Synchronization

CS432: Distributed Systems
Spring 2017

Reading

- Chapter 14 (14.2,14.4,14.5) [Coulouris ’11]
- Chapter 6 (6.1,6.2,6.3) [Tanenbaum ‘06]

Outline

- **Clocks**
- Logical Clocks
- Global State
- Mutual Exclusion

Time

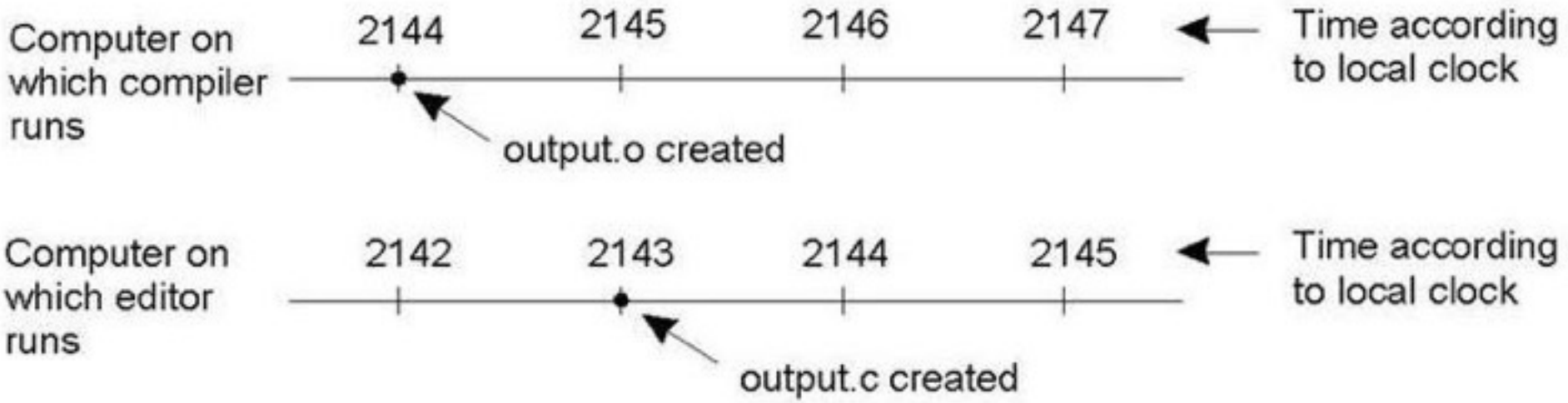
- Time is an important and interesting issue in distributed systems
  - Measure accurately what time of day a particular event occurred at a particular computer
  - Many distributed systems algorithm rely on time / clock synchronization
- Examples:
  - Mutual exclusion in a distributed system
  - Agree on the relative of ordering events
- Our objective is to study:
  - Clocks, synchronizing them
  - Capture a global state of a system

Clock – Centralized System

- In a centralized system, time is unambiguous
  - A process issues a system call to get the time
  - If process A gets time t_A, then process B gets time t_B. It is guaranteed that $t_B > t_A$
- Why time is important?

  Take Unix Make as an example, a change to one source file only requires one file to be recompiled, not all the files
  - if time(input.c) > time(input.o) => re-compile
  - if time(input.c) <= time(input.o) => no compilation

Clock — Distributed System



When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time

Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms. Prentice-Hall, Inc. 2002

Physical Clocks

- All computers have a circuit for tracking time (timer)
  - Oscillates with a frequency to generate 60 interrupts a second (clock tick)
  - **Counter register** is updated each oscillation until it reaches zero, an interrupt is issued
  - The value is the counter register is reset by loading the value from another **holding register**
  - When the first first starts, user enters date and time, which is converted to number of ticks since another stored date in memory
  - Battery-backed CMOS RAM is used to store date and time
- Challenge: it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency (clock skew)

Coordinating Universal Time

- Objective: clocks on different machines might not be exactly the same, but they should not deviate too much from real-time
- Universal Coordinated Time UTC
  - Atomic clocks were invented
  - Various labs have multiple clocks and send it to BIH (Bureau International de l'Heure) in Paris
  - BIH averages them to produce the TAI (Temps Atomique International)
  - Leap second is also introduced, it is inserted – or, more rarely, deleted – occasionally to keep it in step with astronomical time
- Algorithms to synchronize clocks: Cristian's method, Berkeley algorithm

Outline

- Clocks
- **Logical Clocks**
- Global State
- Mutual Exclusion

Logical Time and Logical Clocks

- Lamport [1978]
  - Since we cannot synchronize clocks perfectly across a distributed system, we cannot in general use physical time to find out the order of any arbitrary pair of events occurring within it
  - 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
- Make example:
  - What counts is whether input.c is older or newer than input.o, not their absolute creation times
  - What matters is keeping track of each others events: input.o is updated

Lamport's Logical Clocks

- Happens-before relation:
  - If *a* and *b* are events in the same process, and *a* happens-before *b*, then *a*—>*b* is true
  - If *a* is the event of a message being sent by one process, and *b* is the event of the same message being received by another process, then *a*—>*b* is also true
  - If two events, *a* and *b*, happen in different processes that do not exchange messages , then *a*—>*b* is not true, but neither is *b*—>*a*. These events are said to be concurrent (*a*||*b*)
  - happens-before is transitive: *a*—>*b* and *b*—>*c* => *a*—>*c*

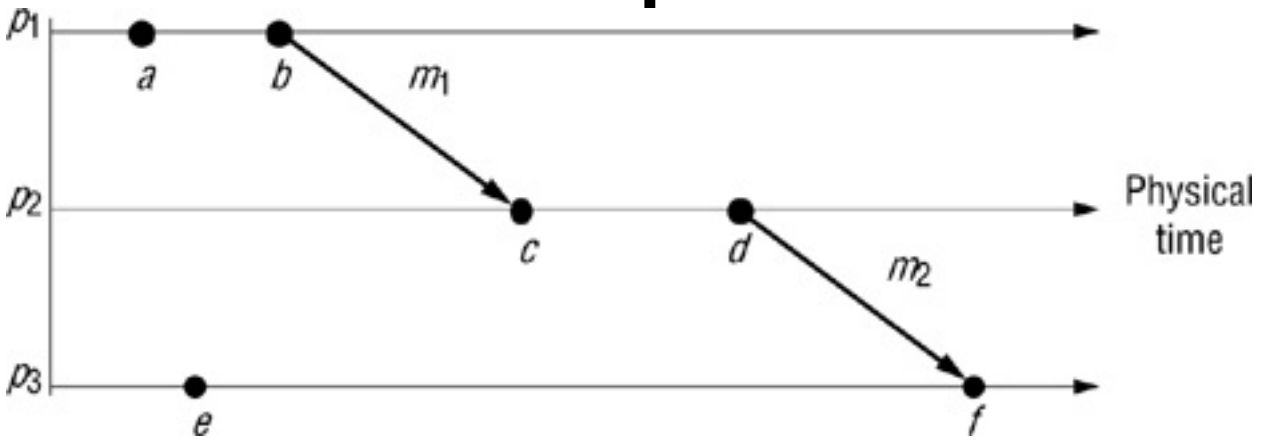
Credit: UW Lecture Notes

Lamport's Algorithm

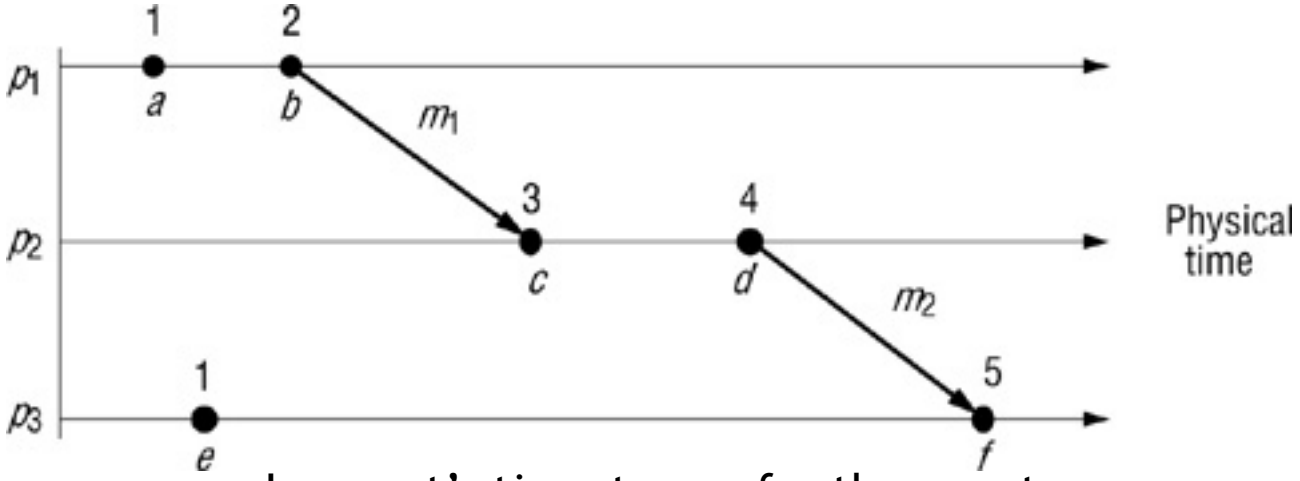
- Capturing happens-before relation
- Each process p_i has a local monotonically increasing counter, called its logical clock L_i
- Each event e that occurs at process p_i is assigned a Lamport timestamp $L_i(e)$
- Rules:
  - L_i is incremented before event e is issued at p_i such that $L_i := L_i + 1$
  - When p_i sends message m, it adds $t = L_i$: (m, t) [this is event send(m)]
  - On receiving (m, t), p_j computes $L_j := \max(L_j, t)$;

