Slides for Chapter 14: Time and Global State

From **Coulouris, Dollimore and Kindberg**
Distributed Systems: Concepts and Design
Edition 5, © Pearson Education 2011

Learning Objectives

- To understand the notions of physical and logical time and global states
- To understand the key features of Cristian's synchronization algorithm, the Berkeley algorithm.
- To understand the utility of logical clocks (Lamport and vector) and the rules for updating them and their limitations

How processes can synchronize

- Multiple processes must be able to cooperate in granting each other temporary exclusive access to a resource
- Also, multiple processes may need to agree on the ordering of events, such as whether message m_1 from process P was sent before or after message m_2 from process Q.

Centralized system

- Time is unambiguous
- If a process wants to know the time, it makes a system call and finds out
- If process *A* asks for the time and gets it and then process *B* asks for the time and gets it, the time that B was told will be later than the time that A was told.

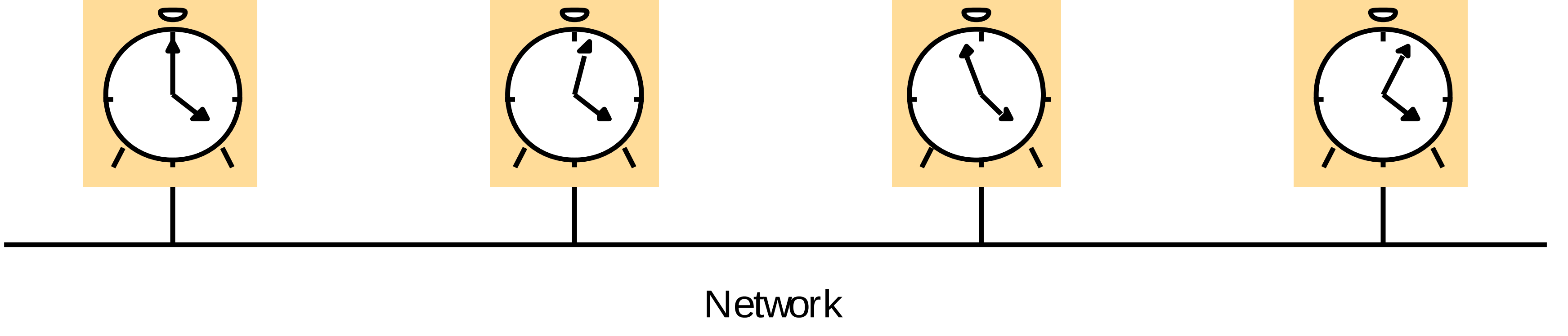
Physical Clocks

- Physical computer clocks are not clocks; they are timers
  - Quartz crystal that oscillates at a well-defined frequency that depends on physical properties
  - Two registers: counter and a holding register
  - Each oscillation decrements the counter by one
  - When counter reaches zero, generates an interrupt and the counter is reloaded from the holding register
  - Each interrupt is called a clock tick
- Interrupt service procedure adds 1 to time stored in memory so the software clock is kept up to date

The one and the many

- What if the clock is “off” by a little?
  - All processes on single machine use the same clock so they will still be internally consistent
  - What matters is relative time
- Impossible to guarantee that crystals in *different* computers run at exactly the same frequency
  - Gradually software clocks get out of synch -- skew
  - A program that expects time to be independent of the machine on which it is run ... fails

Skew between computer clocks in a distributed system



NIST and WWV

- NIST: National Institute of Standards and Technology
- WWV is the call sign of NIST's shortwave radio station located in Fort Collins, Colorado
- WWV's main function is the continuous dissemination of official U.S. Government time signals

Hey buddy, can you spare me a second?

- To provide UTC (Universal Coordinated Time) to those who need precise time, NIST operates a shortwave radio station WWV from Fort Collins, CO
- WWV broadcasts a short pulse at the start of each second
- There are stations in other countries plus satellites
- Using either shortwave or satellite services requires an accurate knowledge of the relative position of the sender and receiver.

