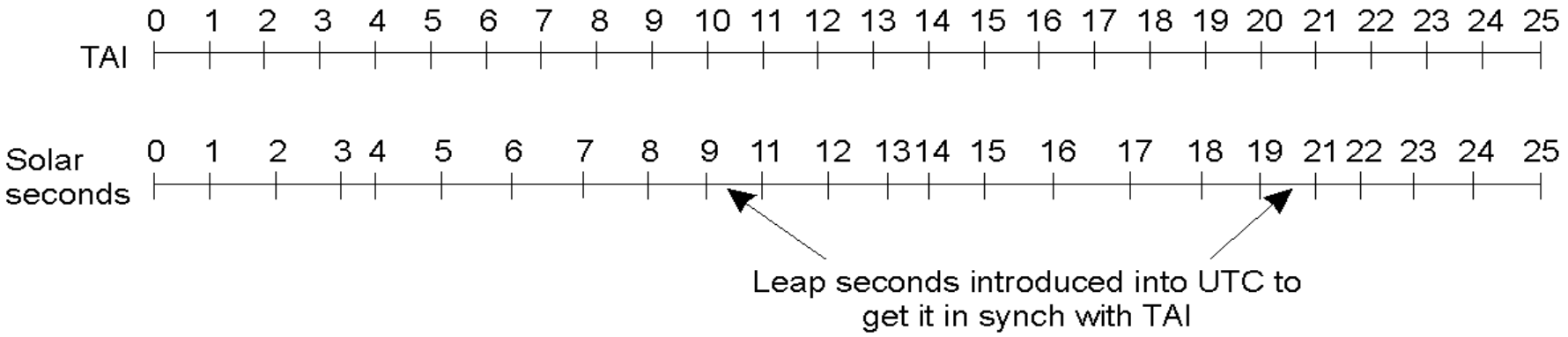
Physical Clocks

- **International Atomic Time (TAI): number of ticks of Cesium 133 atom since 1/1/58 (atomic second)**
- **Atom clock: one second defined as (since 1967) 9,192,631,770 transitions of the atom Cesium 133**

Physical Clocks

- **Because of slowdown of earth, leap seconds have to be introduced**
- **Correction of TAI is called Universal Coordinated Time (UTC): 30 leap seconds introduced so far**
- **Network Time Protocol (NTP) can synchronize globally with an accuracy of up to 50 msec**

Physical Clocks



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

Physical Clocks

- Let $C(t)$ be a perfect clock
- A clock $C_i(t)$ is called correct at time t if $C_i(t) = C(t)$
- A clock $C_i(t)$ is called accurate at time t if $dC_i(t)/dt = dC(t)/dt = 1$
- Two clocks $C_i(t)$ and $C_k(t)$ are synchronized at time t if $C_i(t) = C_k(t)$

Clocks

- **Computers contain physical clock (crystal oscillator)**
  - » **Physical time *t*, hardware time $H_i(t)$, software time $C_i(t)$**
  - » ***The clock output can be read by SW and scaled into a suitable time unit and the value can be used to timestamp any event***

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

- **Clock skew**
  - » **The instantaneous difference between the readings of any two clocks**

- **Clock drift: Crystal-based clocks count time at different rates, and so diverge.**

Clocks

- **Underlying oscillators are subject to physical variations, with the consequence that their frequencies of oscillation differ**
  - **Even the same clock's freq varies with temp.**
  - **Designs exists that attempt to compensate for this variation but they cannot eliminate it.**
  - **The diff in the oscillations between two clocks might be small, but the difference accumulated over many oscillations leads to an observable difference**
- **For clocks based on a quartz crystal, the drift is about 10^{-6} sec/sec – giving a difference of one second every 1,000,000 sec or 11.6 days.**

Why synchronization?

- **You want to catch the 5 pm bus at the Subansiri stop, but your watch is off by 15 minutes**
  - **What if your watch is Late by 15 minutes?**
  - **What if your watch is Fast by 15 minutes?**

- **Synchronization is required for**
  - **Correctness**
  - **Fairness**

Why synchronization?

- Airline reservation system
- Server A receives a client request to purchase last ticket on flight ABC 123.
- Server A timestamps purchase using local clock 9h:15m:32.45s, and logs it. Replies ok to client.
- That was the last seat. Server A sends message to Server B saying “flight full.”
- B enters “Flight ABC 123 full” + local clock value (which reads 9h:10m:10.11s) into its log.
- Server C queries A’s and B’s logs. Is confused that a client purchased a ticket after the flight became full.
  - May execute incorrect or unfair actions.

Basics – Processes and Events

- **An Asynchronous Distributed System (DS) consists of a number of *processes.***
- **Each process has a state (values of variables).**
- **Each process takes actions to change its state, which may be an instruction or a communication action (send, receive).**
- **An event is the occurrence of an action.**
- **Each process has a local clock – events *within* a process can be assigned timestamps, and thus ordered linearly.**
- **But – in a DS, we also need to know the time order of events *across* different processes.**

☹ **Clocks across processes are not synchronized in an asynchronous DS**

(unlike in a multiprocessor/parallel system, where they are). So…

1. **Process clocks can be different**
2. **Need algorithms for either (a) time synchronization, or (b) for telling which event happened before which**

Physical Clocks & Synchronization

- In a DS, each process has its own clock.
- Clock Skew versus Drift
  - Clock Skew = Relative Difference in clock *values* of two processes
  - Clock Drift = Relative Difference in clock *frequencies (rates)* of two processes
- *A non-zero clock drift causes skew to increase (eventually).*
- Maximum Drift Rate (MDR) of a clock
- Absolute MDR is defined relative to Coordinated Universal Time (UTC). UTC is the "correct" time at any point of time.
  - MDR of a process depends on the environment.
- Max drift rate between two clocks with similar MDR is 2 * MDR

  Max-Synch-Interval =

  (MaxAcceptableSkew—CurrentSkew) / (MDR * 2)

  (i.e., time = distance/speed)

Clock synchronization

- If the UTC time is *t* and the process *i's* time is $C_i(t)$ then ideally we would like to have $C_i(t) = t$, *or* $dC/dt = 1$.

- In practice, we use a tolerance variable ρ, such that

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

- In external synchronization, clock is synchronized with an authoritative external source of time
- In internal synchronization clocks are synchronized with one another with a known degree of accuracy

Synchronizing Physical Clocks

- **$C_i(t)$: the reading of the software clock at process *i* when the real time is *t*.**
- **External synchronization: For a synchronization bound *D>0*, and for source S of UTC time,**

$$\left| S(t) - C_i(t) \right| < D,$$

for *i=1,2,...,N* and for all real times *t*.

Clocks C_i are externally accurate to within the bound *D*.

In external synchronization, clock is synchronized with an authoritative external source of time

Synchronizing Physical Clocks

- **Internal synchronization: For a synchronization bound *D>0*,**

$$\left|C_i(t) - C_j(t)\right| < D$$

for *i, j=1,2,...,N* and for all real times *t.*

Clocks C_i are internally accurate within the bound *D*.

In internal synchronization clocks are synchronized with one another with a known degree of accuracy

- **External synchronization with *D* ⇒ Internal synchronization with *2D***
- **Internal synchronization with D ⇒ External synchronization with ??**

Clock synchronization

- **UTC signals are synchronized and broadcast regularly from land-based radio stations and satellites covering many parts of the world**
  - **E.g. in the US the radio station WWV broadcasts time signals on several short-wave frequencies**
  - **Satellite sources include Geo-stationary Operational Environmental Satellites (GOES) and the GPS**