Credit: UW Lecture Notes

Example 1

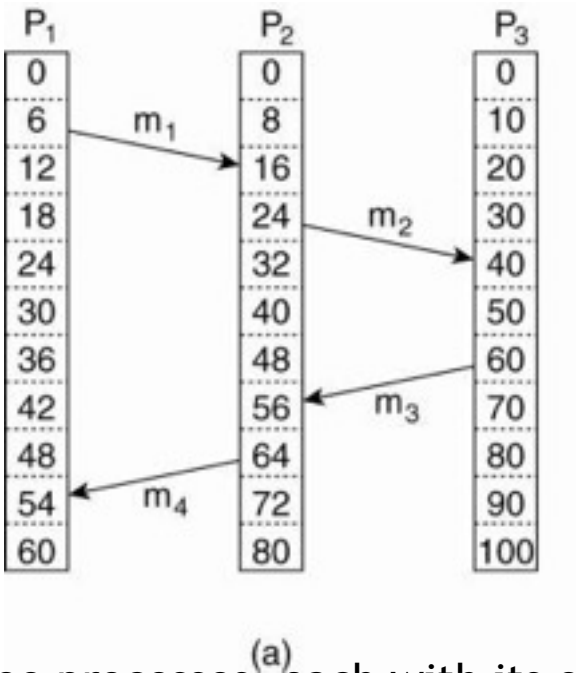


Events occurring at three processes

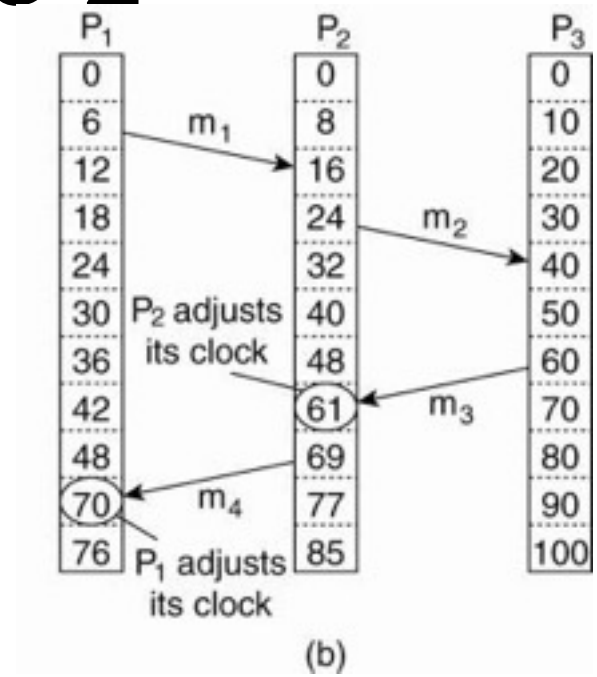


Lamport's timestamps for the events

Example 2



Three processes, each with its own clock. Clocks run at different rates

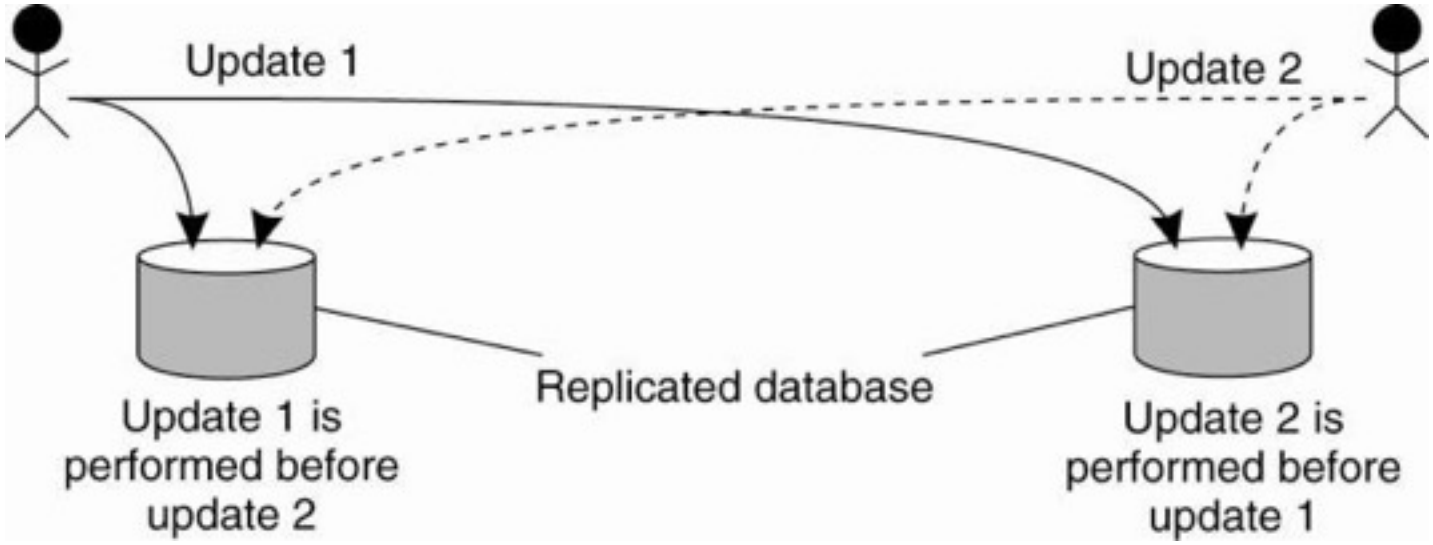


Lamport's algorithm corrects the clocks

- Rules:
  - Between every two events, the clock must tick at least once
  - No two events occur at exactly the same time. If two events happen in processes 1 and 2, both with time 40, the former becomes 40.1 and the latter becomes 40.2

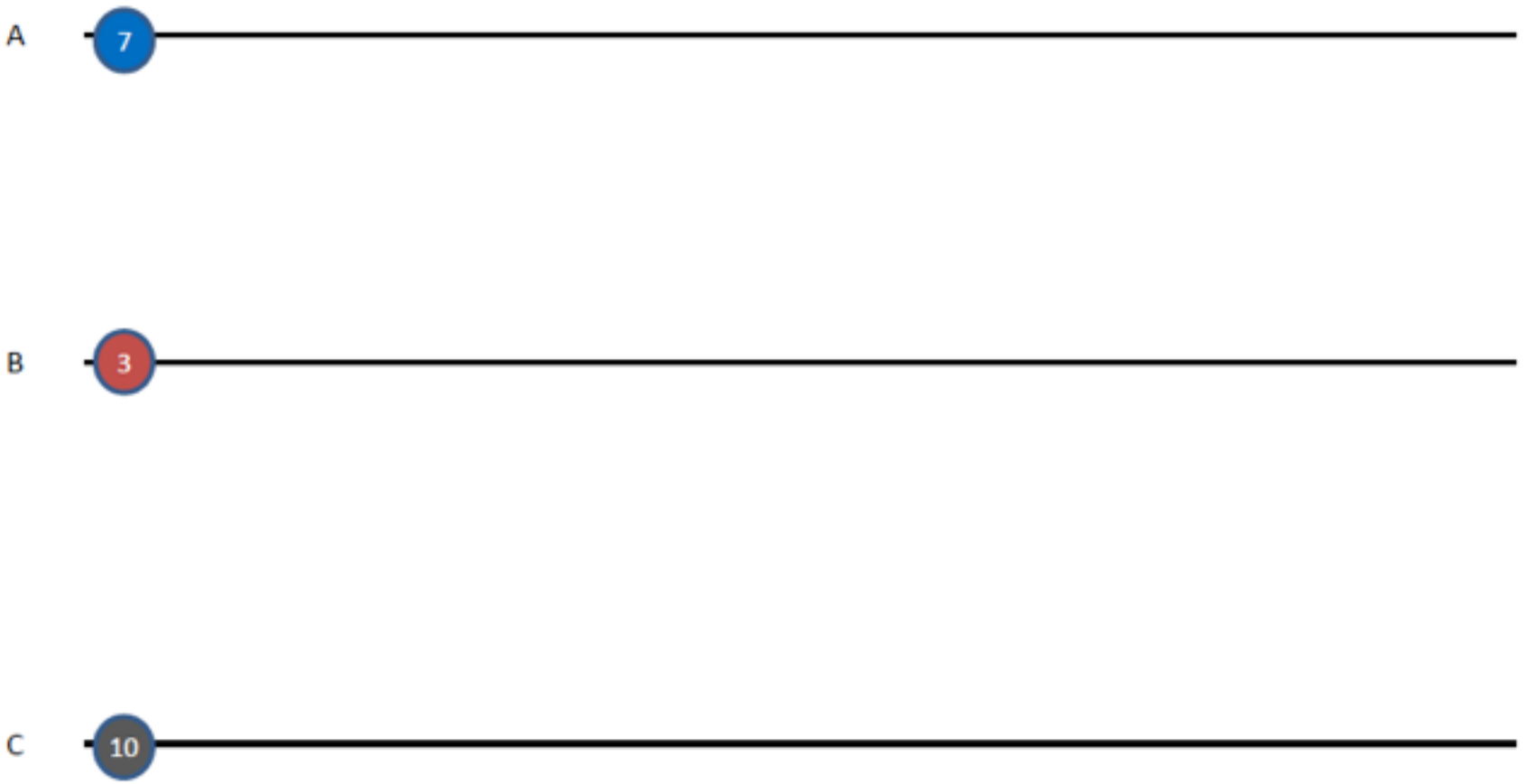
Totally Ordered Multicast

- A database has been replicated across several sites
- A query is always forwarded to the nearest copy
- Updates must be carried out at each replica in the **same order (consistent)**
- Solution: Totally ordered multicast: a multicast operation by which all messages are delivered in the same order to each receiver

