Computer Science 425
Distributed Systems

CS 425 / CSE 424 / ECE 428

Fall 2012

Indranil Gupta (Indy)

September 11, 2012

Lecture 5

Time and Synchronization

Reading: Sections 14.1-14.4

Jack Dorsey in CS425 UIUC (Distributed Systems)

- **Co-founder of Twitter, and former CEO**
- **Founder and CEO of Square**
- **MIT TR35 Top 35 Innovators under 35, 2008**

- **Tech Talk today here at 7 pm (1310 DCL)**

Why synchronization?

- You want to catch the 13N Silver bus at the Illini Union stop at 6.05 pm, 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?

- Cloud airline reservation system
- Server A receives a client request to purchase last ticket on flight ABC 123.
- A timestamps purchase using local clock 9h:15m:32.45s, and logs it. Replies ok to client.
- 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.
  - 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)**
  - **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., distance/speed = time)

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,**

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

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

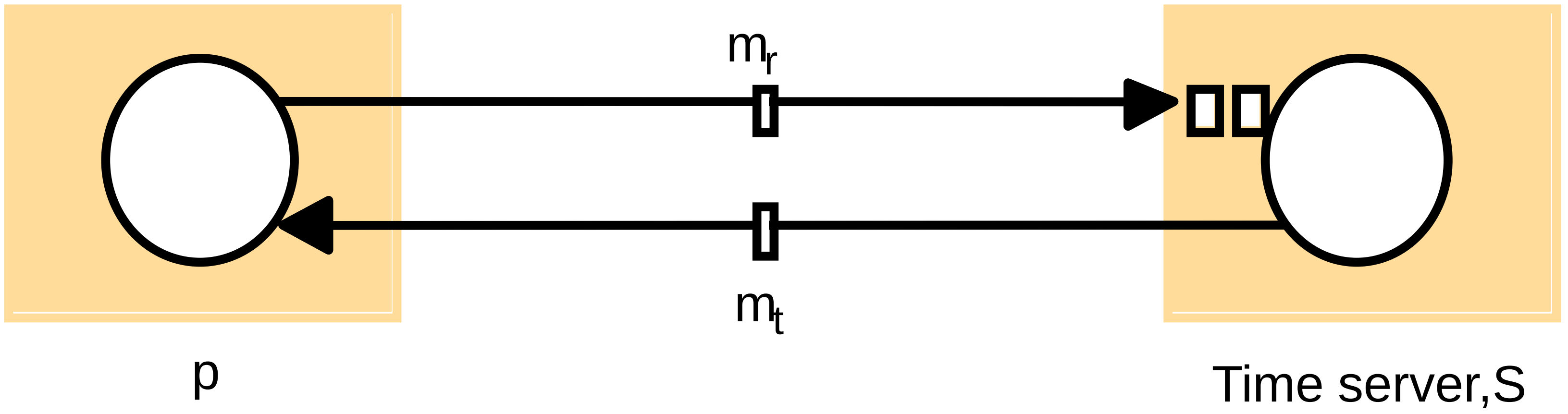
  Clocks C_i are externally accurate to within the bound *D*.
- **Internal synchronization: For a synchronization bound *D>0*,**

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

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

  Clocks C_i are internally accurate within the bound *D*.
- **External synchronization with *D* $\Rightarrow$ Internal synchronization with *2D***
- **Internal synchronization with D $\Rightarrow$ External synchronization with ??**

Clock Synchronization Using a Time Server



Cristian's Algorithm

- Uses a *time server* to synchronize clocks
- Time server keeps the reference time (say UTC)
- A client asks the time server for time, the server responds with its current time *T*, and the client uses this received value to set its clock
- But network round-trip time introduces an error…

Let *RTT = response-received-time – request-sent-time* (measurable at client)

Also, suppose we know: (1) the minimum value *min* of the client-server one-way transmission time [Depends on what?]

(2) and that the server timestamped the message at the *last* possible instant before sending it back

Then, the actual time could be between [T+min,T+RTT— min]

What are the two extremes?