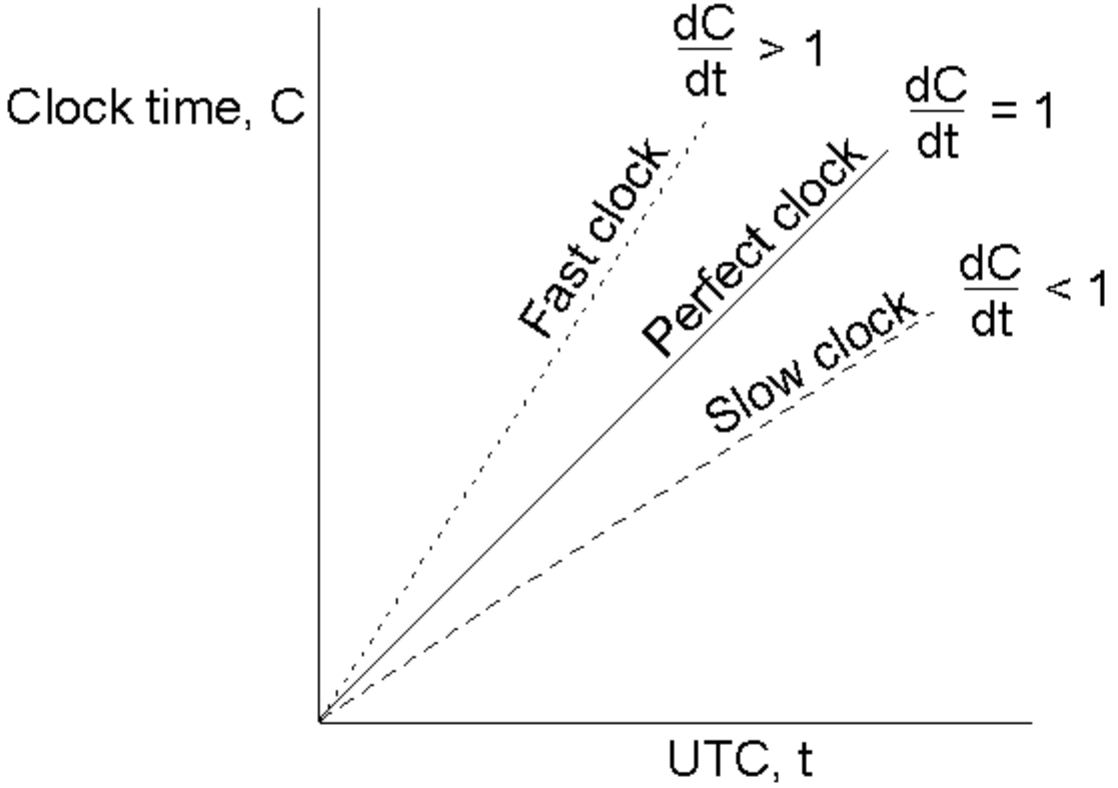
Clock synchronization

- Radio waves travel at near the speed of light. The propagation delay can be accounted for if the exact speed and the distance from the source are known
- Unfortunately, the propagation speed varies with atmospheric conditions – leading to inaccuracy
- Accuracy of a received signal is a function of both the accuracy of the source and its distance from the source through the atmosphere

Clock Synchronization Algorithms

- **The relation between clock time and UTC when clocks tick at different rates.**

Problem: show that, in order to guarantee that no two clocks differ by more than δ, clocks must be resynchronized at least every δ/2ρ seconds.



Clock Synchronization Algorithms

- **The constant ρ is specified by the manufacturer and is known as the maximum drift rate.**
- **If two clocks are drifting from the Universal Coordinated Time (UTC) in opposite direction, at a time Δt after they are synchronized, they maybe as much as $2^* \rho * \Delta t$ apart.**
- **If the operating system designer want to guarantee that no two clocks ever differ by more than δ, clocks must be synchronized at least every $\delta / 2 \rho$ seconds.**

Clock synchronization

- **Remember the definition of synchronous distributed system?**
  - » **Known bounds for message delay, clock drift rate and execution time.**
    - **Clock synchronization is easy in this case**
  - » **In practice most DS are asynchronous.**
    - **Cristian's Algorithm**
    - **The Berkeley Algorithm**

Clock synchronization in a synchronous system

- **Consider internal synch between two procs in a synch DS**
- **P sends time *t* on its local clock to Q in a msg *m***
- **In principle, Q could set its clock to the time $t + T_{trans}$, where T_{trans} is the time taken to transmit m between them**
- **The two processes would then agree (internal synch)**

Clock synchronization in a synchronous system

- **Unfortunately, T_{trans} is subject to variation and is unknown**
  - **All processes are competing for resources with P and Q and other messages are competing with *m* for the network**
  - **But there is always a minimum transmission time *min* that would be obtained if no other processes executed and no other network traffic existed**
  - ***min* can be measured or conservatively estimated**

Clock synchronization in a synchronous system

- **In synch system, by definition, there is also an upper bound *max* on the time taken to transmit any message**
- **Let the uncertainty in the msg transmission time be *u*, so that *u* = (*max* – *min*)**
  - **If Q sets its clock to be (*t* + *min*), then clock skew may be as much as *u* (since the message may in fact have taken time *max* to arrive).**
  - **If Q sets it to (*t* + *max*), the skew may again be as large as *u*.**
  - **If, however, Q sets it clock to (*t* + *(max* + *min)/2*), then the skew is at most *u/2*.**
  - **In general, for a synch system, the optimum bound that can be achieved on clock skew when synchronizing *N* clocks is *u(1-1/N)***
- **For an asynch system T_{trans} = *min* + *x*, where *x* >=0**

Cristian's clock synchronization algorithm

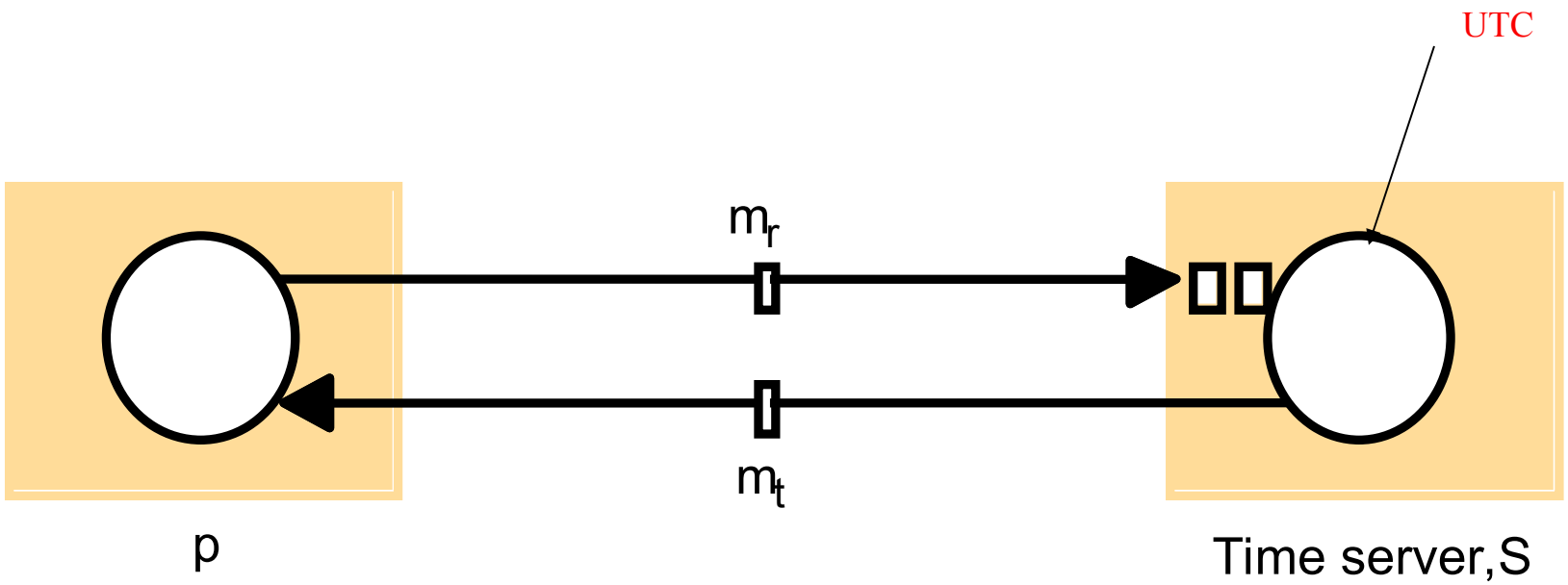
- **Asynchronous system**
  - **Achieves synchronization only if the observed RTT between the client and server is sufficiently short compared with the required accuracy.**

Cristian's clock synchronization algorithm

- **Observations:**
  - **RTT between processes are reasonably short in practice, yet theoretically unbounded**
  - **Practical estimate possible if RTT is sufficiently short in comparison to required accuracy**
  - **In LAN RTT should be around 1-10ms during which a clock with a drift rate of 10^{-6} s/s varies by at most 10^{-5} ms. Hence the estimate of RTT is reasonably accurate**

Cristian's clock synchronization algorithm

- Periodically (< $\delta/2\rho$) each machine sends a request message m_r and the server sends the time in m_t.
- The roundtrip message transmission delay time is computed, say this is T_{round}.
- The algorithm sets $t + T_{round}/2$ as the new time. (naïve i.e. the elapsed time is split equally before & after S placed *time* in m_t.)