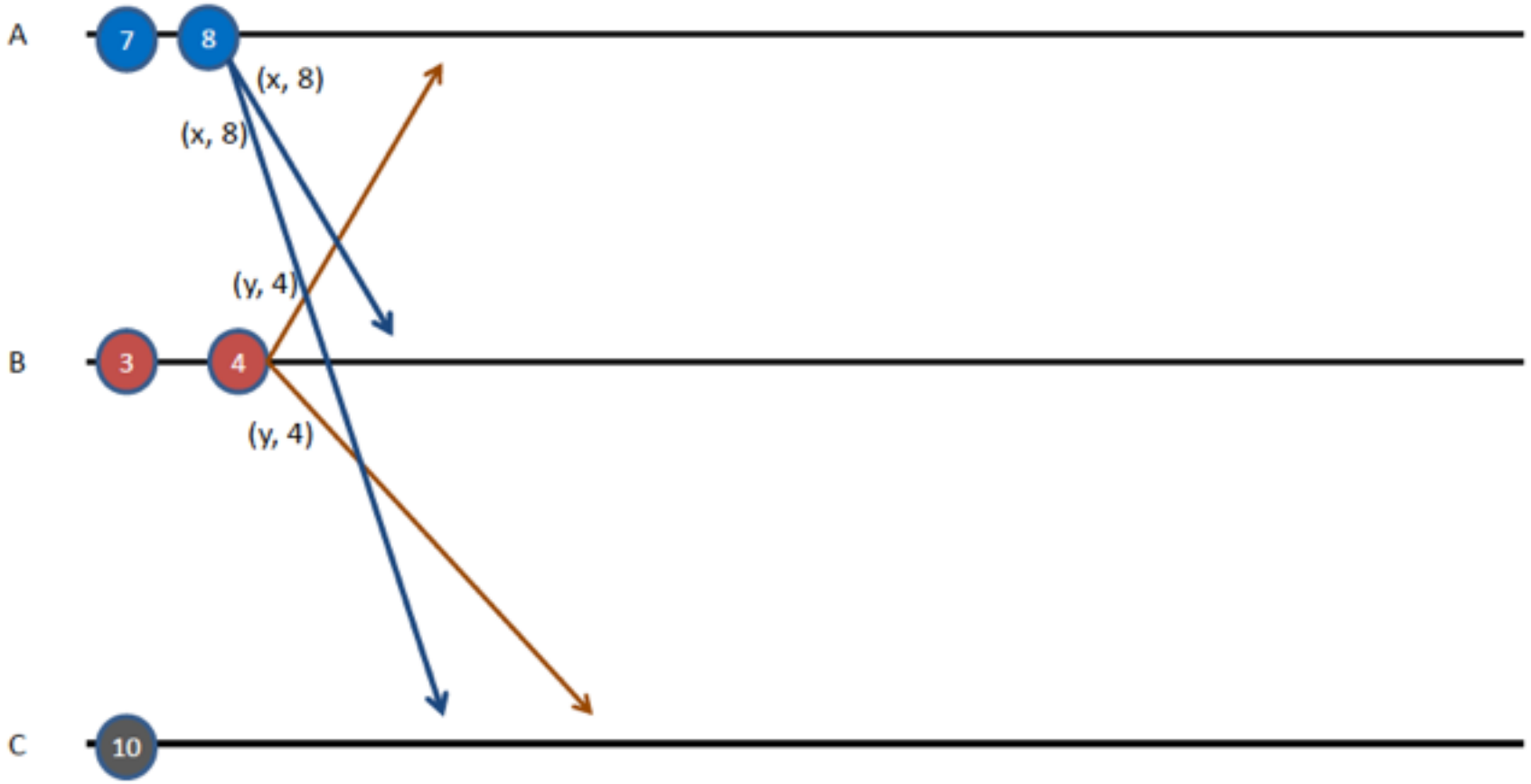


Lamport's Solution for Totally Ordered Multicast

- When a process receives a message, it is put into a local queue, ordered according to its timestamp
- The receiver multicasts an acknowledgment to the other processes
- We follow Lamport's algorithm for adjusting local clocks, the timestamp of the received message is lower than the timestamp of the acknowledgment
- All processes will eventually have the same copy of the local queue (provided no messages are removed)



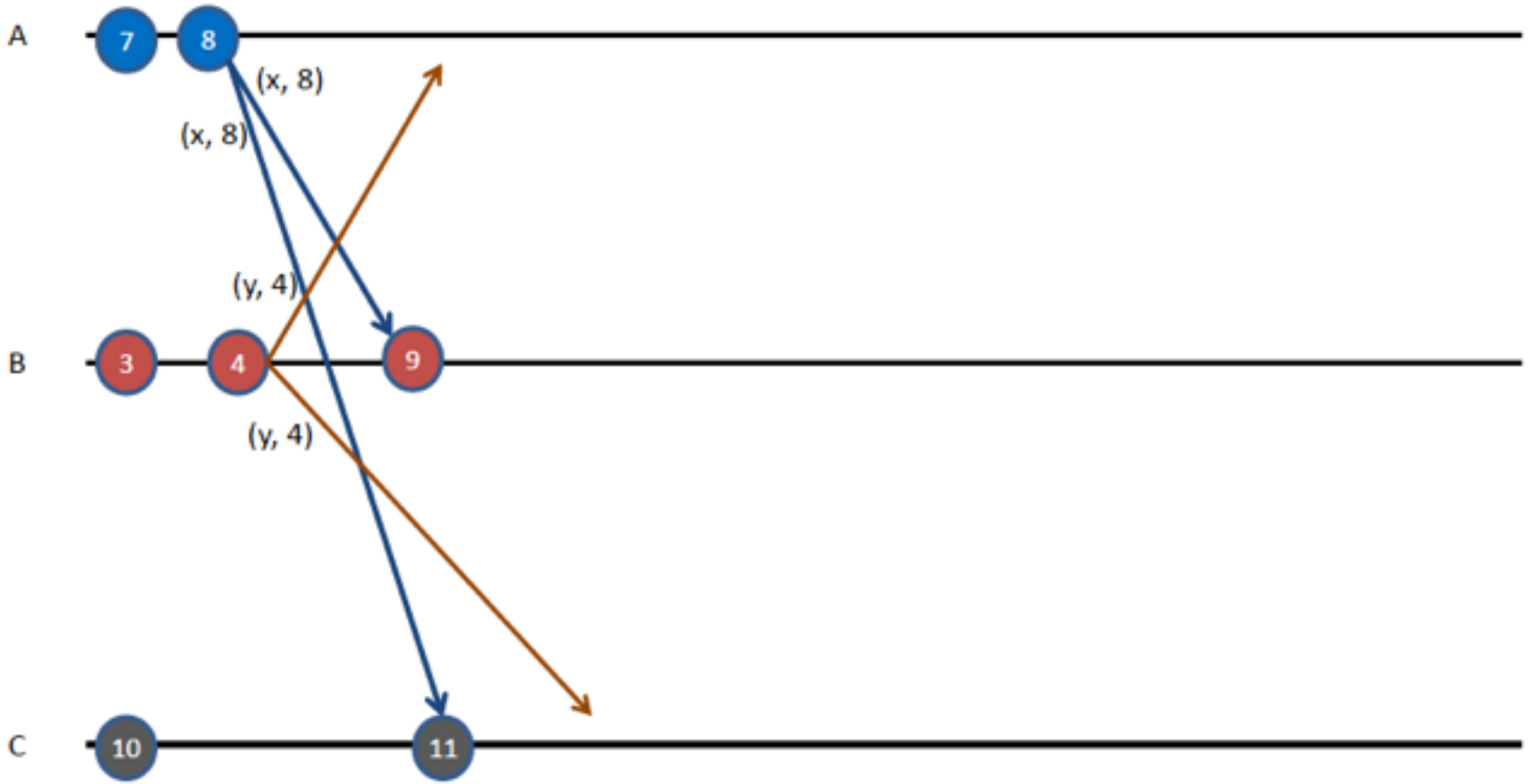
Credit: UW Lecture Notes



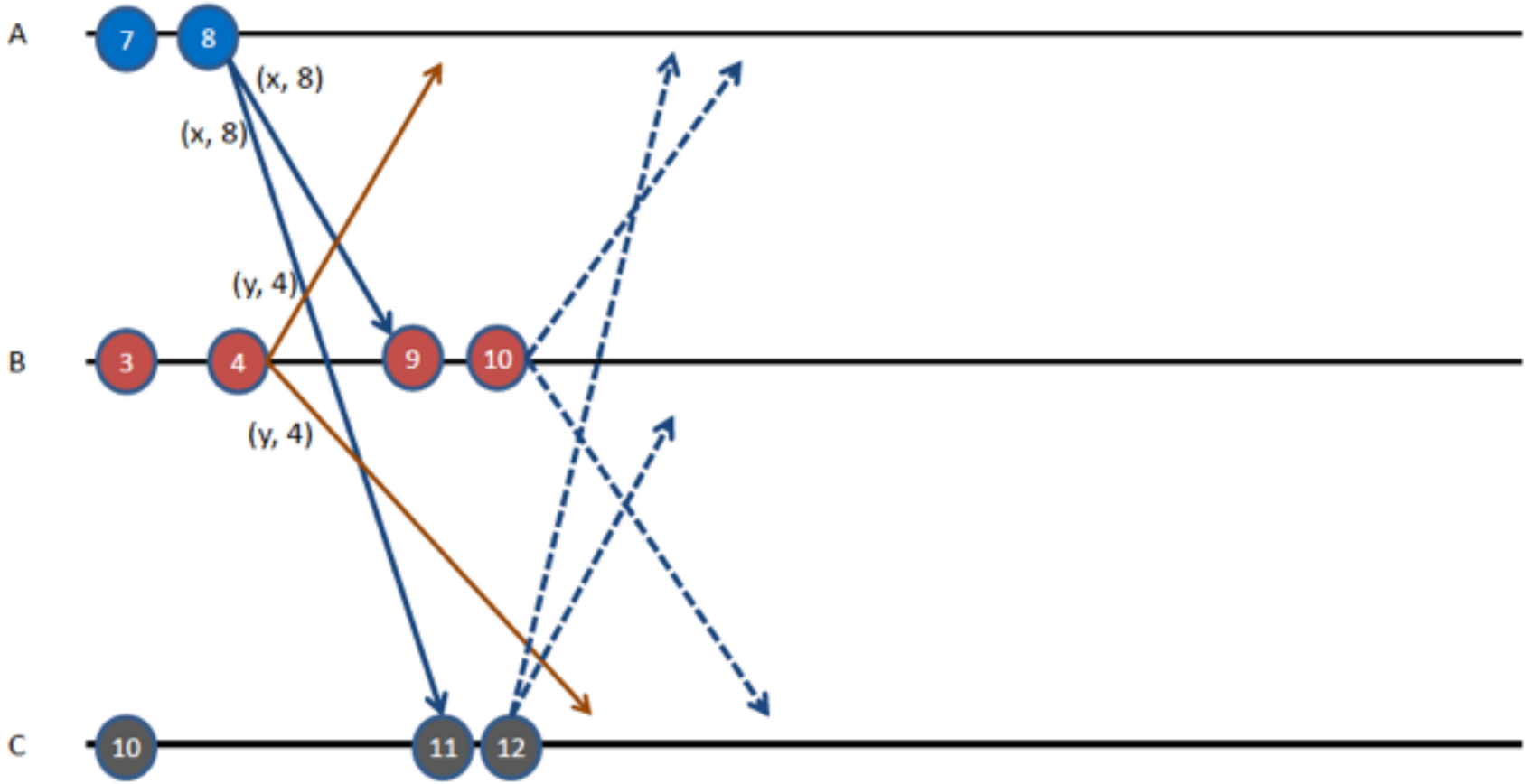
2 criteria before a node can commit a write:

- All write requests with a smaller timestamp has been committed.
- It has received a write-request acknowledgement from every node.