Cristian's Algorithm (2)

- Client sets its clock to halfway between T+min and T+RTT— min i.e., at T+RTT/2
  - ☹ Expected (i.e., average) skew in client clock time will be = half of this interval = (RTT/2 – *min*)
- Can increase clock value, but should *never* decrease it – Why?
- Can adjust speed of clock too (take multiple readings) – either up or down is ok.
- For unusually long RTTs, repeat the time request
- For non-uniform RTTs, use *weighted average*

$$\text{avg-clock-error}_n = (w * \text{latest-clock-error}) + (1 - w) * \text{avg-clock-error}_{n-1}$$

Berkeley Algorithm

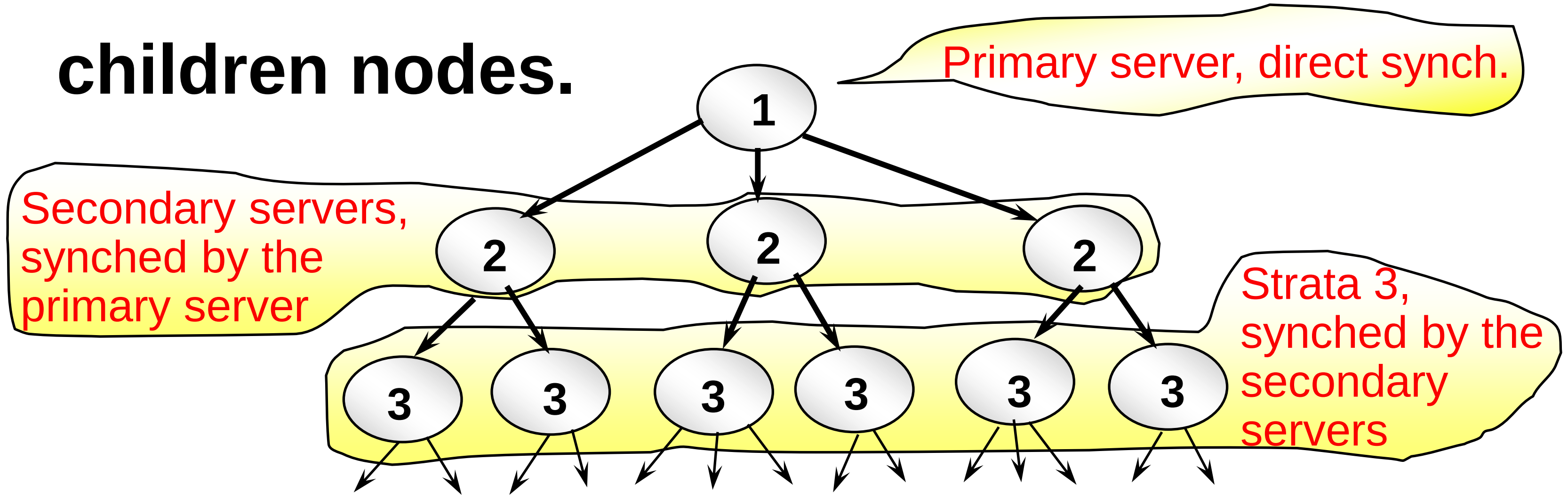
- **Uses an *elected master process* to synchronize among clients, without the presence of a time server**
- **The *elected master* broadcasts to all machines requesting for their time, adjusts times received for RTT & latency, averages times, and tells each machine how to adjust.**
- **Multiple leaders may also be used.**

☹ Averaging client′s clocks may cause the entire system to drift away from UTC over time (Internal Synchronization)