Cristian's Algorithm (Accuracy)

- **Assumption**
  - **Request & reply via same network**
  - **The value of minimum transmission time *min* is known or conservatively estimated.**

Cristian's Algorithm (Accuracy)

- **The earliest point at which S could have placed the time in m_t was *min* after p dispatched m_r.**
- **The latest point at which it could have done this was *min* before m_t arrived at p.**
- **The time by S's clock when the reply message arrives is therefore in the range [t + *min*, t + T_{round} – *min*]**
- **The width of this range is T_{round} – 2*min*, so the accuracy is +/- (T_{round} / 2 – *min*)**

Cristian's Algorithm (Discussion)

- **Variability can be dealt with making several requests & taking the min value of RTT to give the accurate estimate**

- **Cristian suggested making a series of measurements, and throw away measurements which exceed some threshold value. He assumes that these discarded entries are victims of network congestion. These measurements are then averaged to give a better number for network delay and got added to the current time.**

- **Cristian stated that the message that came back the fastest can be taken to be the most accurate since it presumably encountered the least traffic.**

Cristian's Algorithm (Discussion)

- **Only suitable for deterministic LAN environment of Intranet**
- **Problem of failure of single time server S**
  - **Redundancy through a group of servers (use multicasting and use only the first reply obtained)**
  - **How to decide if replies vary? (byzantine agreement)**
- **Imposter providing false clock readings**
  - **Authentication**

The Berkeley Algorithm (internal synchronization)

- **A coordinator (time server): *master***
- **Just the opposite approach of Cristian's algorithm**
  - **Periodically the *master* polls the time of each client (slave) whose clocks are to be synchronized.**
  - **Based on the answer (by observing the RTT as in Cristian's algorithm), it computes the average (including its own clock value) and broadcasts the new time.**
- **This method is suitable for a system in which no machine has a WWV receiver getting the UTC.**
- **The time daemon's time must be set manually by the operator periodically.**

The Berkeley Algorithm

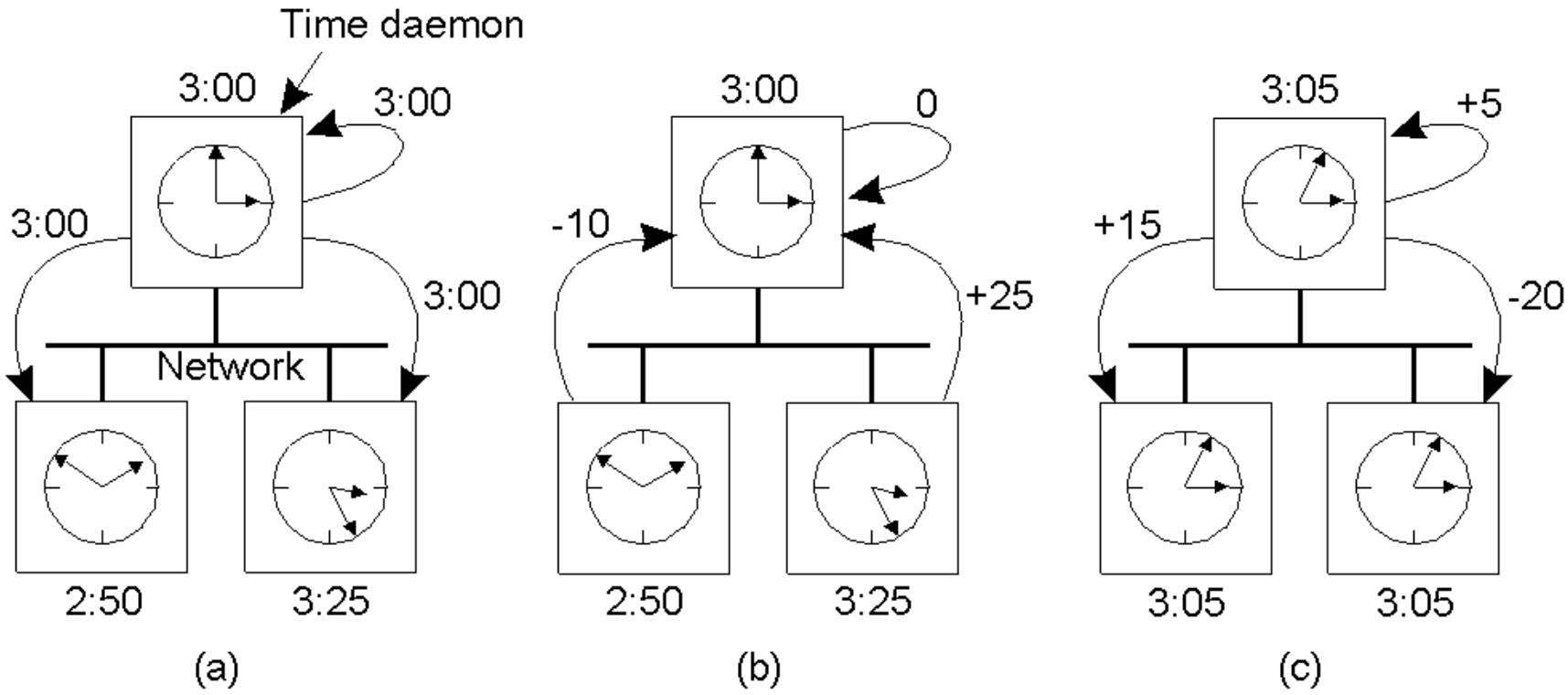
- **The balance of probabilities is that the average cancels out the individual clock's tendencies to run fast or slow**
- **The accuracy depends upon a nominal maximum RTT between the master and the slaves**
- **The master eliminates any occasional readings associated with larger times than this maximum**

The Berkeley Algorithm

- Instead of sending the updated current time back to the comps – which will introduce further uncertainty due to message transmission time – the master send the amount by which each individual slave's clock requires adjustment (+ or -)

- The algorithm eliminates readings from faulty clocks (since these could have significant adverse effects if an ordinary average was taken) – a subset of clock is chosen that do not differ by more than a specified amount and then the average is taken.

The Berkeley Algorithm



- **The time daemon asks all the other machines for their clock values**
- **The machines answer**
- **The time daemon tells everyone how to adjust their clock**

Averaging Algorithms

- Both Cristian's and Berkeley's methods are highly centralized, with the usual disadvantages - single point of failure, congestion around the server, … etc.
- One class of decentralized clock synchronization algorithms works by dividing time into fixed-length re-synchronization intervals.
- The i^{th} interval starts at $T_0 + iR$ and runs until $T_0 + (i+1)R$, where T_0 is an agreed upon moment in the past, and R is a system parameter.

Averaging Algorithms

- At the beginning of each interval, every machine broadcasts the current time according to its clock.
- After a machine broadcasts its time, it starts a local timer to collect all other broadcasts that arrive during some interval S.
- When all broadcasts arrive, an *algorithm* is run to compute a new time.

Averaging Algorithms

- **Some algorithms:**
  - **average out the time.**
  - **discard the *m* highest and *m* lowest and average the rest -- this is to prevent up to m faulty clocks sending out nonsense**
  - **correct each message by adding to it an estimate of propagation time from the source. This estimate can be made from the known topology of the network, or by timing how long it takes for probe message to be echoed.**

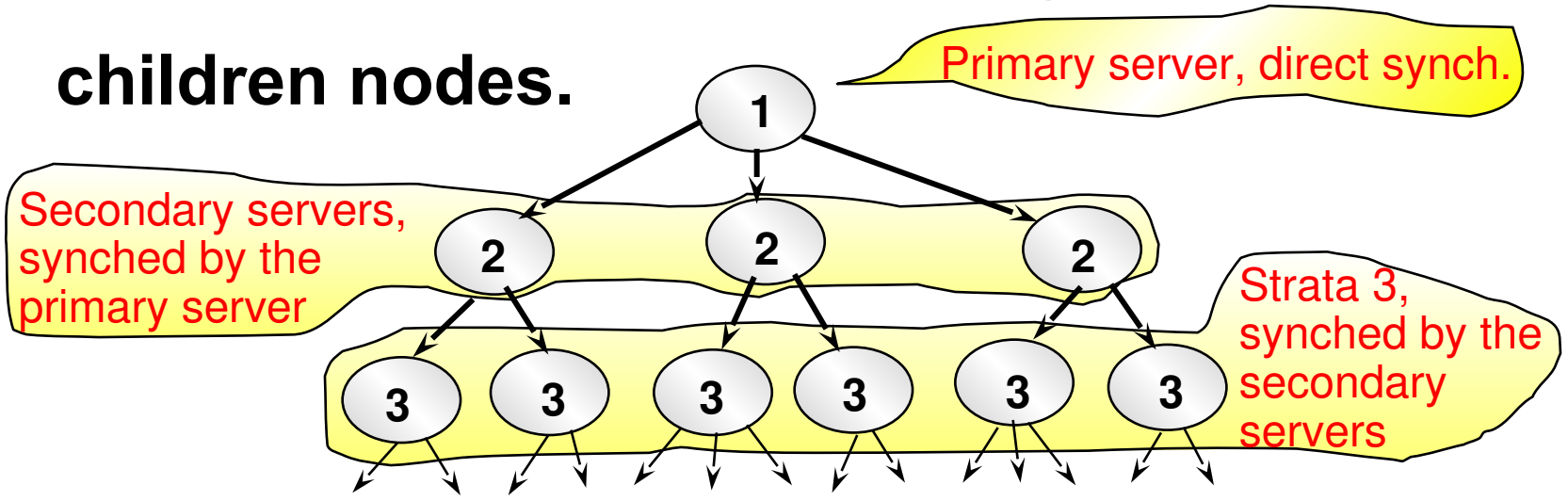
The Network Time Protocol (NTP)