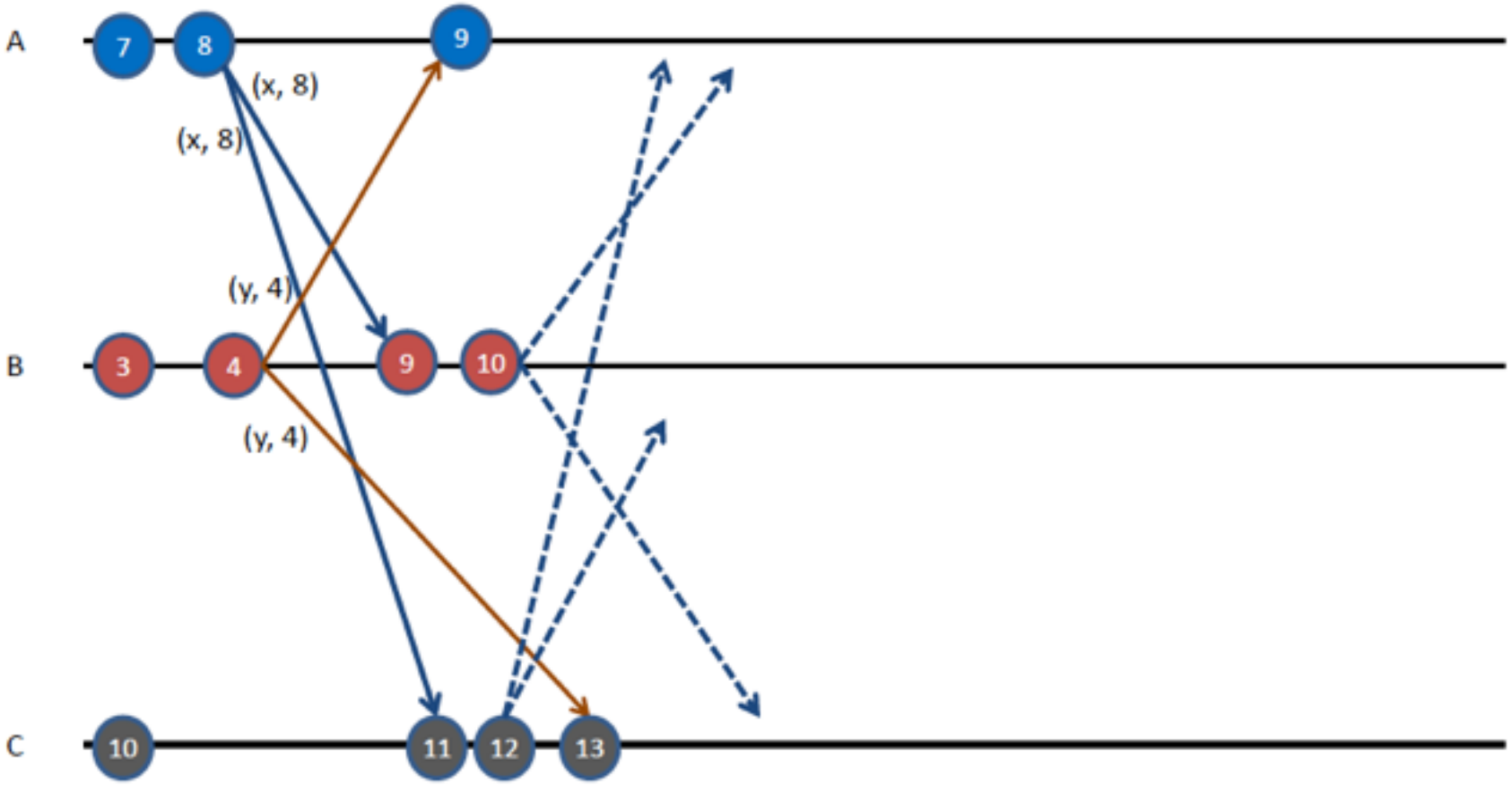
Credit: UW Lecture Notes



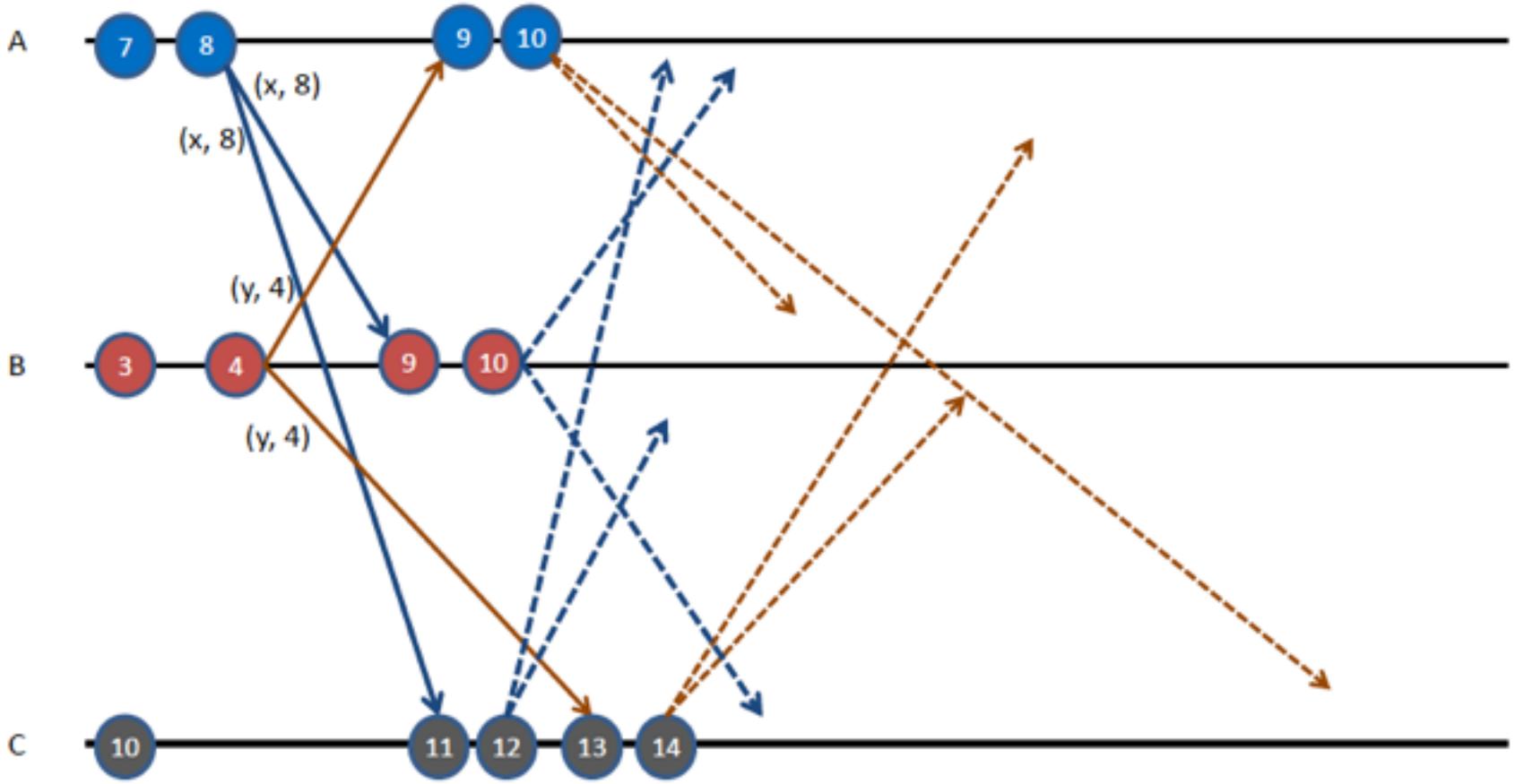
Credit: UW Lecture Notes



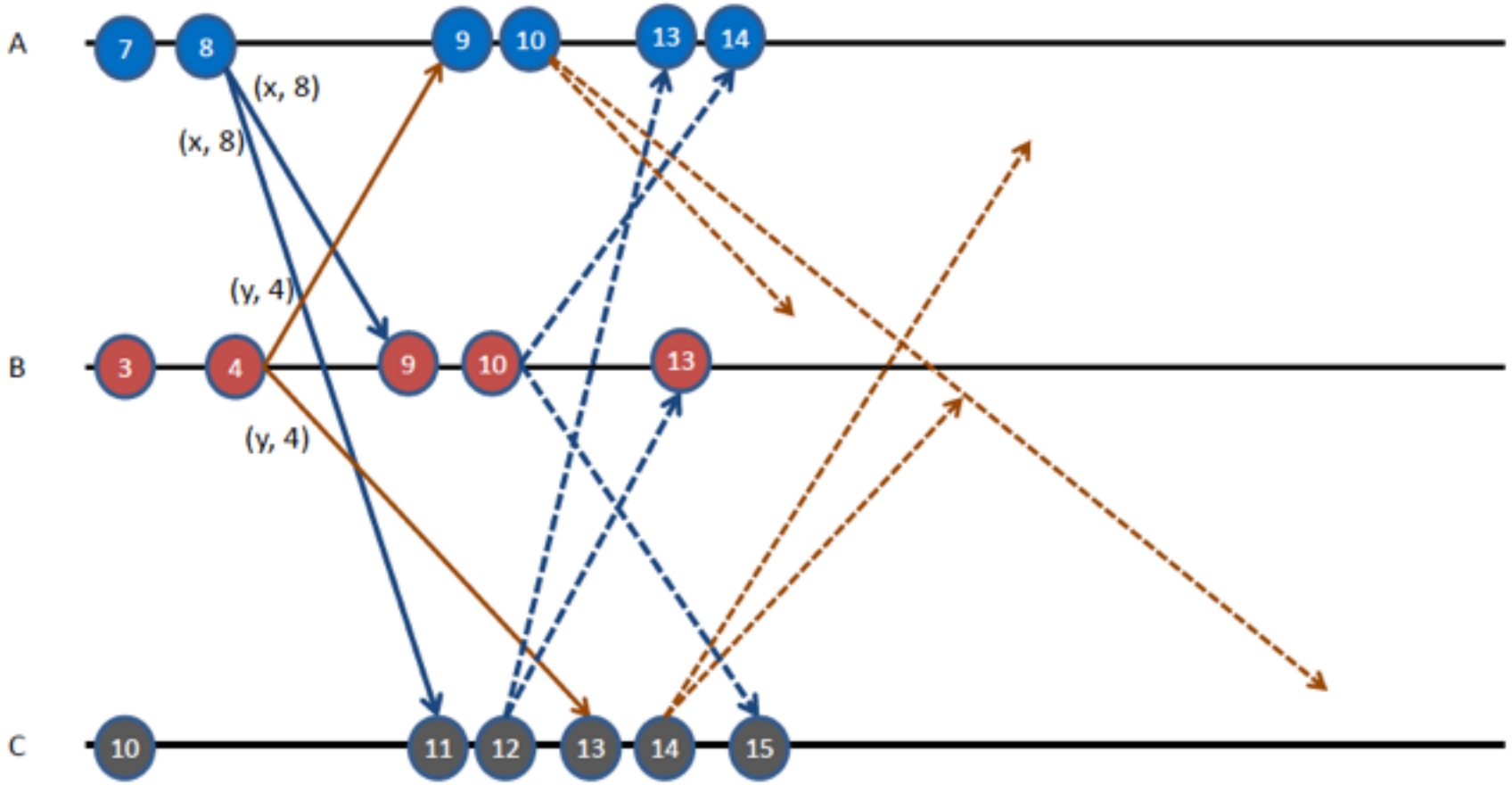
Credit: UW Lecture Notes



Credit: UW Lecture Notes

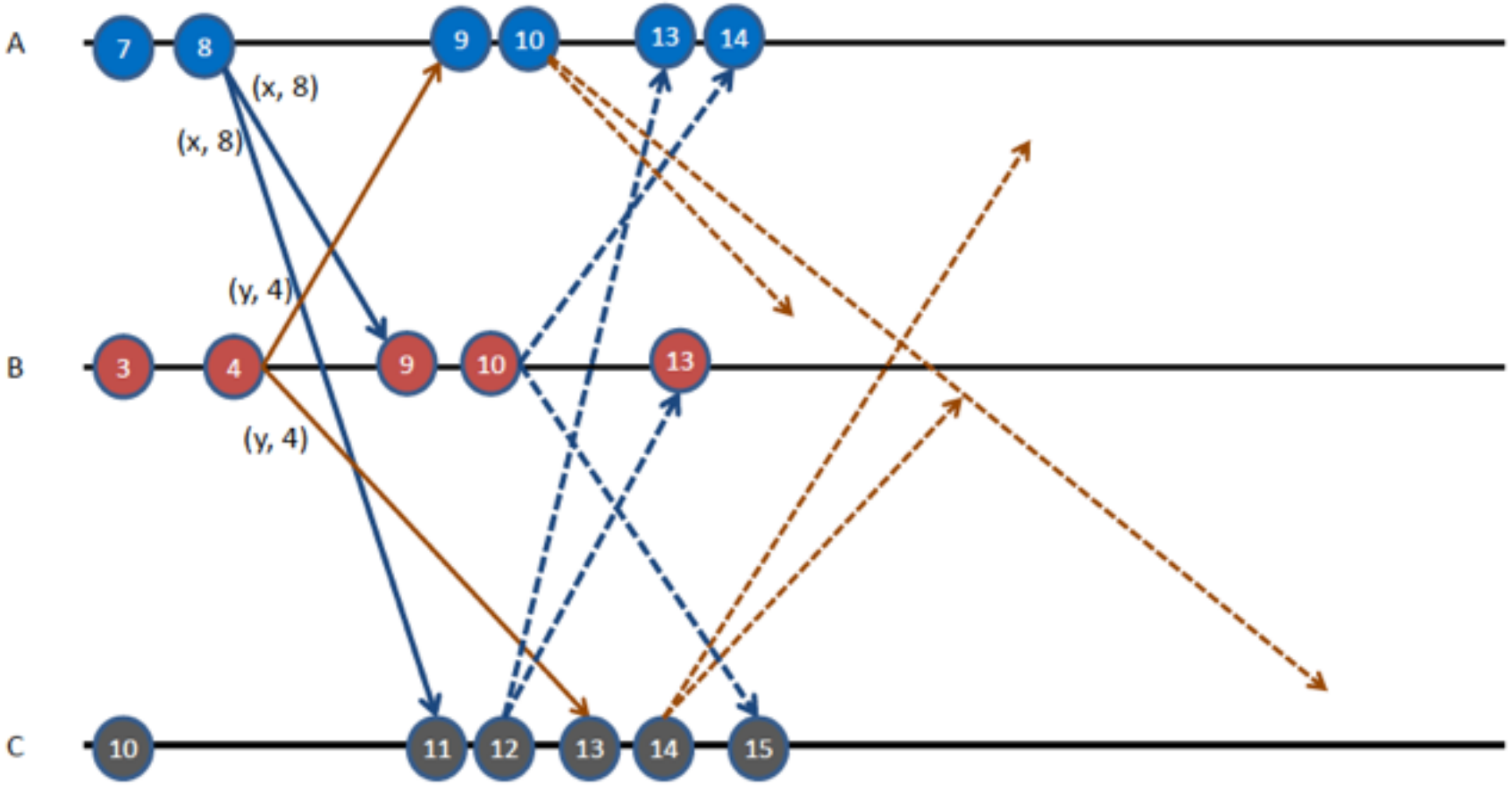


Credit: UW Lecture Notes



Credit: UW Lecture Notes