To WWV or not to WWV

- If one computer has a WWV receiver, the goal is keeping all the others synchronized to it.
- If no machines have WWV receivers, each machine keeps track of its own time
  - Goal -- keep all machines together as well as possible
  - There are many algorithms

Underlying model for synchronization models

- Each machine has a timer that interrupts H times a second
  - Interrupt handler adds 1 to a software clock that keeps track of the number of ticks since some agreed-upon time in the past
  - Call the value of the clock C
- Notationally, when UTC time is t, the value of the clock on machine p is $C_p(t)$
- In a perfect world, $C_p(t) = t$ for all p and all t

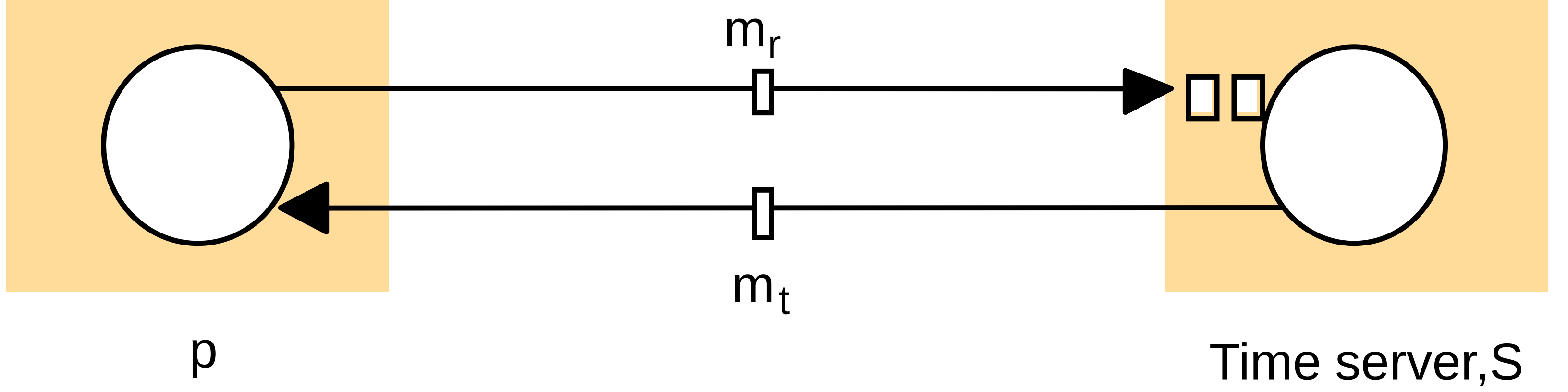
Back to reality

- Theoretically, a timer with H=60 should generate 216,000 ticks per hour
- Relative error is about 10^-5 meaning a particular machine gets a value in the range 215,998 to 216,002
- There is a constant called the maximum drift rate and a timer will work with "perfect" $\pm$ maximum drift rate.
- If two clocks are drifting in the opposite direction at a time delta-t after they were synchronized
  - may be as much as twice the max drift rate apart
  - To differ by no more than delta, clocks must be resynchronized every (delta/2*max-drift-rate) seconds

Cristian's algorithm (1)

- Well suited to one machine with a WWV receiver and a goal to have all other machines stay synchronized with it.
- Call the one with the WWV receiver the *time server*
- Periodically, each machine sends a message to the time server asking for the current time
- Machine responds with C_{UTC} as fast as it can

Clock synchronization using a time server



Cristian's algorithm (2)

- T_{round} : round-trip time taken to send the request m_r and receive the reply m_t
- T_{round} is in the order of 1-10 milliseconds on a LAN
- A clock with a drift rate of 10^{-6} seconds/second is sufficient
- A simple estimate of the time to which *p* should set its clock is $t + T_{round}/2$, assuming that the elapsed time is split equally before and after S placed *t* in m_t
- What is the problem?

