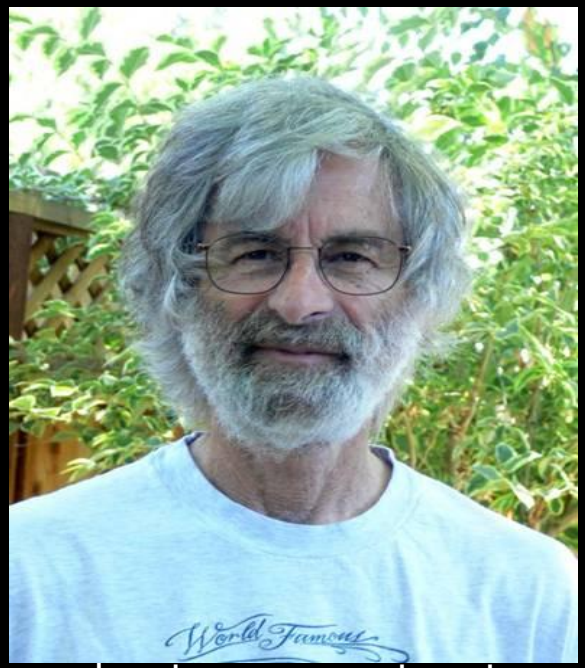
Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport, ACM 1978

CS523 Presentation by:
M.Fatih BOYACI and Firat KIYAK
University of Illinois at Urbana-Champaign

Background:



- Lessie Lamport
  - Phd, Math, 1972
  - MSR, Slicon Valley
  - TLA+, temporal logic of Actions.
    - Logic for specifying and reasoning about concurrent systems.
- Time, Clocks, Ordering
  - Most Cited
  - Origin of paper: The maintenance of Duplicate Databases
    - by Paul Johnson and Bob Thomas
    - Using timestamps in distributed algorithm
    - + Special Relativity
  - Dijkstra Prize in Dist. Computing in 2000.

Two System Models:

- Synchronous Distributed System
  - Each message is received within bounded time
  - Drift of each process' local clock has a known bound
  - Each step in a process takes lb < time < ub

  Ex:A collection of processors connected by a communication bus, e.g., a Cray supercomputer or a multicore machine
- Asynchronous Distributed System
  - No bounds on process execution
  - The drift rate of a clock is arbitrary
  - No bounds on message transmission delays

  Ex: The Internet is an asynchronous distributed system, so are ad-hoc and sensor networks

- *This is a more general (and thus challenging) model than the synchronous system model. A protocol for an asynchronous system will also work for a synchronous system (though not vice-versa)*