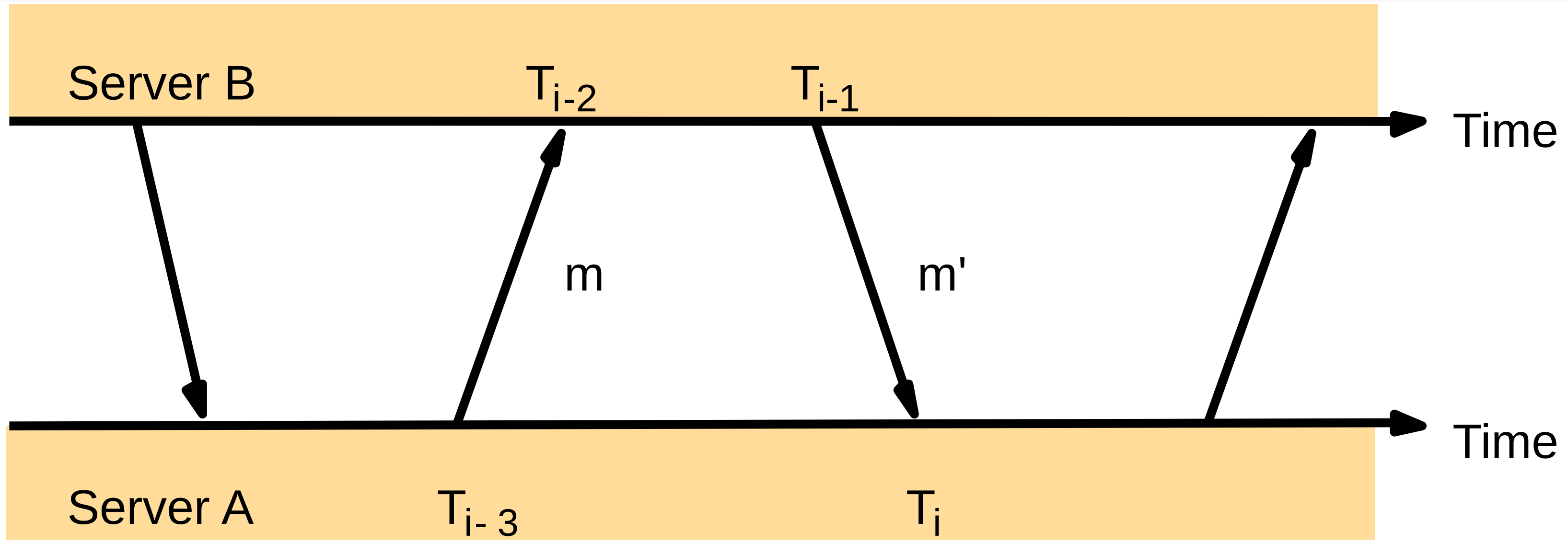
☹ Failure of the master requires some time for re-election, so drift/skew bounds cannot be guaranteed

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.**

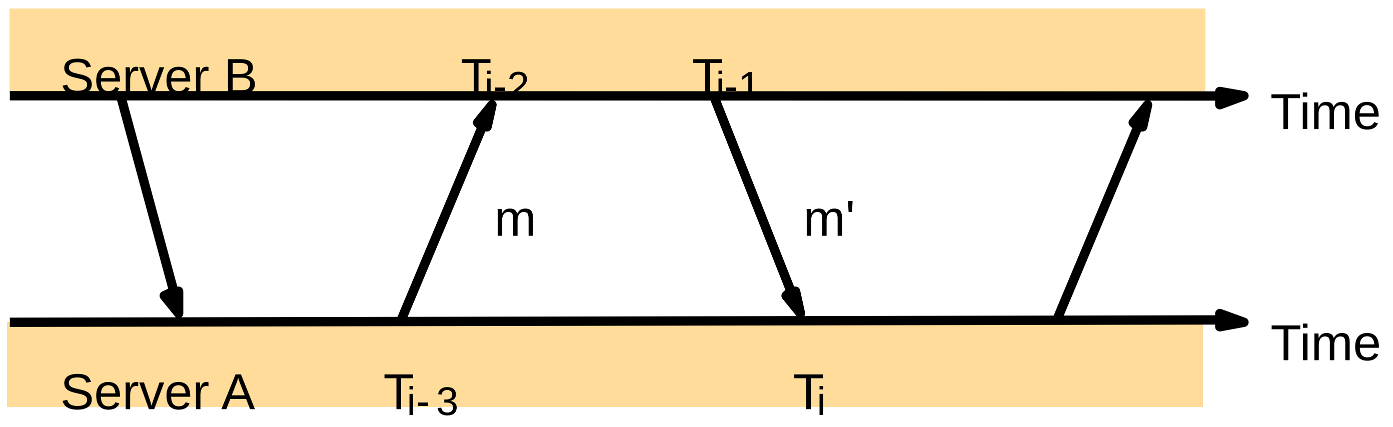

Primary server, direct synch.
1
Secondary servers, synched by the primary server
2
2
2
Strata 3, synched by the secondary servers
3
3
3
3
3
3