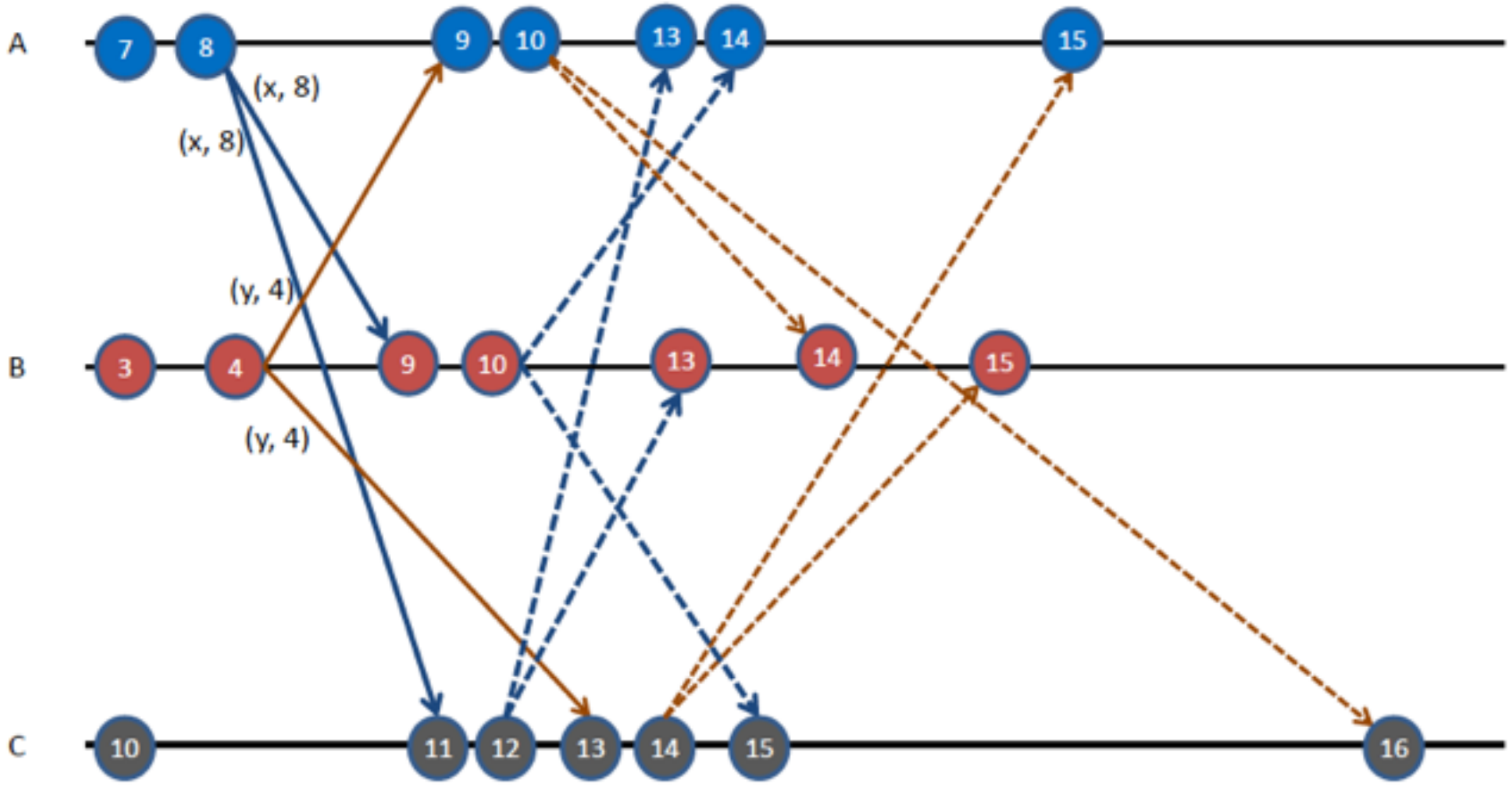
Can not commit (x, 8) until (y, 4) is committed as the nodes have already seen the (y, 4) message.



Node *B* will always send the (y, 4) message before sending an acknowledgement for (x, 8).

Credit: UW Lecture Notes

Can now commit (y, 4) and (x, 8) in that order.