Big Trouble

- Major problem
  - The single time server becomes bottleneck (multiple time servers can be used)
  - A faulty time server can reply an incorrect time
  - If sender’s clock was fast, C_{UTC} will be smaller than the sender’s current value of C
  - Change must be introduced gradually
    - If timer generates 100 interrupts/second, each interrupt adds 10 ms to the time
    - To slow down, ISR adds only 9 ms until correct
    - To speed up, add 11 ms at each interrupt

Little Trouble

- Minor problem
  - Takes a nonzero amount of time for the time server's reply to get back to the sender
  - Delay may be large and vary with network load
- To improve accuracy, measure several and average

If no WWV Receiver

- Berkeley UNIX algorithm
- The time server (actually time daemon) is active, not passive
- It polls every machine and asks what time it is
- Based on answers, it computes an average time and tells all machines to adjust their clocks to the new time
- The time daemon's time is set manually by the operator periodically
- Centralized algorithm though the time daemon does not have a WWV receiver

Berkeley Algorithm

- It eliminates readings from faulty clocks
- The master takes a fault-tolerant average, a subset of clocks is chosen that do not differ from one another by more than a specified amount
- If the master fails, another can be elected to take over

Decentralized synchronization

- Cristian and Berkeley UNIX are centralized algorithms with the usual downside.
- They are intended primarily for use within intranets
- There are several decentralized algorithms, for example:
  - Divide time into fixed length resynchronization intervals
  - At the beginning of each interval, every machine broadcasts its current time
  - Each starts a local timer to collect all broadcasts arriving during a certain interval
  - Algorithm to compute a new time based on some/all

Internet Synchronization

- New hardware and software technology in the past few years make it possible to keep millions of clocks synchronized to within a few ms of UTC
- New algorithms using these synchronized clocks are beginning to appear
- Synchronized clocks can be used
  - to achieve cache consistency
  - to use time-out tickets in distributed system authentication
  - to handle commitment in atomic transactions

Logical Clocks

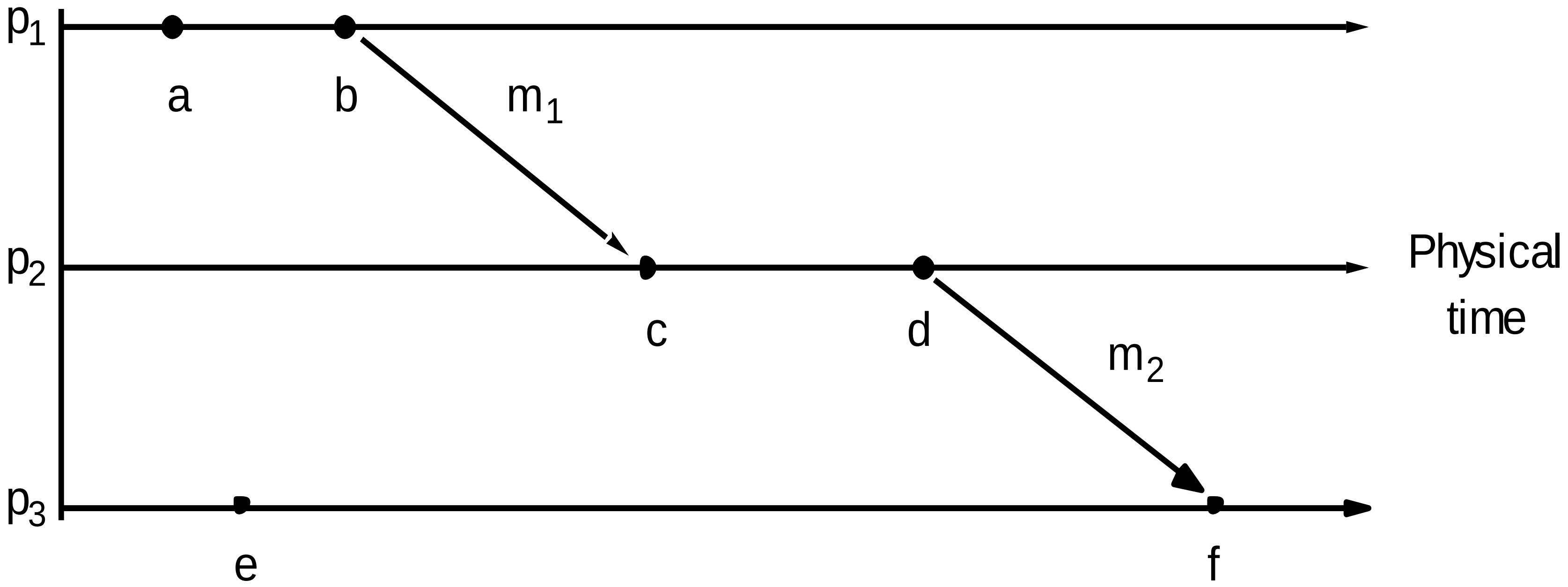
- For many purposes, it is sufficient that machines agree on the same time even if it is not the "right" time
- Internal consistency of the clocks matters
- Clock synchronization is possible but does not have to be absolute
  - If 2 processes do not interact, their clocks need not be synchronized; the lack of synch would not be seen
  - What is important is that all processes agree on the *order* in which events occur