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



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

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

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

$$o_i - d_i/2 £ o £ o_i + d_i/2.$$

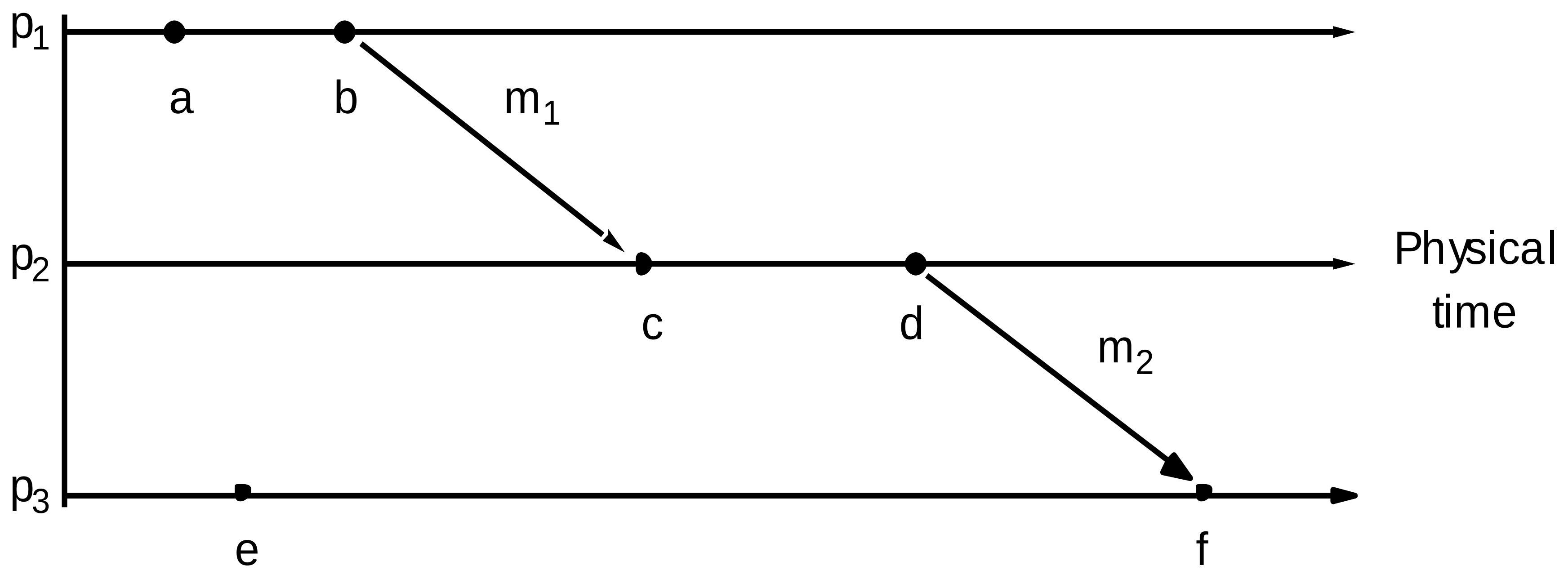
- *t* and *t′:* actual transmission times for *m* and *m′(unknown)*
- *o*: true offset of clock at *B* relative to clock at *A*
- o_i: estimate of actual offset between the two clocks
- d_i: estimate of accuracy of o_i ; total transmission times for *m* and *m′*; d_i=*t*+*t′*

Logical Clocks

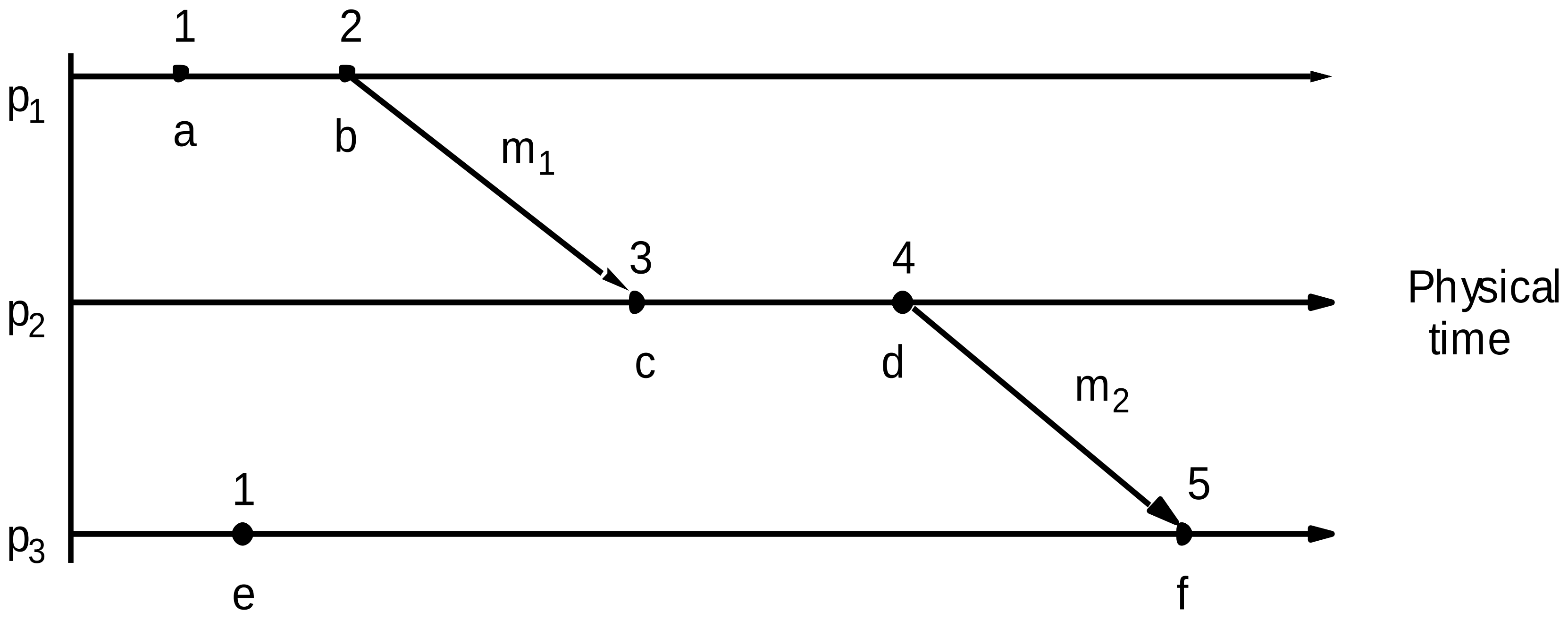
- Is it always necessary to give *absolute* time to events?
- Suppose we can assign *relative* time to events, in a way that does not violate their causality
  - Well, that would work – we humans run our lives without looking at our watches for everything we do
- First proposed by Leslie *Lamport* in the 70's
- Define a logical relation *Happens-Before* ($\rightarrow$) 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 $a \rightarrow c$
- 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