Credit: UW Lecture Notes

Vector Clocks

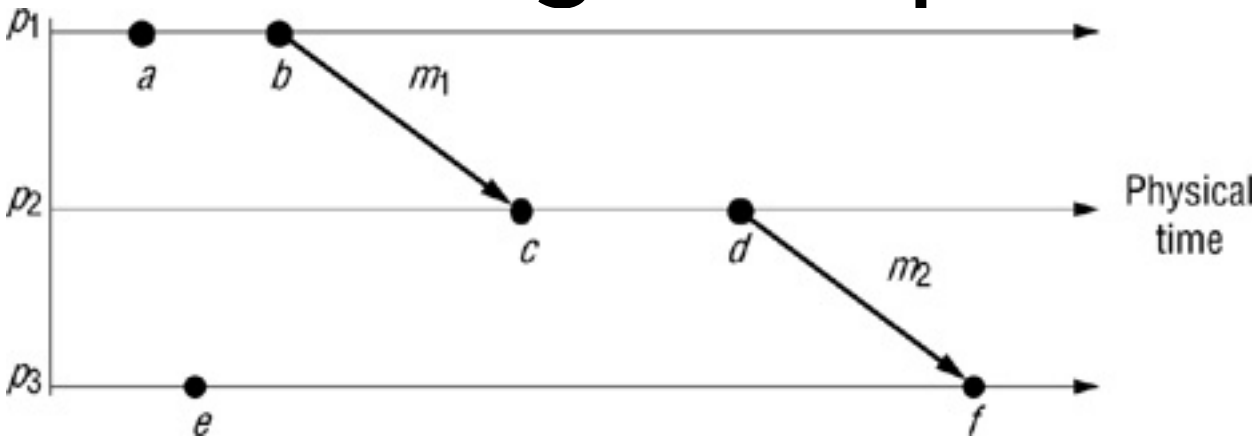
- Lamport's clock guarantees:
  - If a —> b, then L(a) < L(b)
- Shortcoming of Lamport's clock:
  - L(a) < L(b) does not imply a —> b
  - Does not capture causality
- Solution: each process P_i maintains a vector VC_i
  - VC_i [i] is the number of events that have occurred so far at P_i (local logical clock)
  - If VC_i [j] = k then P_i knows that k events have occurred at P_j (P_i's knowledge of the local time at P_j)

Rules for Updating Vector Clocks

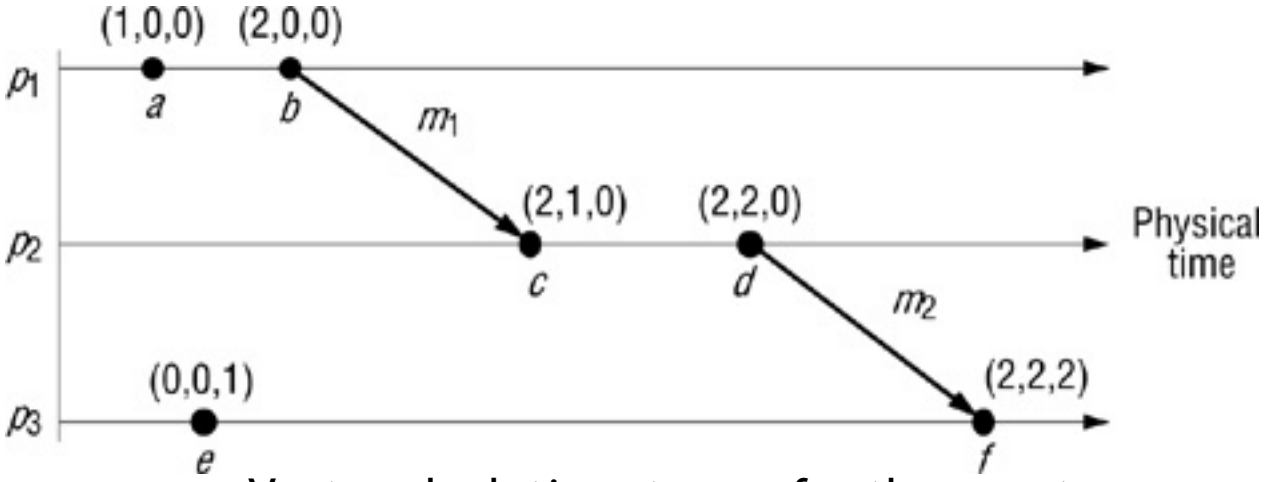
- $VC_i[j] = 0$ for all $i,j=1,2,\ldots N$
- Before P_i executes an event it updates its vector clock $VC_i[i] := VC_i[i]+1$
- When P_i sends a message to another process, it includes its entire vector clock VC_i as t
- When P_i receives a message with timestamp t, it updates its vector clock:

  $VC_i[j] := \max(VC_i[j],t[j])$, for $j = 1,2....,N$

Revisiting Example 1



Events occurring at three processes



Vector clock timestamps for the events

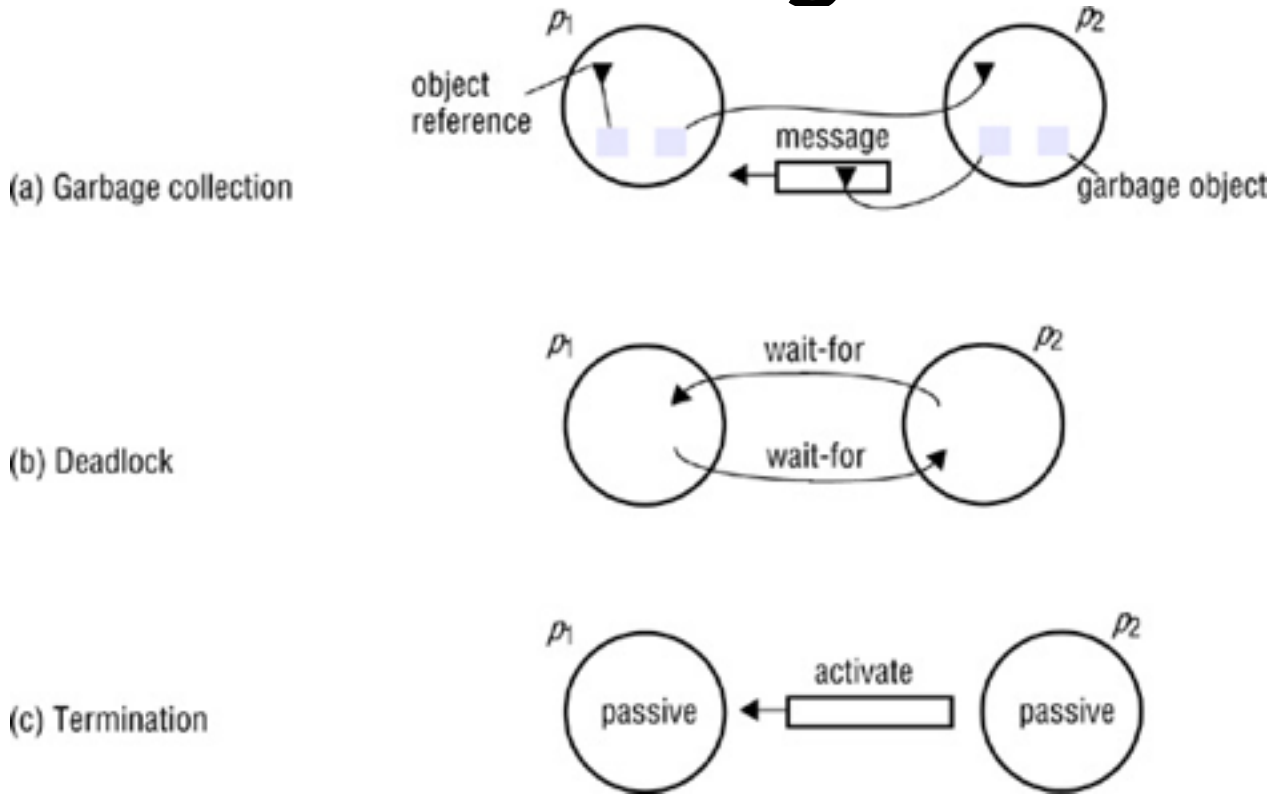
Comparing Vector Timestamps

- Equal time stamps (V = V‘)
  - iff V[j] = V‘[j] for j = 1,2....,N
- V ≤ V‘
  - iff V[j] ≤ V‘[j] for j = 1,2....,N
- V < V‘
  - iff V ≤ V‘ and V ≠ V‘
- Otherwise, V and V‘ are concurrent

Outline

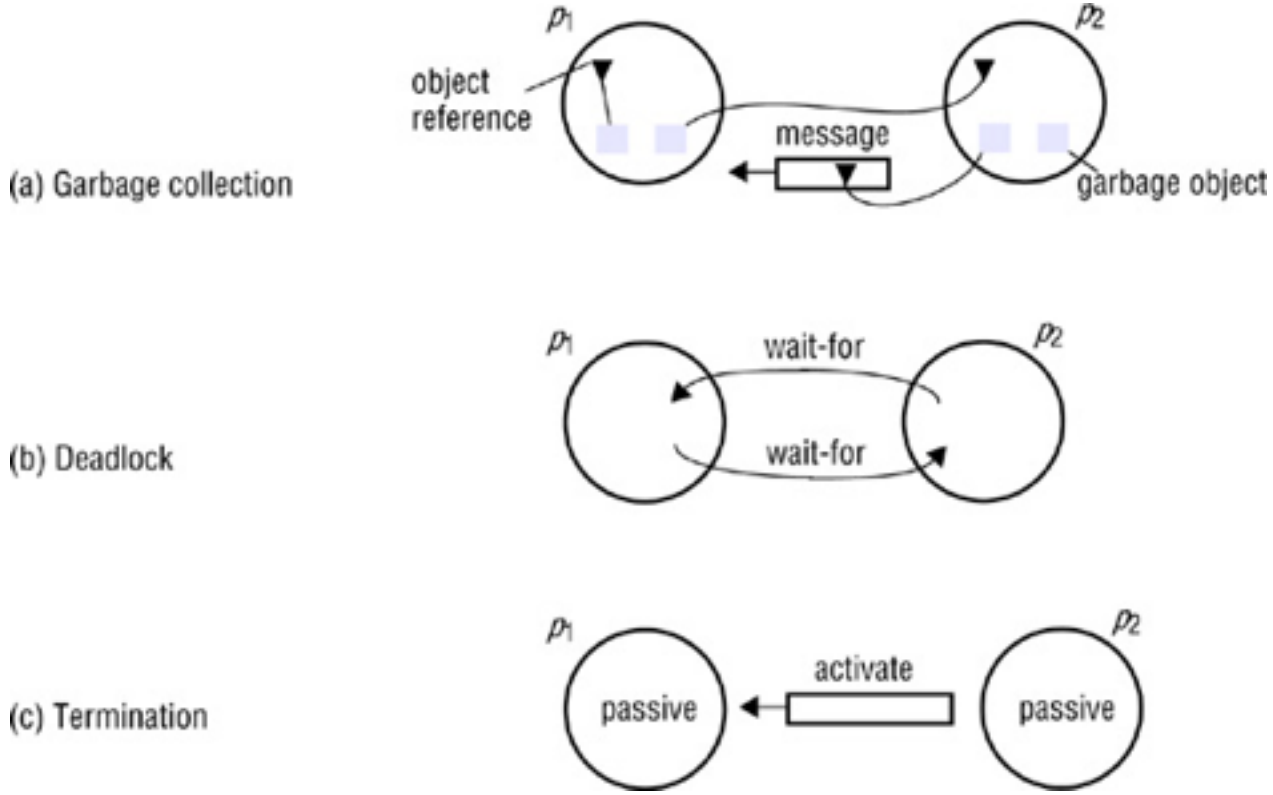
- Clocks
- Logical Clocks
- **Global State**
- Mutual Exclusion