- **Cristian's and Berkeley algorithms → synch within intranet**

- **NTP – defines an architecture for a time service and a protocol to distribute time information over the Internet**
  - **Provides a service enabling clients across the Internet to be synchronized accurately to UTC**
  - **Provides a reliable service that can survive lengthy losses of connectivity**
  - **Enables client to resynchronize sufficiently frequently to offset clock drifts**
  - **Provides protection against interference with the time service**

The Network Time Protocol (NTP)

- **Uses a network of time servers to synchronize all processes on a network.**
- **Time servers are connected by a synchronization subnet tree. The root is in touch with UTC. Each node synchronizes its children nodes.**



Messages exchanged between a pair of NTP peers

- **Three modes of synchronization**
- **Multicast**
  - **Intended for use in high speed LAN**
  - **One or more servers periodically multicasts the time to servers running in the other computers connected to LAN**
  - **Servers set their clocks assuming a small delay**
  - **Can achieve relatively low accuracies – considered sufficient for many purposes**

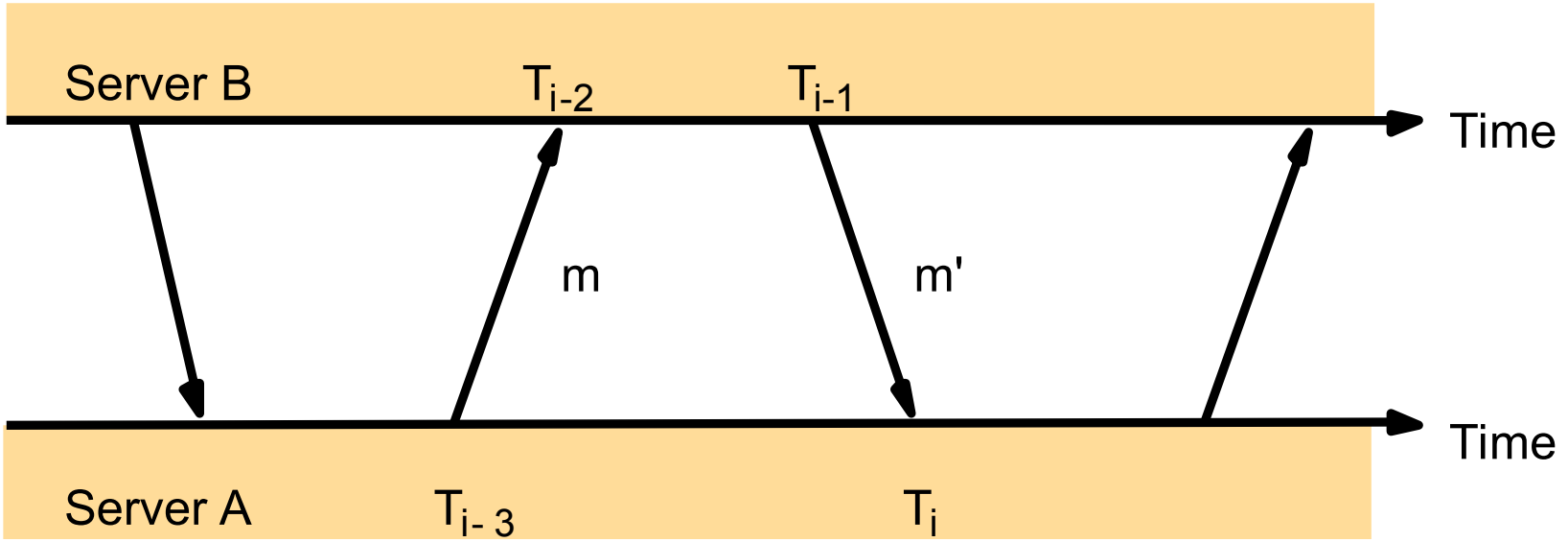
Messages exchanged between a pair of NTP peers

- **Procedure-call mode**
  - **Similar to the operation of the Cristian Algorithm**
  - **One servers accepts requests from other servers, which it processes by replying with its timestamp**
  - **Suitable where higher accuracies are required than can be achieved with multicast or where multicast is not supported in hardware**
  - **E.g. file servers on the same or other LAN could contact a local server in procedure-call mode**

Messages exchanged between a pair of NTP peers

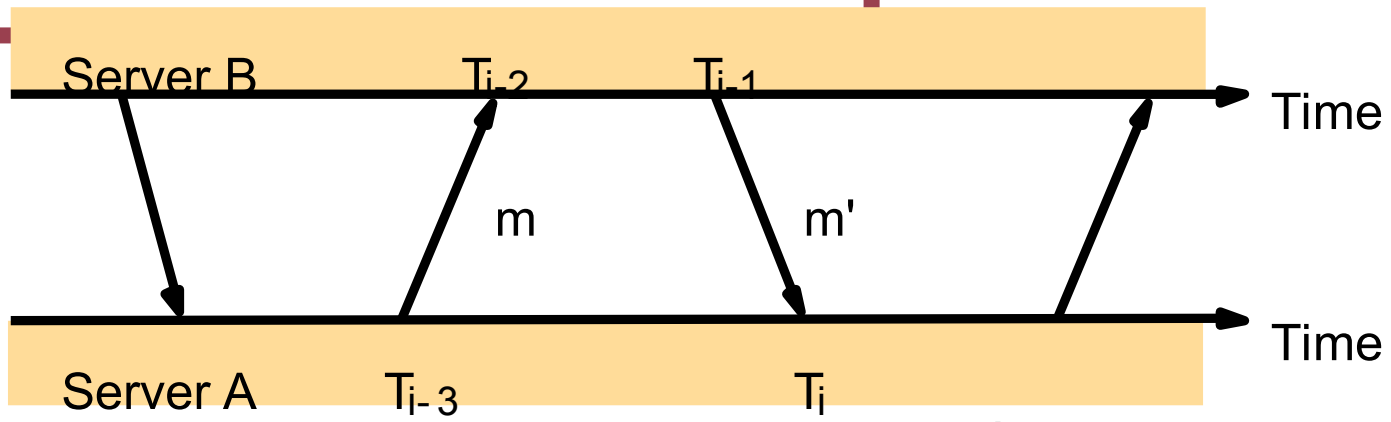
- **Symmetric mode**
  - **Intended for use by the servers that supply time information in LANs and by the higher levels (lower strata) of the synchronization subnet, where the highest accuracies to be achieved**
  - **A pair of servers operating in symmetric mode exchange messages bearing timing information**

Messages Exchanged Between a Pair of NTP Peers ("Connected Servers")



Each message bears timestamps of recent message events: the local time when the previous NTP message was sent and received, and the local time when the current message was transmitted.

Theoretical Base for NTP



- *t* and *t':* actual transmission times for *m* and *m'(unknown)*
- *o*: true offset of clock at *B* relative to clock at *A*
- o_i: estimate of actual offset between the two clocks
- d_i: estimate of accuracy of o_i ; total transmission times for *m* and *m'*; $d_i = t + t'$

$$T_{i-2} = T_{i-3} + t + o$$

$$T_i = T_{i-1} + t' - o$$

This leads to

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

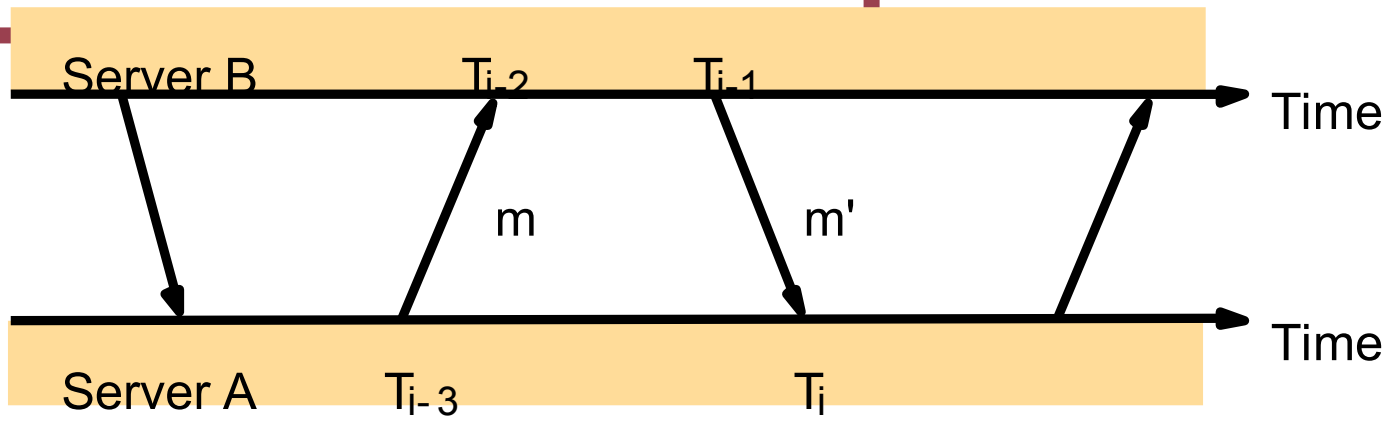
$$o = o_i + (t' - t)/2, \quad \text{where}$$

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

It can then be shown that

$$o_i - d_i/2 \le o \le o_i + d_i/2.$$

Theoretical Base for NTP



- NTP servers apply a data filtering algorithm to successive pairs $< o_i , d_i >$ which estimates the offset *o* and calculates the quality of this estimate as a statistical quantity called the *filter dispersion*.