Lamport timestamps

- *a happens-before b* means that all processes agree that first event *a* occurs, then afterward, event *b* occurs
- We write *a happens-before b* as *a* --> *b*
- If *a* occurs before *b* in the same process, we say *a* --> *b* is true
- If the event *a* sends a message and event *b* receives that message in another process, *a* --> *b* is also true because a message cannot be received until after it is sent.
- *happens-before* is transitive

Events occurring at three processes

What we can say?



We can say that a -> f

We cannot say …

- If x and y happen in different processes that do not exchange messages, then
  - we cannot say x --> y
  - we cannot say y --> x
  - nothing can be said about when the events happened or which event happened first
  - we call these events *concurrent*: a and e occur at different processes and there’s no chain of messages intervening between them. We say that a || e

Invent time

- Need a way of measuring time so that for every event we can assign a time C(*a*) on which all processes agree.
  - Such that, if *a* --> *b*, then C(*a*) < C(*b*)
  - If *a* and *b* are two events in the same process and *a* happens before *b*, then C(*a*) < C(*b*)
  - If *a* is the sending of a msg by one process and *b* is the receiving of that msg by another, then C(*a*) and C(*b*) must be assigned so that everyone agrees on the values of C(*a*) and C(*b*) with C(*a*) < C(*b*)
  - Corrections to C can only be made by addition, never subtraction so that the clock time always goes forward

If msg leaves at time N, it arrives at >= N+1

- Each message carries the time according to its sender's clock
- When it arrives, if the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be 1 more than the sending time
- Between every two events the clock must tick at least once
  - If a process sends or receives 2 messages in quick succession, it must advance its clock by (at least) 1 tick in between
  - Sometimes: no 2 events ever occur at exactly the same time

Lamport Algorithm

- LC1: Li is incremented before each event is issued at process pi:

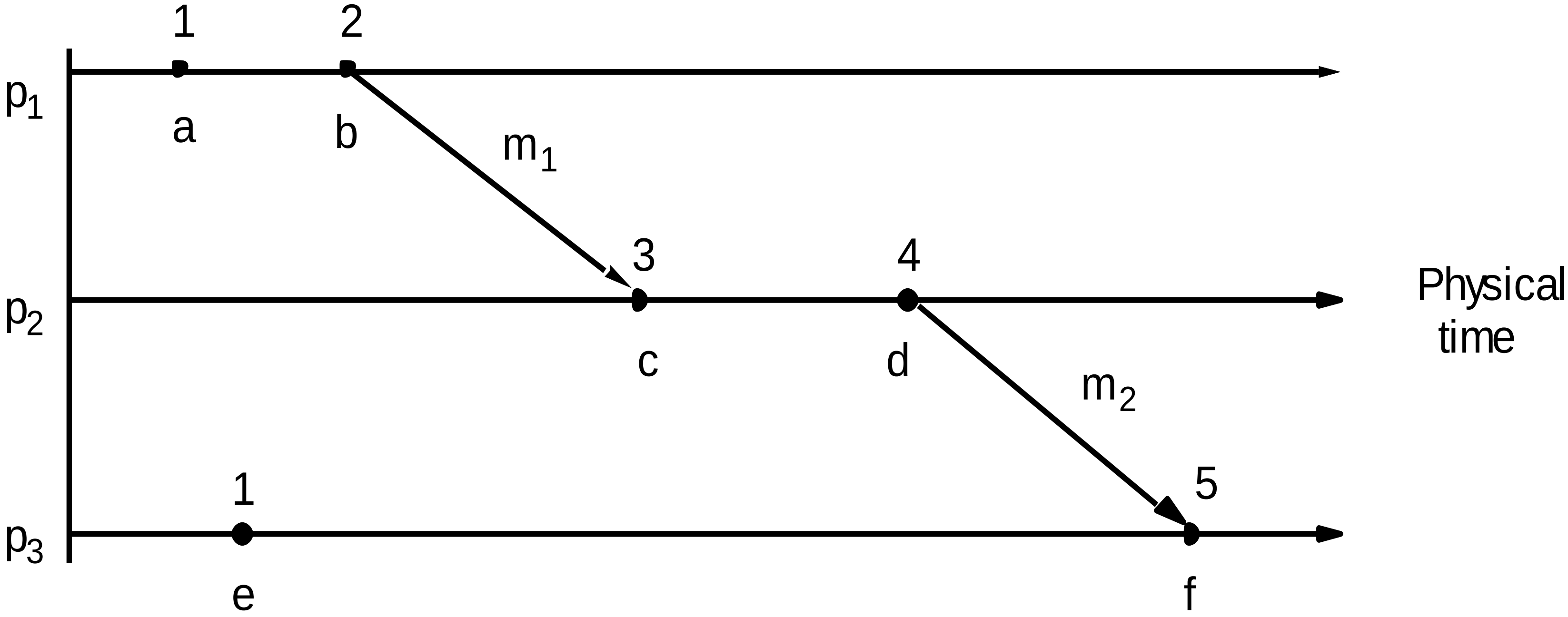
  Li:=Li+1

- LC2: (a) When Pi sends a message m, it piggybacks on m the value t=Li.

  (b) On receiving (m,t), a process pj computes Lj:=max(Lj,t) and then applies LC1 before timestamping the event receive(m).

Lamport timestamps for the events

L(b) > L(e) but b || e.



Each of the processes has its logical clock initialized to 0.

e -> e' => L(e) < L(e'), correct?

The converse is also correct? How about b and e?

Totally-ordered Multicast

- Consider a bank with replicated data in San Francisco and New York City.
- Customer in SF wants to add $100 to the account of $1000
- Meanwhile, a bank employee in NY initiates an update by which the customer's account will be increased with 1% interest.
- Due to communication delays, the instructions could arrive at the replicated sites in different orders with differing final answers
- Should have been performed at both sites in same order