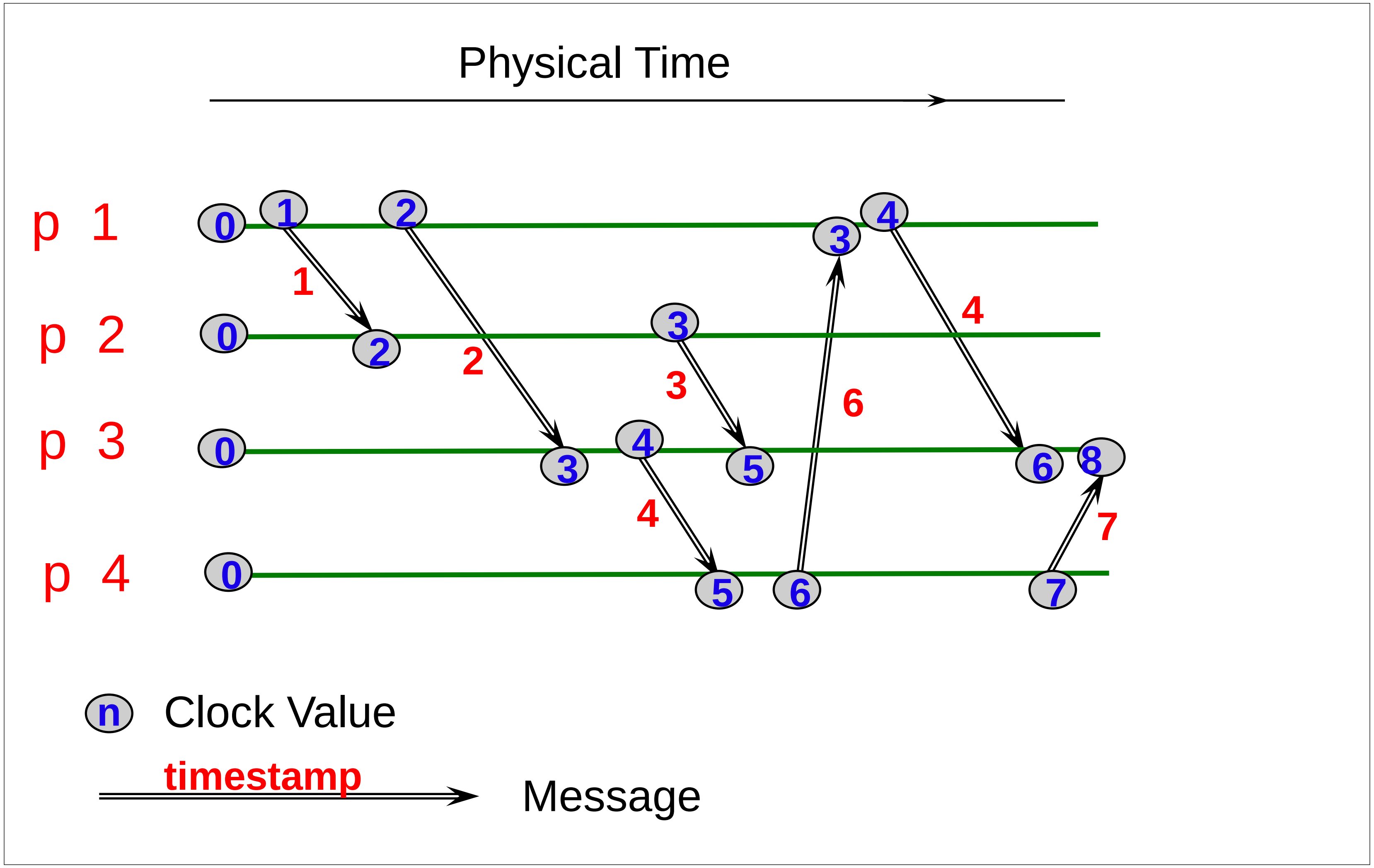
Events Occurring at Three Processes



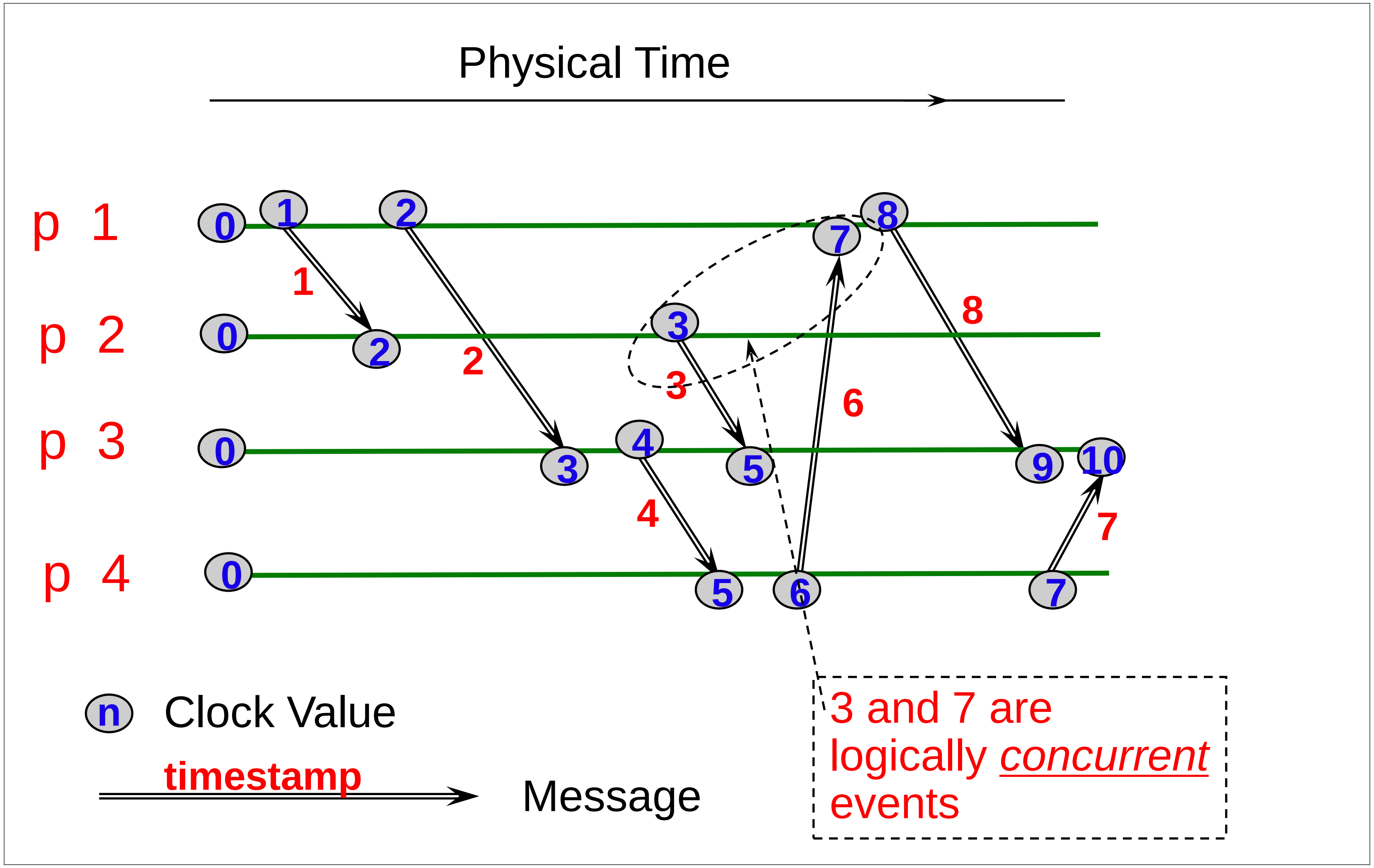
Lamport Timestamps



Find the Mistake: Lamport Logical Time