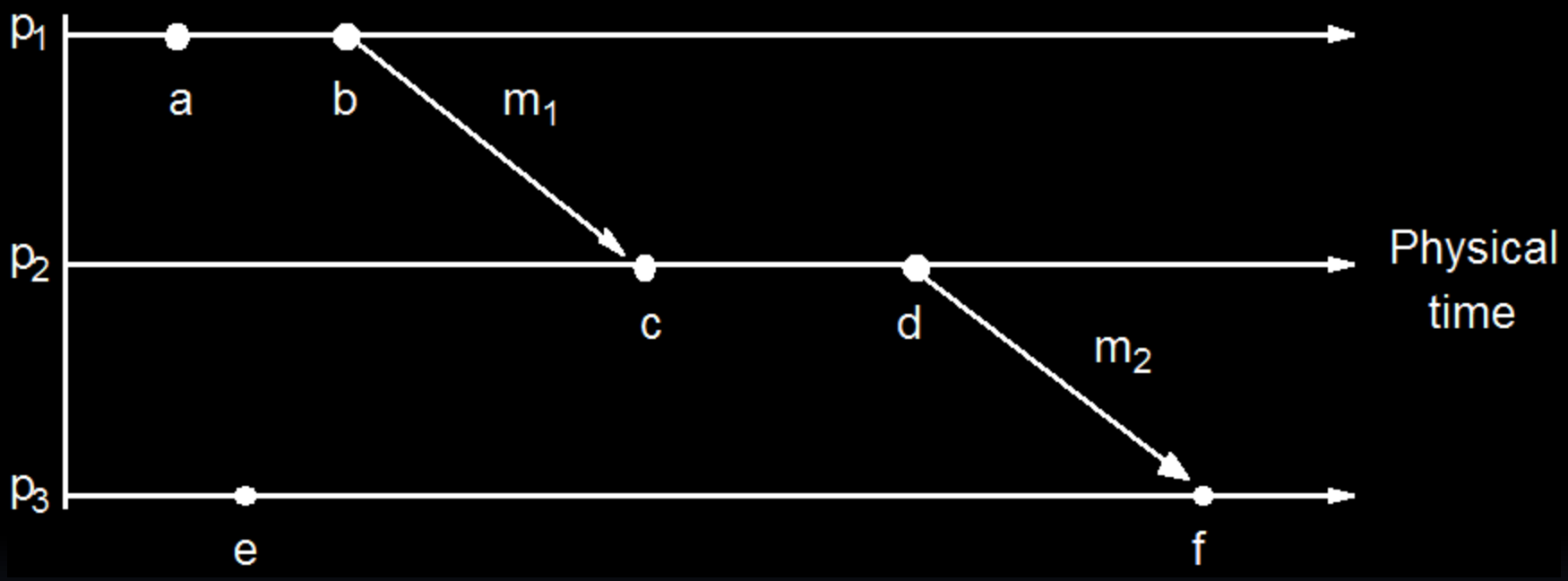
- It would be ***im***possible to accurately synchronize the clocks of two communicating processes in an asynchronous system

Logical Clocks:

- Lamport's **happens-before** ($\rightarrow$) among **events**:
  - On the same process: $a \rightarrow b$, if $time(a) < time(b)$
  - If p1 sends *m* to p2: $send(m) \rightarrow receive(m)$
  - If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
- Lamport's **logical timestamps** preserve causality:
  - All processes use a **local counter** (logical clock) with initial value of zero
  - Just before each **event**, the local counter is incremented by 1 and 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(receiver's-local-counter, message-timestamp) + 1

Example:



- *Logical timestamps preserve causality of events:*
  i.e., $a \rightarrow b$ ==> TS(a) < TS(b)

Scalar Logical Times: Pros and Cons

- **Advantages**
  - We get a total ordering of events in the system. All the benefits gained from knowing the causality of events in the system apply.
  - Small overhead: one integer per process.
- **Disadvantage**
  - Clocks are not strongly consistent: clocks lose track of the timestamp of the event on which they are dependent on. This is because we are using a single integer to store the local and logical time.

Vector Clocks:

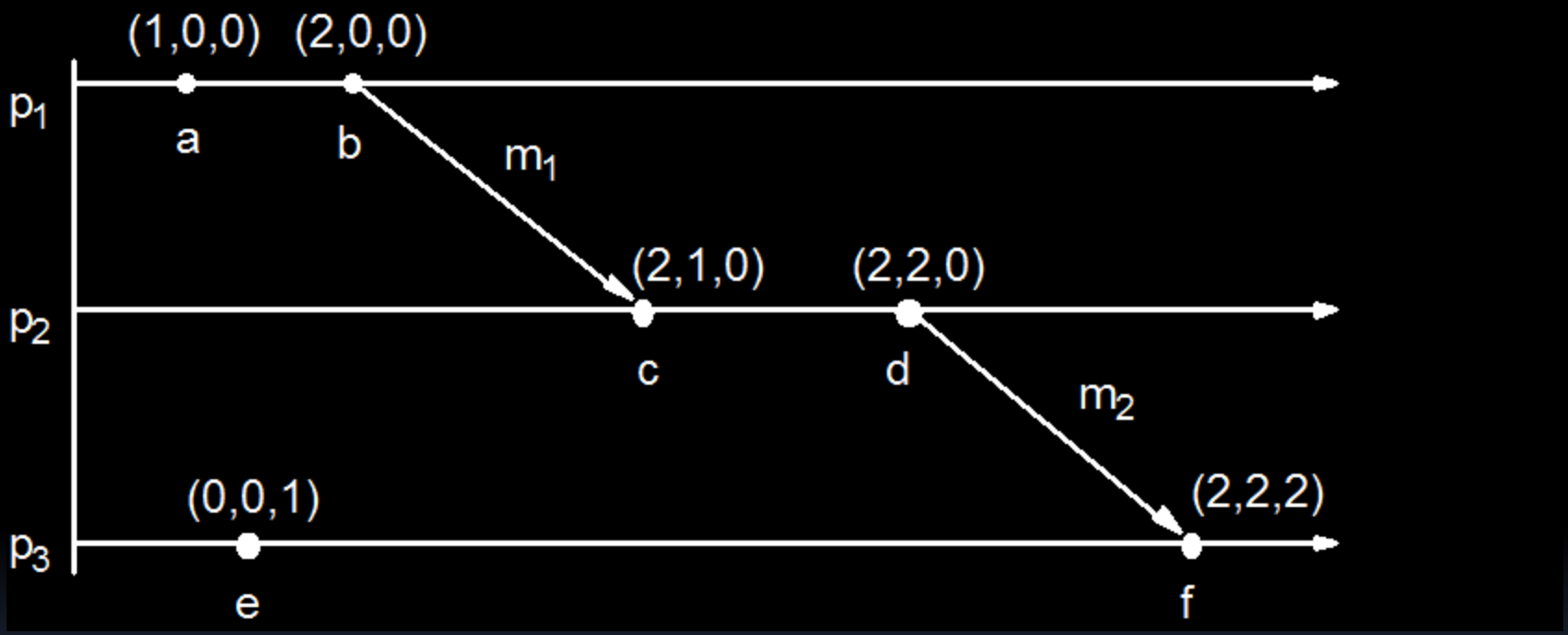
Each process P_i maintains a vector V_i with the following two properties:

1. $V_i[i]$ = no. of events that have occurred so far at P_i
2. If $V_i[j] = k$, then P_i knows that k events have occurred at P_j

- Occurrence of an event at process P_i causes an increment in $V_i[i]$ (i'th entry in its own vector clock)
- When P_i sends a message to P_j we have

  $V_j[k] = \max (V_j[k], V_i[k])$ (element-by-element)

Example:



Comparing Vector Timestamps:

- $VT_1 = VT_2$,

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

- $VT_1 <= VT_2$,

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

- $VT_1 < VT_2$,

  iff $VT_1 <= VT_2$ &

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

- VT_1 is concurrent with VT_2

  iff not ($VT_1 <= VT_2$ OR $VT_2 <= VT_1$)

Recommended Reading:

- Distributed snapshots: determining global states of distributed: systems (http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#chandy)
  - Can you capture (record) the states of all processes and communication channels at exactly 10:04:50 am?
  - Chandy and Lamport snapshot algorithm: records a *logical (or causal)* snapshot of the system.
- Synchronizing clocks in the presence of faults: (http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#clocks)

Discussion:

- What did you like/dislike about the paper?
- What future work did this inspire? (quite many ☺)

1. What are the known applications of logical ordering?
2. How to handle failures? Can this problem be solved with a reliable protocol to transfer messages? Or do we need to ping/hello all other processes? Centralized vs. Decentralized approaches.
3. Do physical clock synchronization algorithms actually tend to work like the one Lamport described?
4. Lamport Clock(LC) can be implemented very efficiently because each node needs to maintain a counter and messages exchanged by processes only need to carry a single value read from the sender's counter. Unfortunately, if communication between processes is infrequent, the clock value at one process does not meaningfully describe the progress of other processes. How can we solve this problem ?
5. The partial order leaves us with the need to agree on how to break ties -- how to resolve the ambiguities where we can't agree which event took place first -- and thus create a total order of events. We want to do so in a way that is *fair*, in other words, in a way that cannot be manipulated to the advantage of any particular party. How tie-breaking can be handled in a fair way for LC?
6. What are the possible security problems with LC and How to resolve it ?