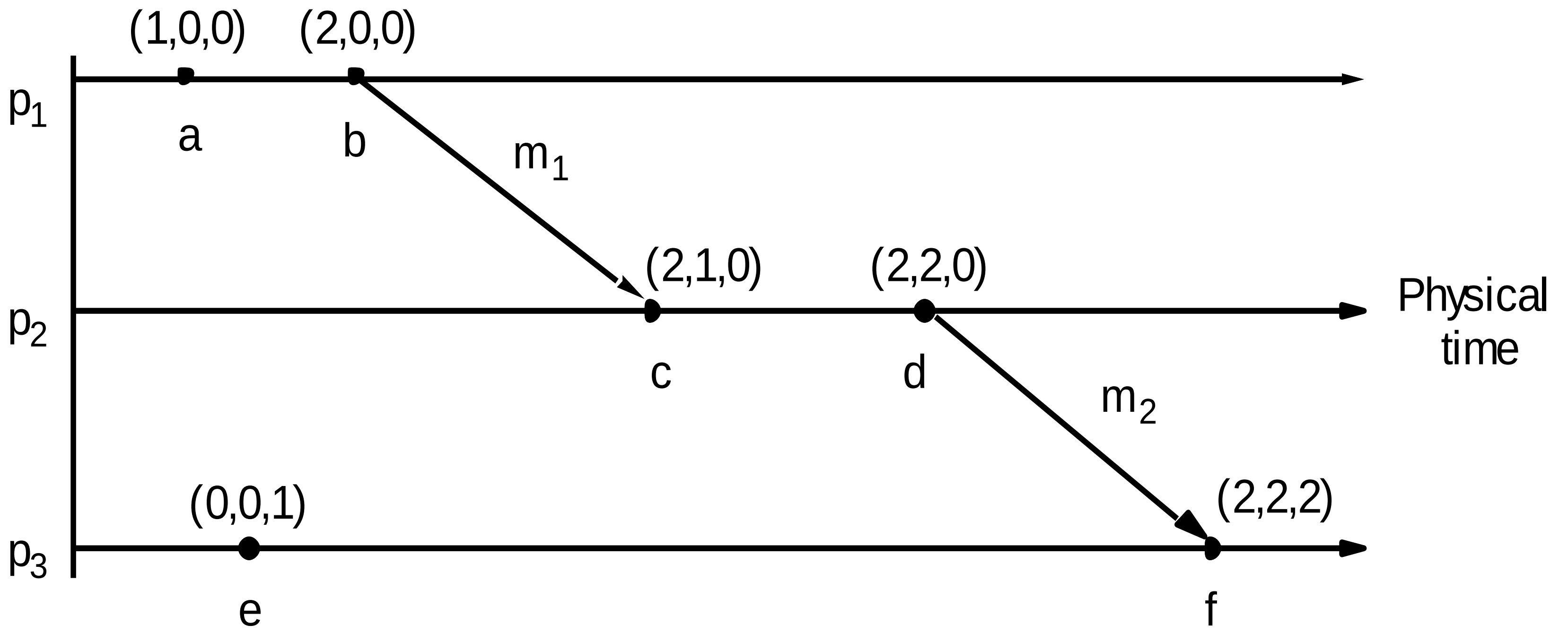
Limitation of Lamport Timestamps

- With Lamport timestamps, *nothing* can be said about the relationship between *a* and *b* simply by comparing their timestamps C(*a*) and C(*b*).
  - Just because C(*a*) < C(*b*), doesn't mean *a* happened before *b* (remember *concurrent* events)

Vector Clock

- A vector clock for a system of N processes is an array of N integers
- Each process keeps its own vector clock V_i, which it uses to timestamp local event
- Processes piggyback vector timestamps on the messages they send to one another