Corrected Example: Lamport Logical Time



3 and 7 are logically *concurrent* events

Vector Logical Clocks

- **With Lamport Logical Timestamp**

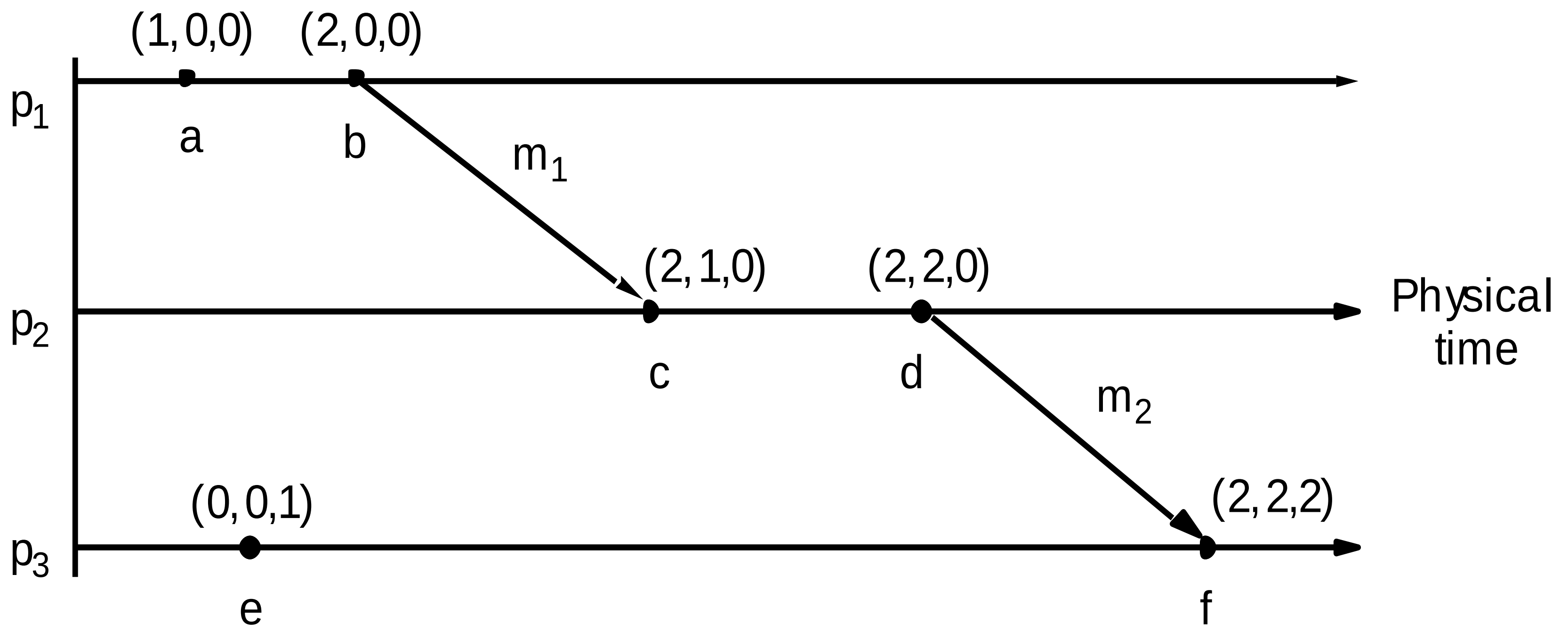
  $e \rightarrow f \Rightarrow$ timestamp(e) < timestamp (f), but

  timestamp(e) < timestamp (f) $\Rightarrow$ {e $\rightarrow$ f} OR {e and f concurrent}