Distributed Garbage Collection



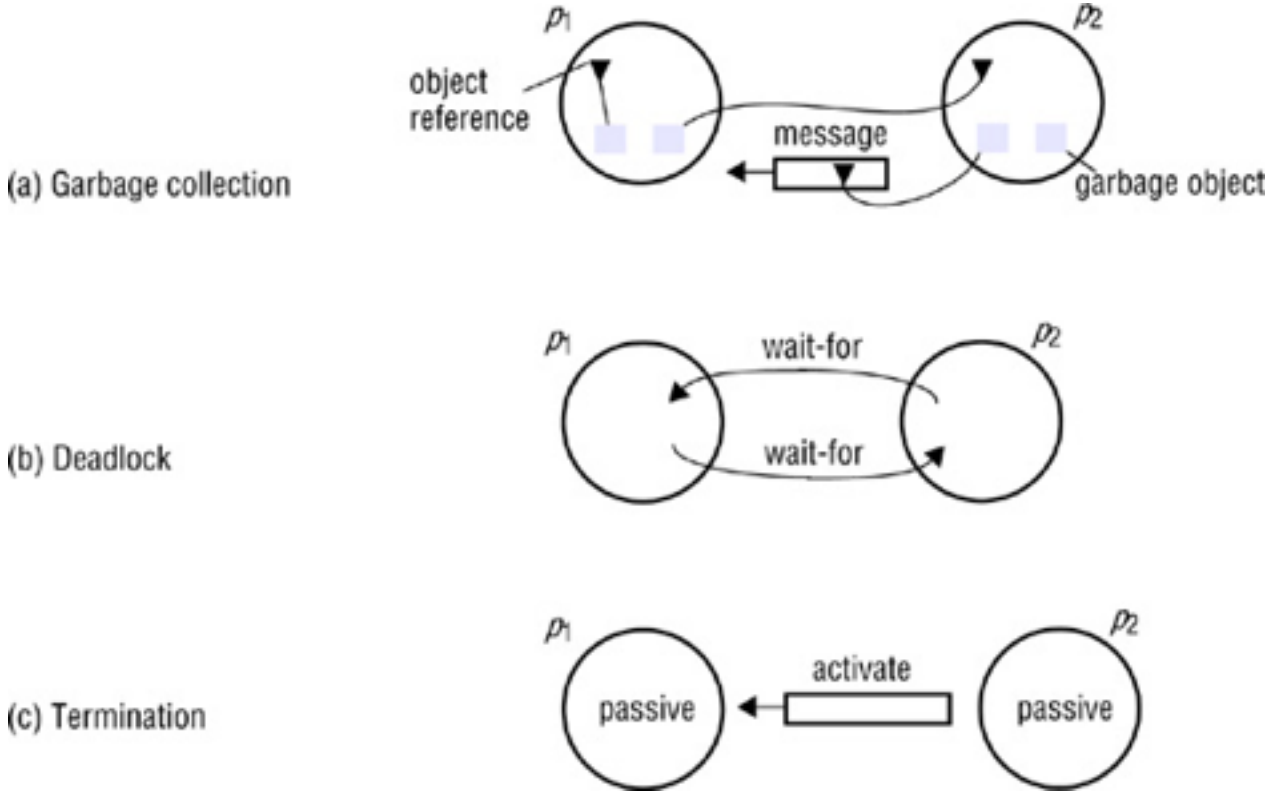
- An object is considered to be garbage if there are no longer any references to it in the distributed system
- Example: note object in p2, we must include the state of communication channels

Distributed Deadlock Detection



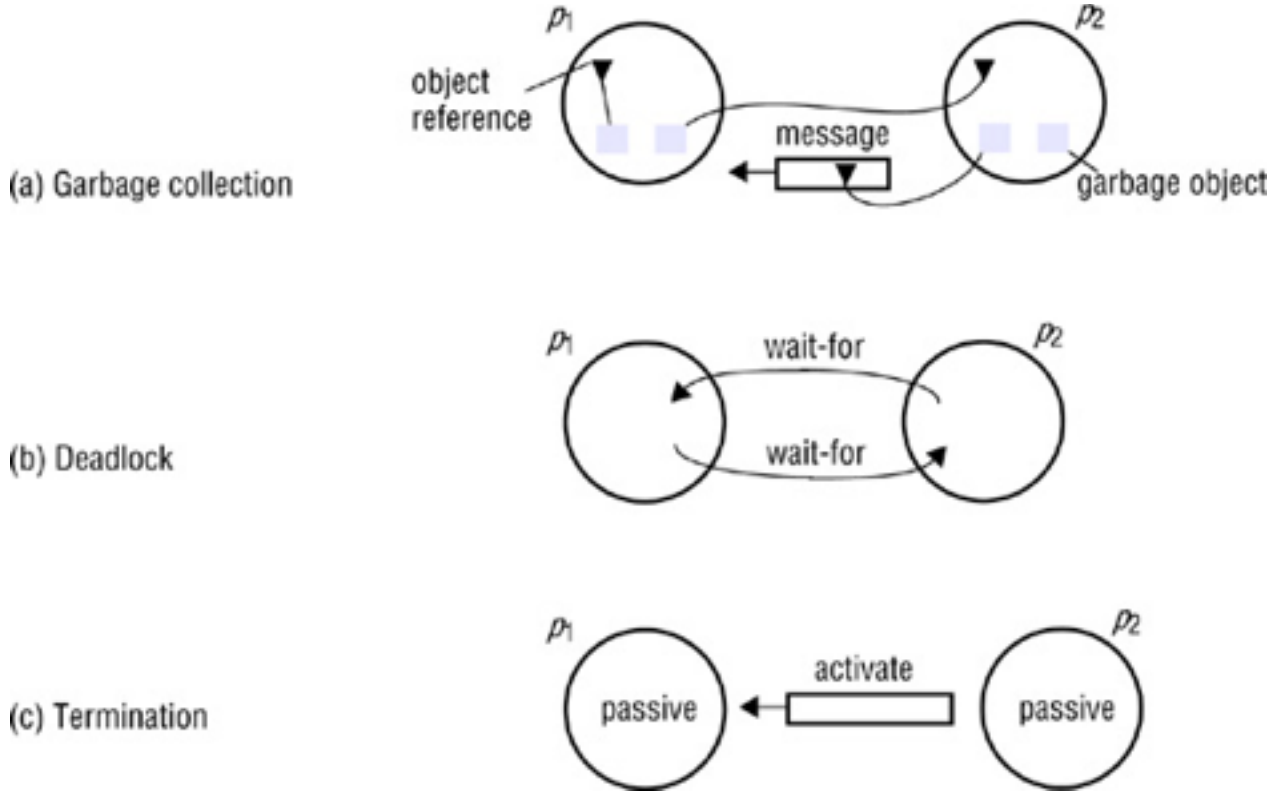
- Occurs when each of a collection of processes waits for another process to send it a message, and
- a cycle in the graph of this 'waits-for' relationship

Distributed Termination Detection



- Test whether each process has halted
- A passive process is not engaged in any activity of its own but is prepared to respond with a value requested by the other ==> becomes active again