Vector timestamps for the events



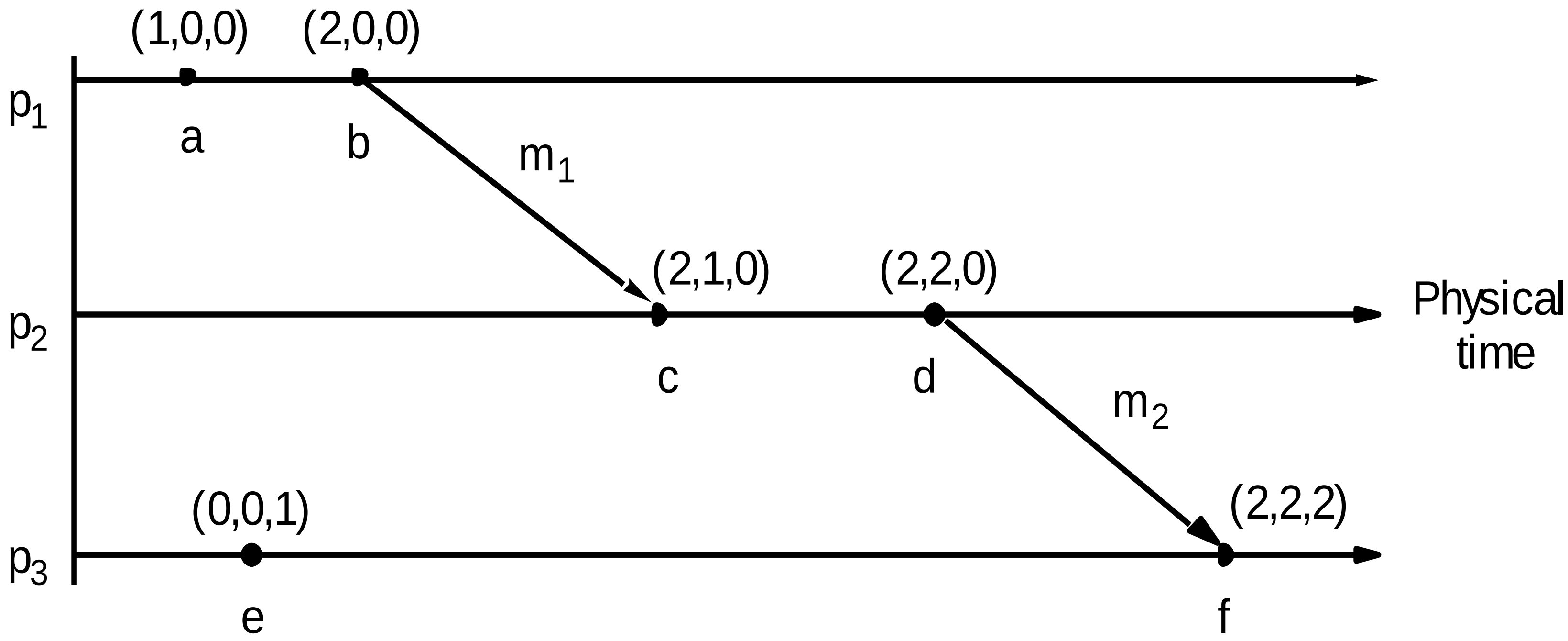
Vector clocks

- Lamport clocks: $L(e) < L(e')$ doesn't imply $e \rightarrow e'$
- each process keeps its own vector clock V_i
- piggyback timestamps on messages
- updating vector clocks:
  - VC1: Initially, $V_i[\ j\ ] := 0$ for p_i, j=1.. N (N processes)
  - VC2: before p_i timestamps an event, $V_i[\ i\ ] := V_i[\ i\ ]+1$
  - VC3: p_i piggybacks $t = V_i$ on every message it sends
  - VC4: when p_i receives a timestamp t, it sets $V_i[\ j\ ] := \max(V_i[\ j\ ]\ ,\ t[\ j\ ])$ for j=1..N (merge operation)

Vector clocks

- At p_i
  - $V_i[i]$ is the number of events p_i timestamped
  - $V_i[j]$ (j≠i) is the number of events that have occurred at p_j that p_i has potentially been affected by
  - Could more events than $V_i[j]$ have occurred at p_j? Yes or No

Vector timestamps (Fig 14.7)



V(a) < V(f), which tells us that a -> f

c||e can be seen from the fact that neither V(c)<=V(e) nor V(e)<=V(c)

Comparing vector timestamps

- $V = V'$ iff
  - $V[j] = V'[j], \quad j = 1 \,..\, N$
- $V <= V'$ iff
  - $V[j] <= V'[j], \quad j = 1 \,..\, N$
- $V < V'$ iff
  - $V <= V'$ and $V \neq V'$
    - Different from less than in all elements

Vector timestamps

- if $e \rightarrow e'$, then $V(e) < V(e')$
- if $V(e) < V(e')$, then $e \rightarrow e'$. (Exercise 14.13)
  - Figure 14.7
    - neither $V(c) <= V(e)$ nor $V(c) >= V(e)$
    - $c \,||\, e$
- Disadvantage compared to Lamport timestamps?

Taking up an amount of storage and message payload that is proportional to N

Assignment#2 (chapter 14)

- 14.1
- 14.2
- 14.4
- 14.13