- *The eight most recent pairs* $< o_i , d_i >$ are retained

- *The value of* o_i that corresponds to the min value of d_i is chosen to estimate *o.*

Compensating for clock drift

- **Compare time T_s provided by the time server to time T_c at computer C**
  - **If $T_s > T_c$ (e.g. 9:07 am vs 9:05 am), could advance C's time to T_s**
  - **May miss some clock ticks, probably OK**
  - **If $T_s < T_c$ (e.g. 9:07 am vs 9:10 am), cannot rollback C's time to T_s**
  - **Many applications assume that time always advances**

Compensating for clock drift

- **The solution is not to set C's clock back – but can cause C's clock to run slowly until it resynchronizes with the time server**
- **This can be achieved in SW, w/o changing the rate at which the HW clock ticks (an operation which is not always supported by HW clocks)**
- **Calculation …**

Is it enough to synchronize physical clocks?

- **Value received from UTC receiver is only accurate to within 0.1–10 milliseconds**
- **At best, we can synchronize clocks to within 10–30 milliseconds of each other**
- **We have to synchronize frequently, to avoid local clock drift**

Is it enough to synchronize physical clocks?

- **In 10 ms, a 100 MIPS machine can execute 1 million instructions**
  - **Accurate enough as time-of-day**
  - **Not sufficiently accurate to determine the relative order of events on different computers in a distributed system**

Logical Clocks

- Lamport (1978) showed that clock synchronization is possible and presented an algorithm for it. He also pointed out that clock synchronization need not be absolute.

- If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems.

Logical Clocks (contd.)

- What usually matters is not that all processes agree on exactly what time it is, but rather, that they agree on the order in which events occur.
- Therefore, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time.
- It is conventional to speak of the clocks as logical clocks.

Logical Clocks (contd.)

- On the other hand, when clocks are required to be the same, and also must not deviate from the real time by more than a certain amount, these clocks are called physical clocks.

Logical Clocks

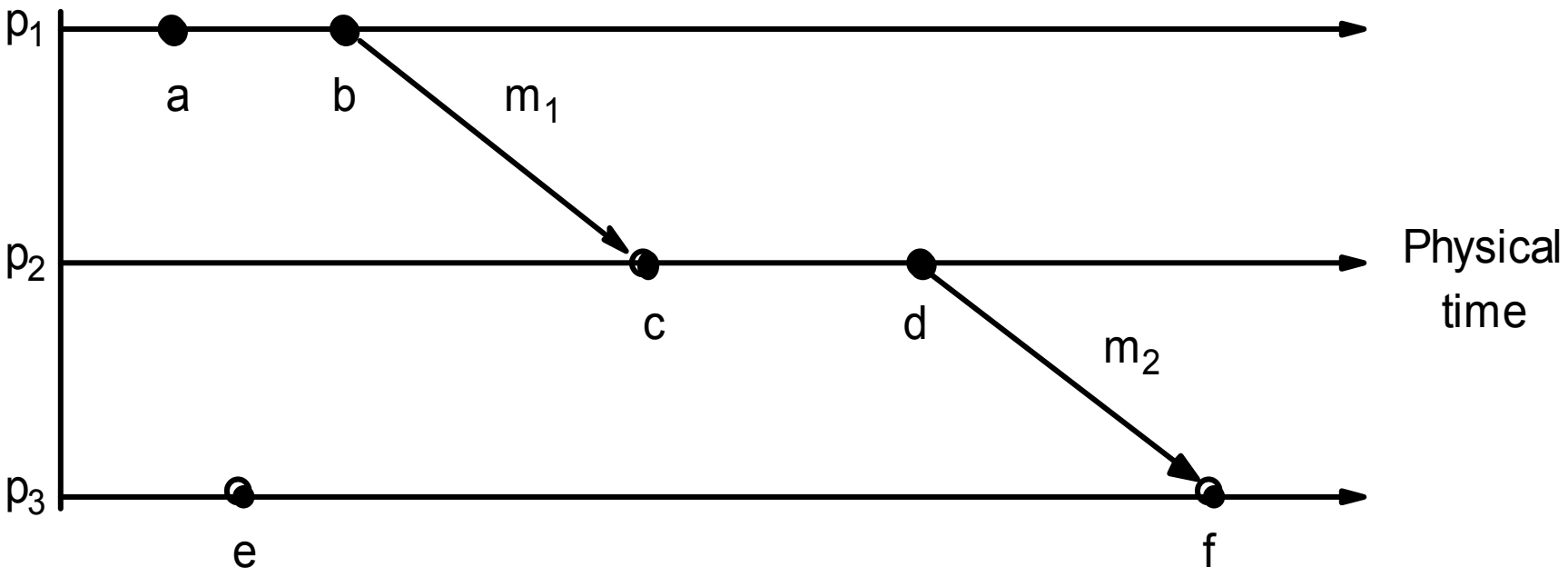
- **Is it always necessary to give *absolute* time to events?**
- **Suppose we can assign *relative* time to events, in a way that does not violate their causality**
  - **Well, that would work – we humans run our lives without looking at our watches for everything we do**
- **First proposed by Leslie *Lamport* in the 70's**
- **Define a logical relation *Happens-Before* (→) among events:**
  1. **On the same process: $a \rightarrow b$, if $time(a) < time(b)$**
  2. **If p1 sends *m* to p2: $send(m) \rightarrow receive(m)$**
  3. **(Transitivity) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$**

Logical Clocks

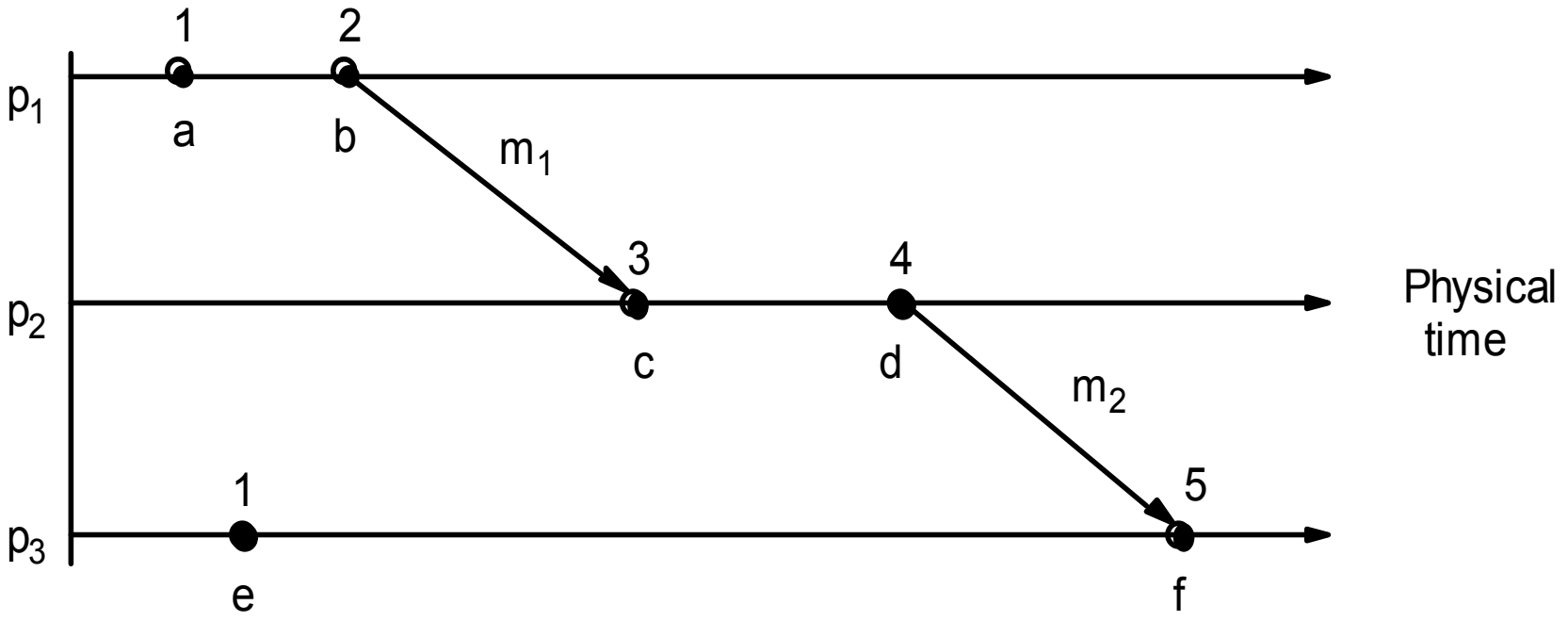
- **Lamport Algorithm assigns logical timestamps to events:**
  - All processes use a counter (clock) with initial value of zero
  - A process increments its counter when a send or an instruction happens at it. The counter is assigned to the event as its timestamp.
  - A send (message) event carries its timestamp
  - For a receive (message) event the counter is updated by