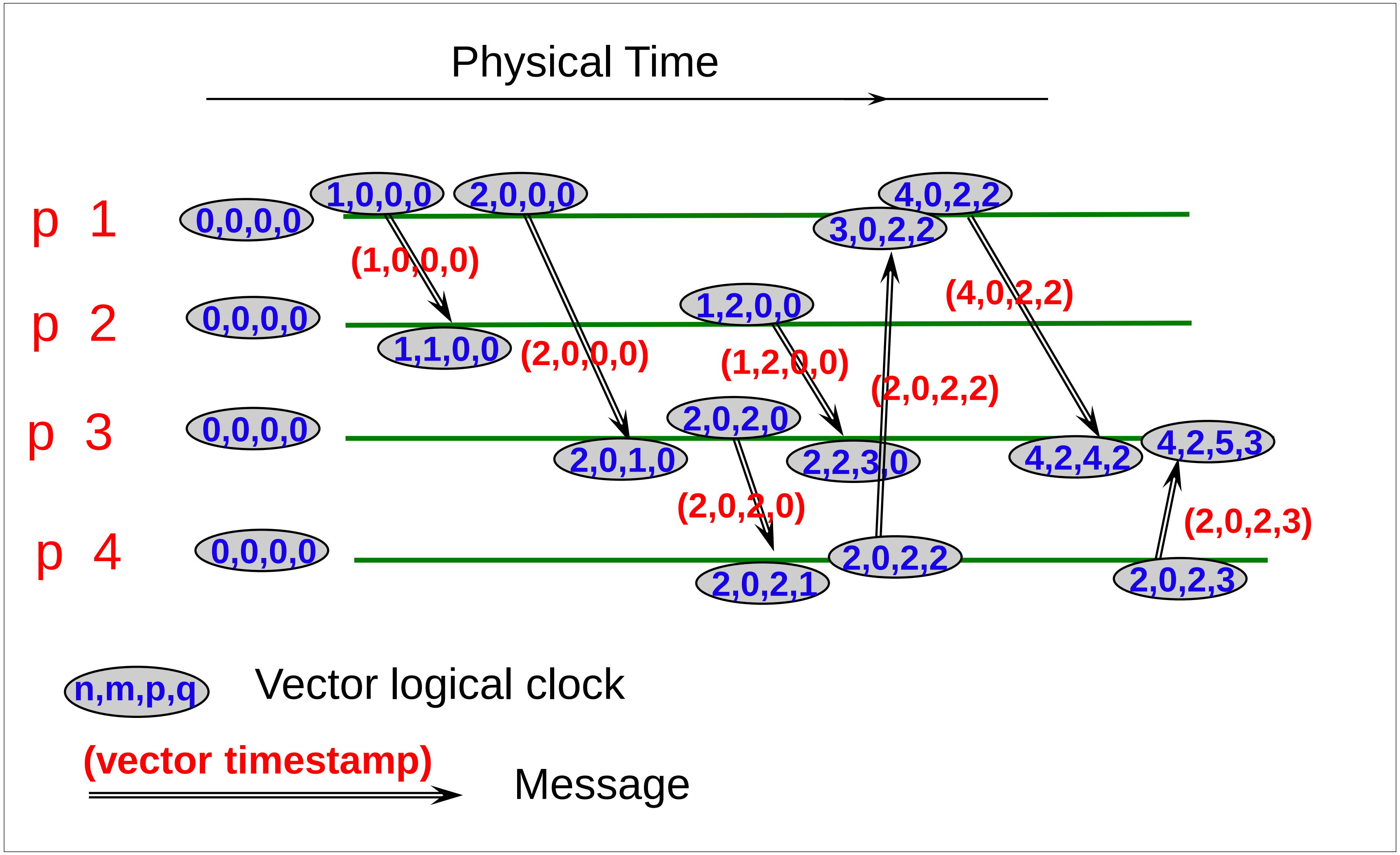
- **Vector Logical time addresses this issue:**
  - **N processes. Each uses a vector of counters (logical clocks), initially all zero. i^{th} element is the clock value for process i.**
  - **Each process i increments the i^{th} element of its vector upon an instruction or send event. Vector value is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

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

Vector Timestamps



Example: Vector Timestamps



Comparing Vector Timestamps

- $VT_1 = VT_2$,

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

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

  iff $VT_1 \leqslant VT_2$ &

  $\exists\, j\ (1 \leqslant j \leqslant n\ \&\ VT_1[j] < VT_2[j])$
- **Then:** VT_1 is concurrent with VT_2

  iff (not $VT_1 \leqslant VT_2$ AND not $VT_2 \leqslant VT_1$)

Summary, Announcements

- 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
- Next class: Global Snapshots. Reading: 14.5
- HW1 due next Thursday 9/20
- MP1: due next Sunday
  - By now, you should have written most of your code.