Global State



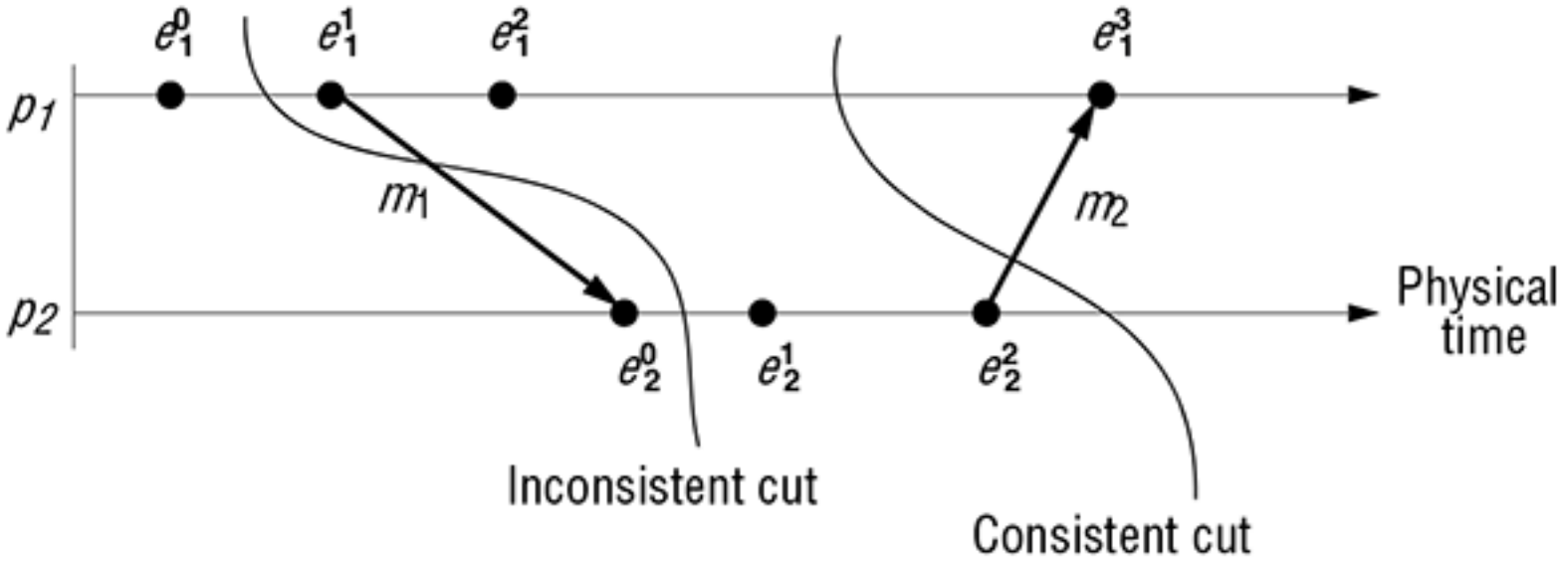
- Global state =

  local states of each process + messages in transit

Consistent Cuts

- $history\ (P_i) = h_i = <e_i^0, e_i^1, e_i^2, \ldots >$
- A finite prefix: $h_i^k = < e_i^0, e_i^1, \ldots e_i^k >$
- Global history: H = $h_0 \cup h_1 \cup \ldots \cup h_{N-1}$
- A cut of the system's execution is a subset of its global history that is a union of prefixes of process histories: $C = h_1^{C1} \cup h_2^{c2} \cup \ldots \cup h_N^{CN}$
- A cut C is consistent if, for each event it contains, it also contains all the events that happened-before that event:
  - For all events $e \in C$, $f \rightarrow e \Rightarrow f \in C$

Example of Consistent Cut



‘snapshot’ Algorithm of Chandy and Lamport

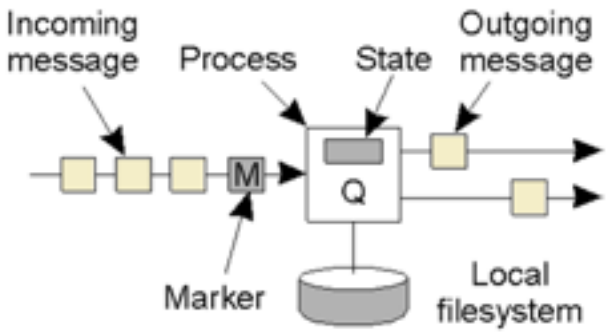
- Goal: record a set of process and channel states (a ‘snapshot’) such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent
- Assumptions:
  - Neither channels nor processes fail
  - Channels are unidirectional and provide FIFO-ordered message delivery
  - The graph of processes and channels is strongly connected
  - Processes continue operation while snapshot

Chandy and Lamport's Algorithm

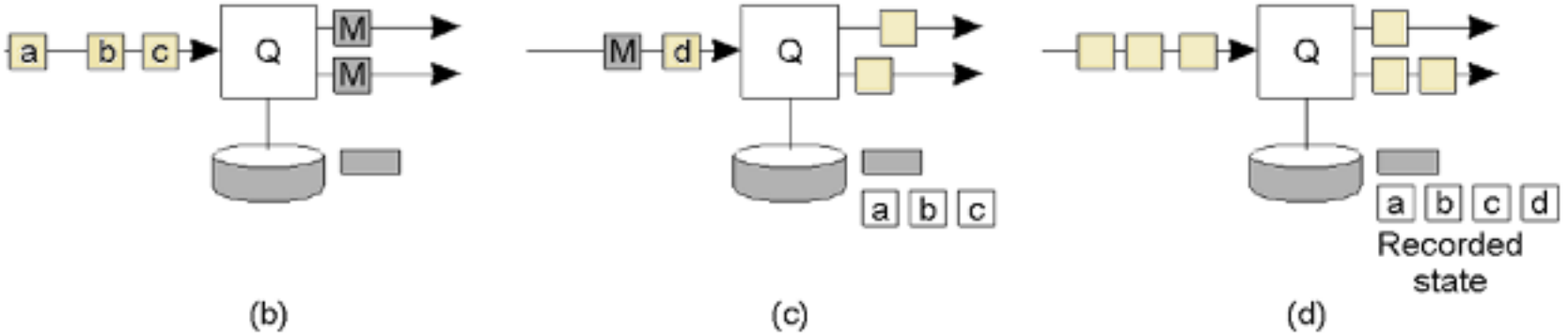
- Sender (Process p)
  - Record the state of (p)
  - For each outgoing channel (c) incident to (p), send a marker before sending ANY other messages
- Receiver (Process q receives marker on channel c1)
  - If (q) has not yet recorded its state
    - Record the state of (q)
    - Record the state of (c1) as null
    - For each outgoing channel (c) incident to (q), send a marker before sending ANY other messages
  - If (q) has already recorded its state
    - Record the state of (c1) as all messages received since the last time the state of (q) was recorded

Credit: VT Lecture Notes

Example



(a) Organization of a process and channels for any distributed snapshot



(b) Process Q receives a marker with regard to the first time as well as records it's local state

(c) Q records all incoming information

(d) Q receives a marker for its incoming channel as well as finishes recording the state from the incoming channel

Outline

- Clocks
- Logical Clocks
- Global State
- **Mutual Exclusion**

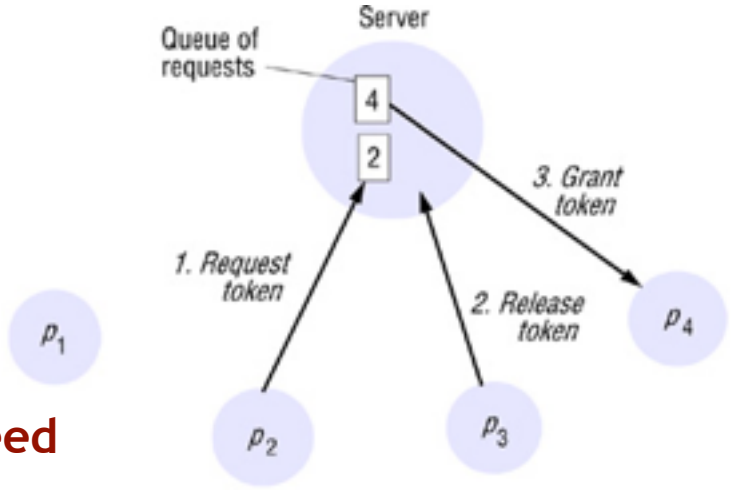
Distributed Mutual Exclusion

- In a distributed system with shared resources, mutual exclusion is required to prevent interference and ensure consistency when accessing the resources
- Example:
  - Multiple processes (distributed) updating a file such as the case of NFS files (server stateless, *locked* is provided by Unix to handle requests from clients)
  - Ethernets and IEEE 802.11 wireless networks: only one node transmits at a time on the shared medium

Requirements for Mutual Exclusion

- ME1 (Safety): At most one process may execute in the critical section (CS) at a time
- ME2 (Liveness): Requests to enter and exit the critical section eventually succeed
- ME3 (—> ordering): If one request to enter the CS happens-before another, then entry to the CS is granted in that order

Central Server Algorithm



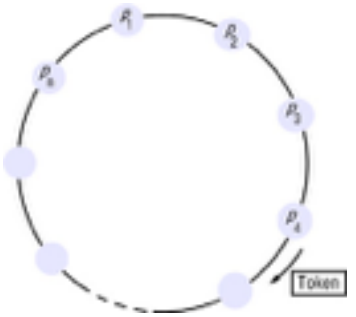
ME1 and ME2: guaranteed
ME3: not guaranteed

ME3: ($\rightarrow$ ordering) If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

- A server grants permission to enter the critical section
  - Server maintains queue of requests
  - To enter a critical section, a process sends a request message to server and waits for permission
  - After finishing, a process send a release lock to server

Ring-Based Algorithm

ME1 and ME2: guaranteed
ME3: not guaranteed

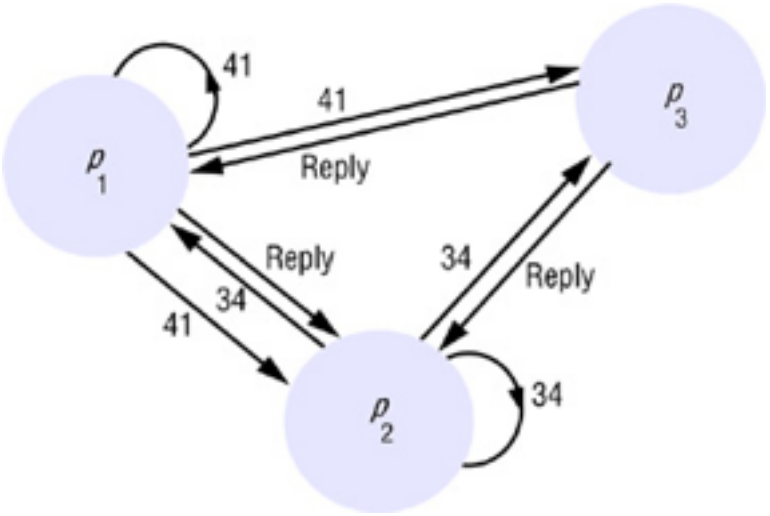


- Arrange processes in a logical ring
- Requires only that each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$
- If a process does not require to enter the critical section when it receives the token —> forward the token to its next neighbour
- A process requiring to enter critical section, wait until receive the token and retain it (1 - N messages)
- To exit the critical section, the process sends the token on to its neighbour

Multicast and Logical Clocks Ricart and Agrawala

- Implement mutual exclusion between N peer processes using multicast
- Idea:
  - Processes that require entry to a critical section multicast a request message, and
  - Can enter CS only when all the other processes have replied
- Number of messages to enter CS: 2(N-1)
- ME1, ME2, and ME3 are guaranteed

Example



- P_1 (t=41) and P_2 (t=34) request entry to CS, concurrently. P_3 does not want to enter CS
- P_3 replies to both P_1 and P_2 requests
- When P_2 receives P_1 request, its own request has the lower timestamp and so does not reply — However, P_1 replies immediately

Thank You