max(local clock, message timestamp) + 1

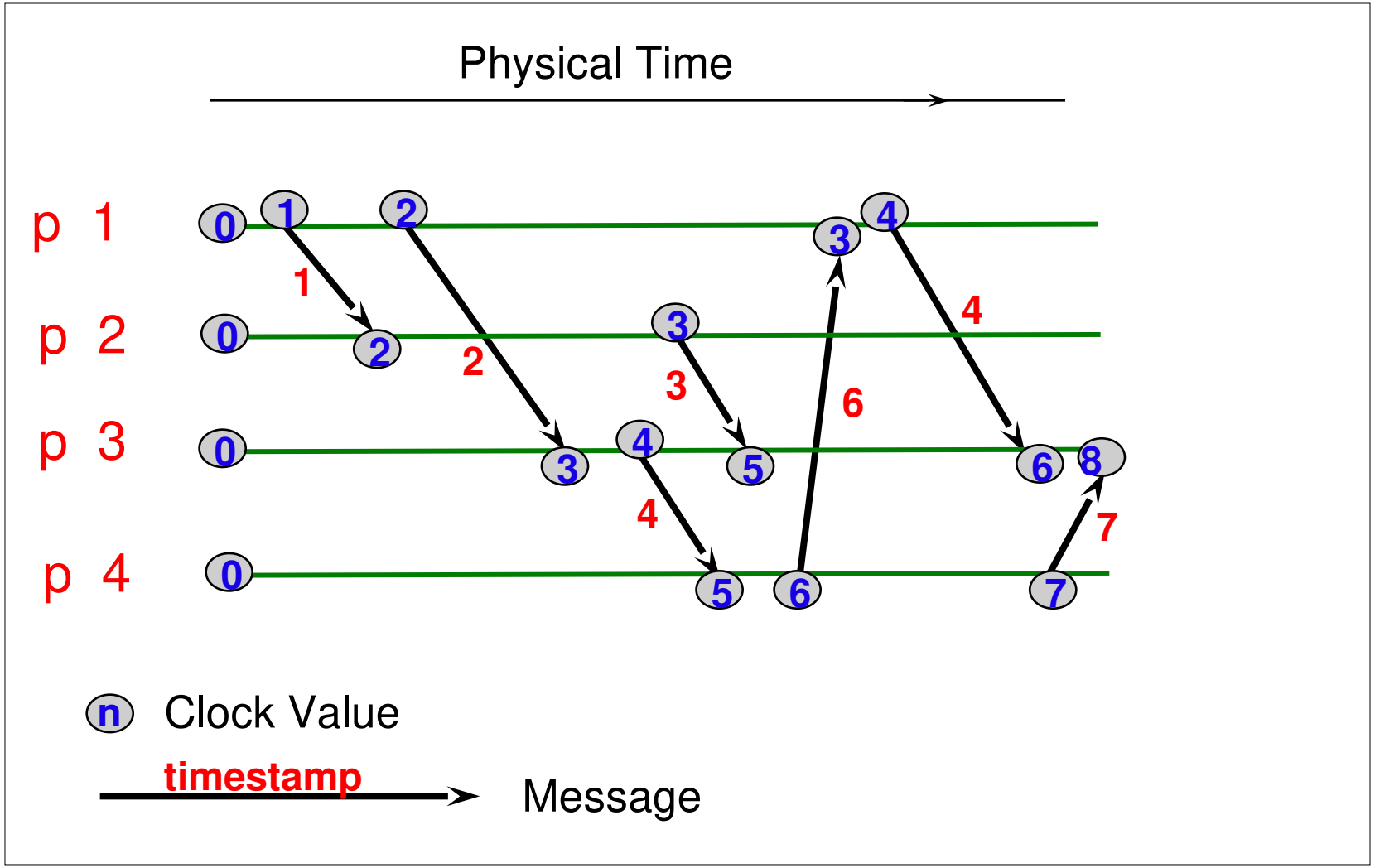
Events Occurring at Three Processes



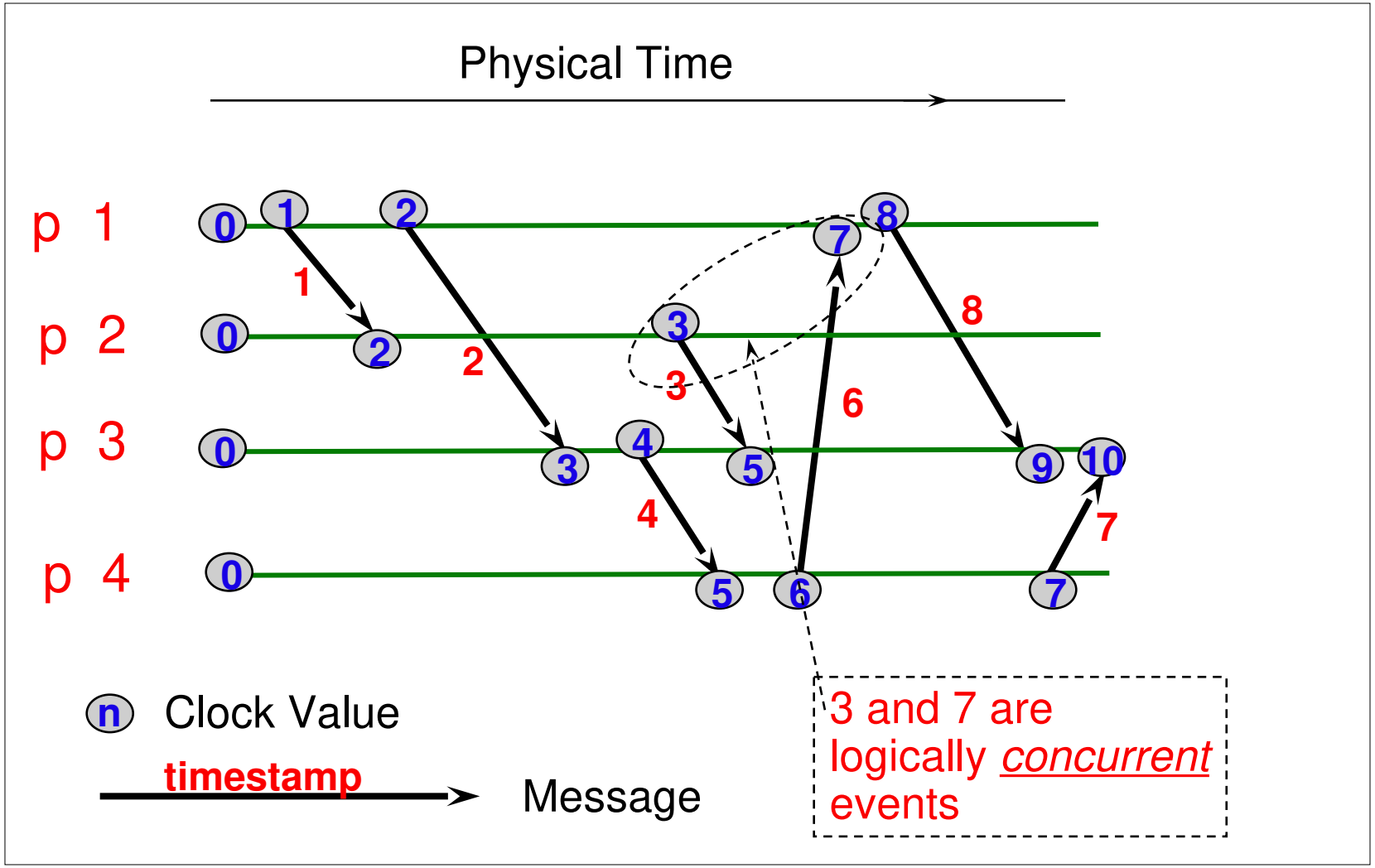
Lamport Timestamps



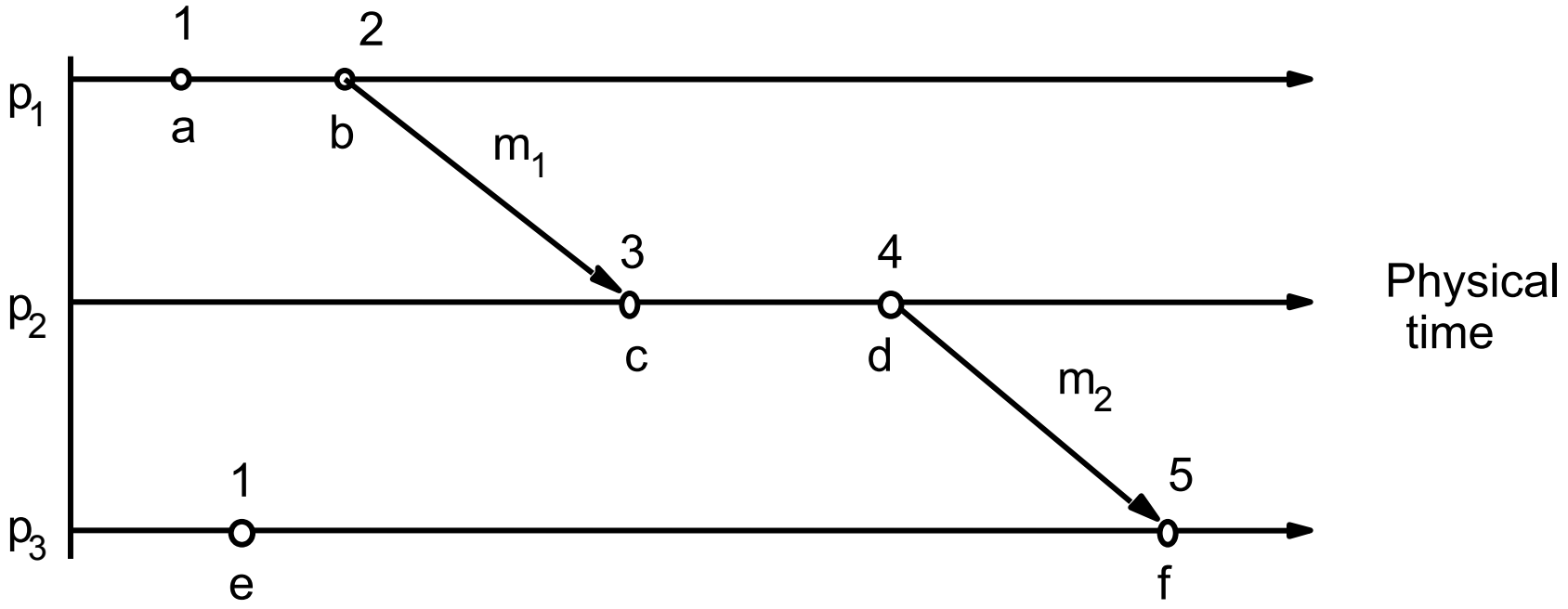
Find the Mistake: Lamport Logical Time



Corrected Example: Lamport Logical Time



Events occurring at three processes



Limitations of Lamport Clocks:

It represents a *partial order*, if $a \rightarrow b$ it implies that $L(a) < L(b)$ but the converse is not true, i.e., if $L(a) < L(b)$ it doesn't imply $a \rightarrow b$. From the fig. $L(b) > L(e)$ but in fact, $b \| e$.

Vector Logical Clocks

- With Lamport Logical Timestamp

  e → f ⇒ timestamp(e) < timestamp (f), but

  timestamp(e) < timestamp (f) ⇒ {e → f} OR {e and f concurrent}

- Vector Logical time addresses this issue:
  - N processes. Each uses a vector of counters (logical clocks), initially all zero. i^{th} element is the clock value for process i.
  - Each process i increments the i^{th} element of its vector upon an instruction or send event. Vector value is timestamp of the event.
  - A send(message) event carries its vector timestamp (counter vector)
  - For a receive(message) event,

$$V_{receiver}[j] = \begin{cases} \max(V_{receiver}[j], V_{message}[j]), & \text{if j is not self} \\ V_{receiver}[j] + 1 & \text{otherwise} \end{cases}$$

Vector Clocks (Mattern '89 & Fidge '91)

- **To overcome the problems of Lamport Clock ➔ Vector clocks**
  - **A *vector clock* for a system of *N* processes is an array of *N* integers. P_i keeps its VC V_i, to timestamp local events.**
  - **Vector clock rules:**

VC1: Initially for each process p_i, $V_i[j] = 0$, for $j = 1,2, \ldots, N$

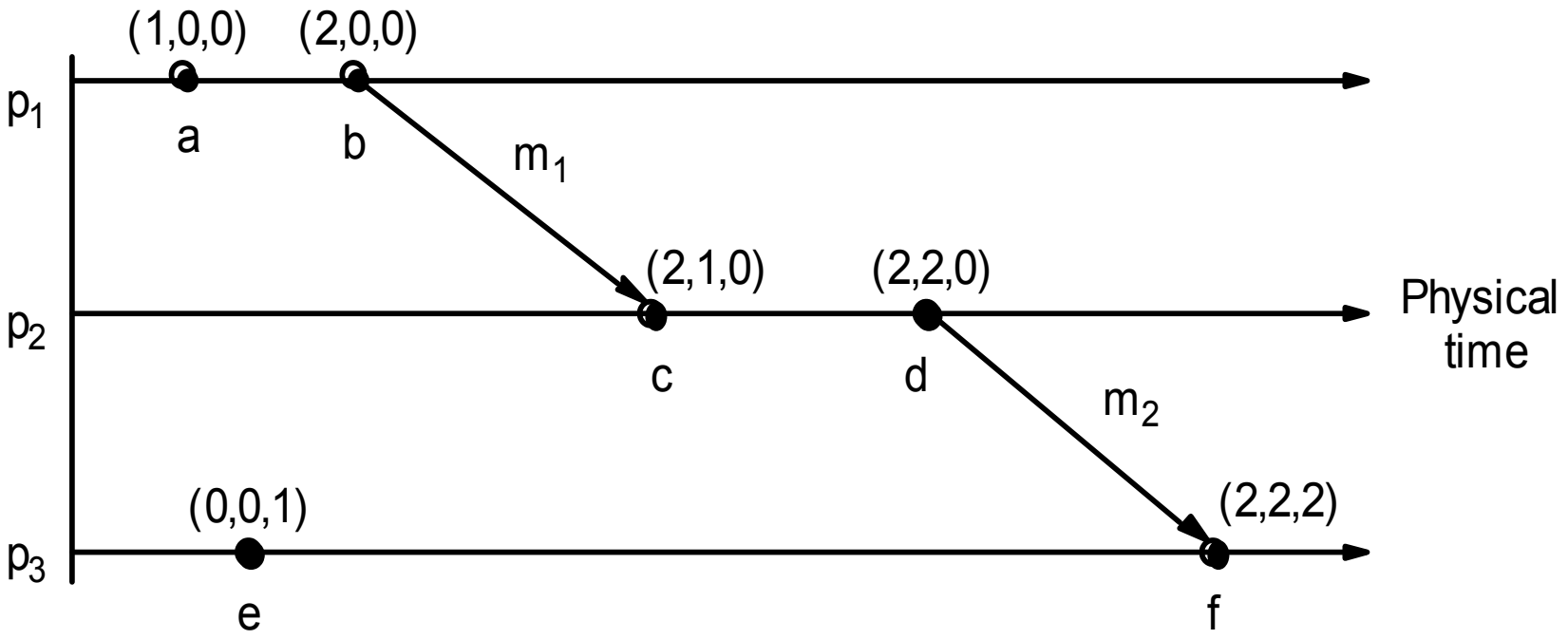
VC2: Just before time stamping an event, p_i sets $V_i[i] = V_i[i] + 1$.

VC3: p_i sends $t = V_i$ in every message it sends

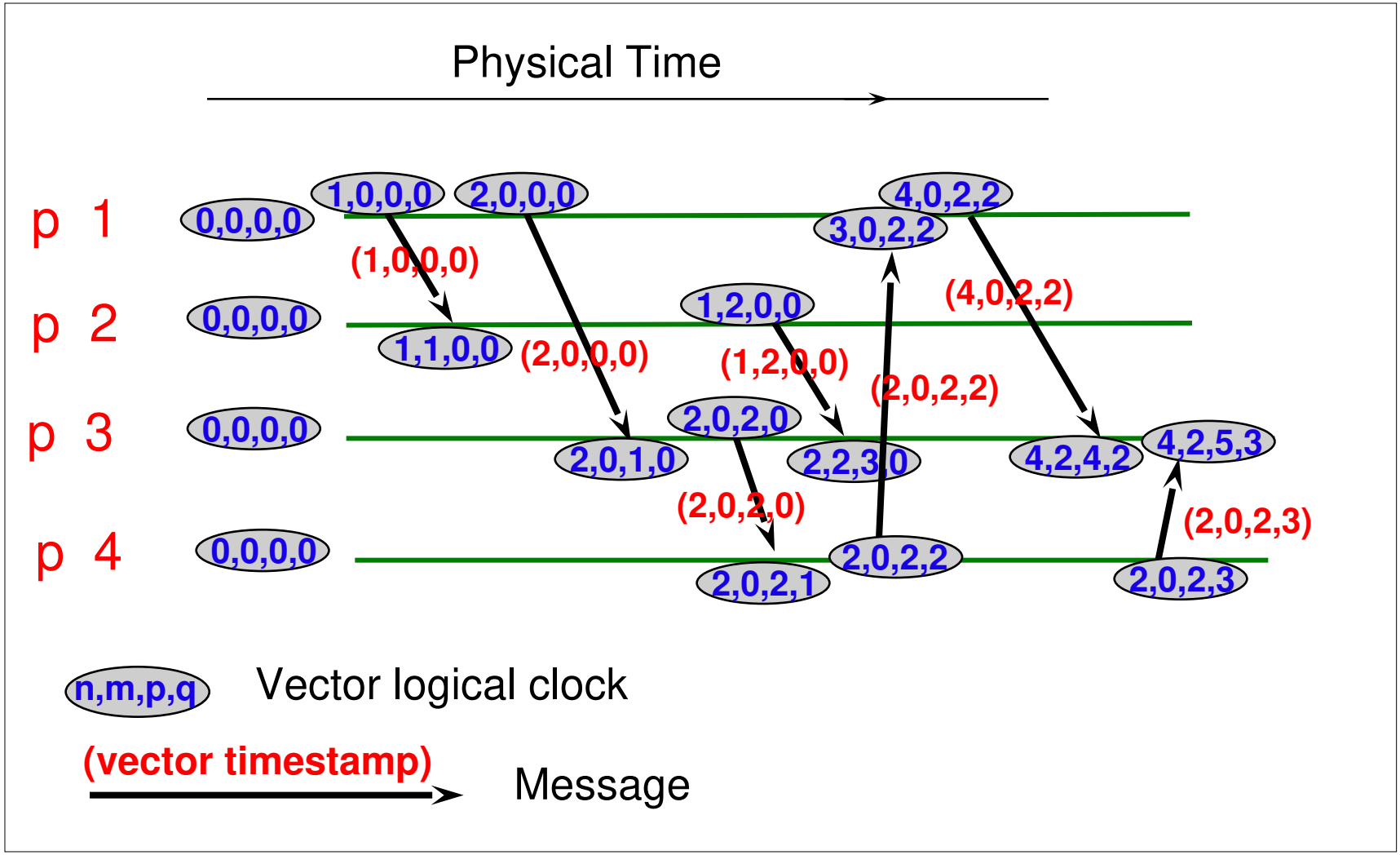
VC4: When p_i receives t in a message, it sets $V_i[j] = max(V_i[j], t[j])$, for $j = 1,2, \ldots, N$

(the max is computed component wise)

Vector Timestamps



Example: Vector Timestamps



Comparing Vector Timestamps

- $VT_1 = VT_2$,

  iff $VT_1[i] = VT_2[i]$, for all $i = 1, \ldots, n$

- $VT_1 < VT_2$,

  iff $VT_1[i] \leq VT_2[i]$, for all $i = 1, \ldots, n$

- $VT_1 < VT_2$,

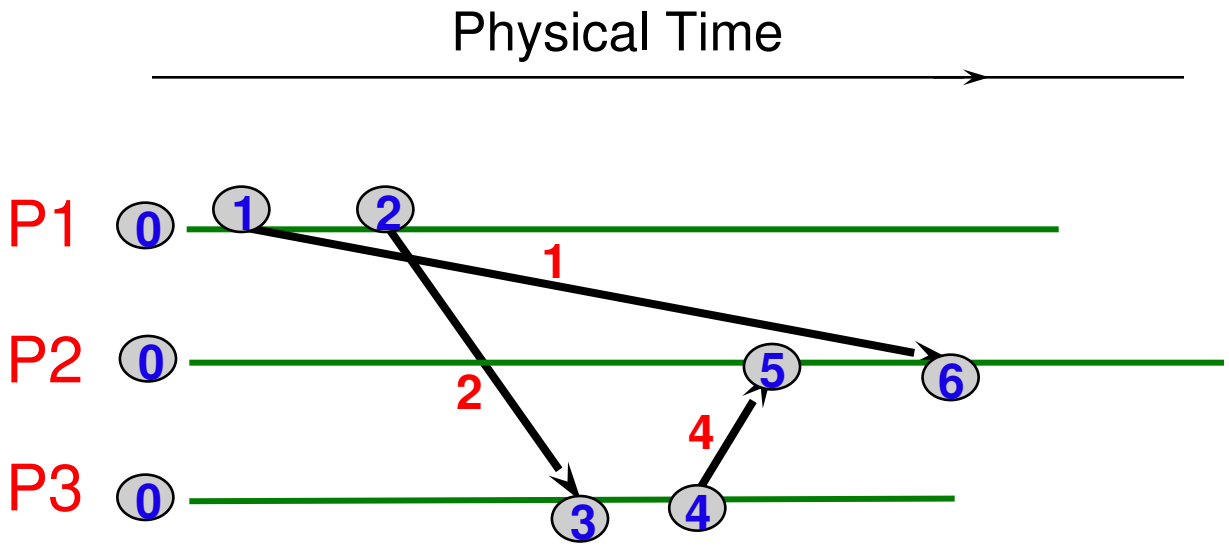
  iff $VT_1 \leq VT_2$ &

  $\exists j \; (1 \leq j < n \;\&\; VT_1[j] < VT_2[j])$

- **Then:** VT_1 is concurrent with VT_2

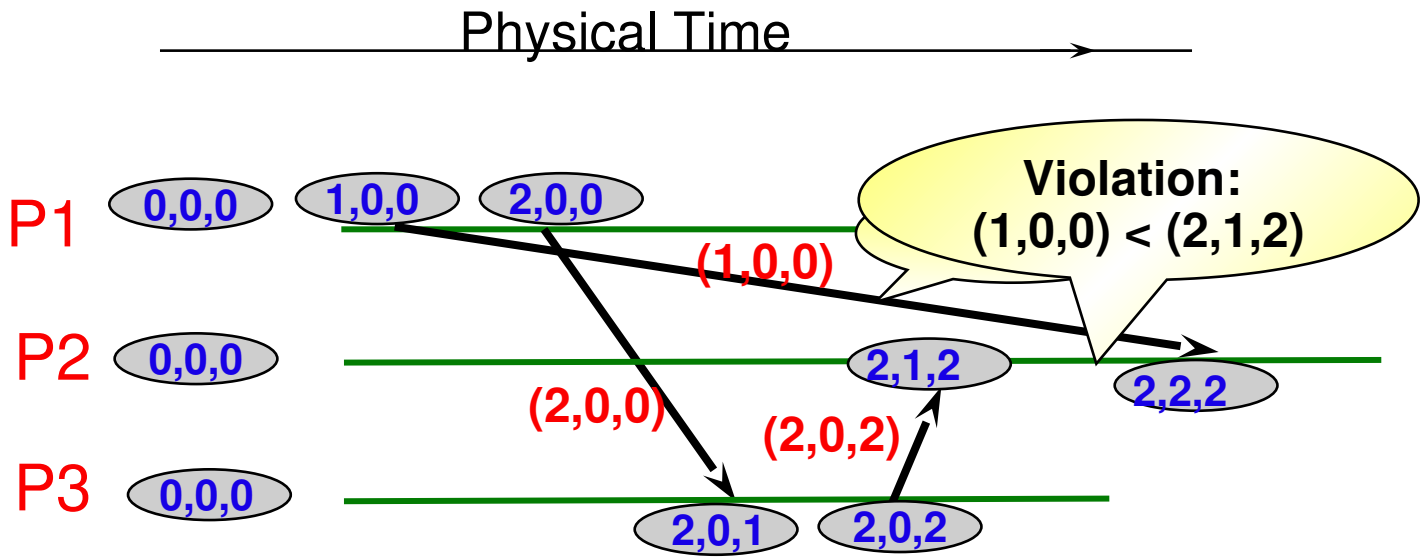
  iff (not $VT_1 < VT_2$ AND not $VT_2 < VT_1$)

Side Issue: Causality Violation



- **Causality violation occurs when order of messages causes an action based on information that another host has not yet received.**
- **In designing a distributed system, potential for causality violation is important to notice**

Detecting Causality Violation



- **Potential causality violation can be detected by vector timestamps.**
- **If the vector timestamp of a message is less than the local vector timestamp, on arrival, there is a potential causality violation.**

Summary

- **Time synchronization important for distributed systems**
  - **Cristian's algorithm**
  - **Berkeley algorithm**
  - **NTP**
- **Relative order of events enough for practical purposes**
  - **Lamport's logical clocks**
  - **Vector clocks**