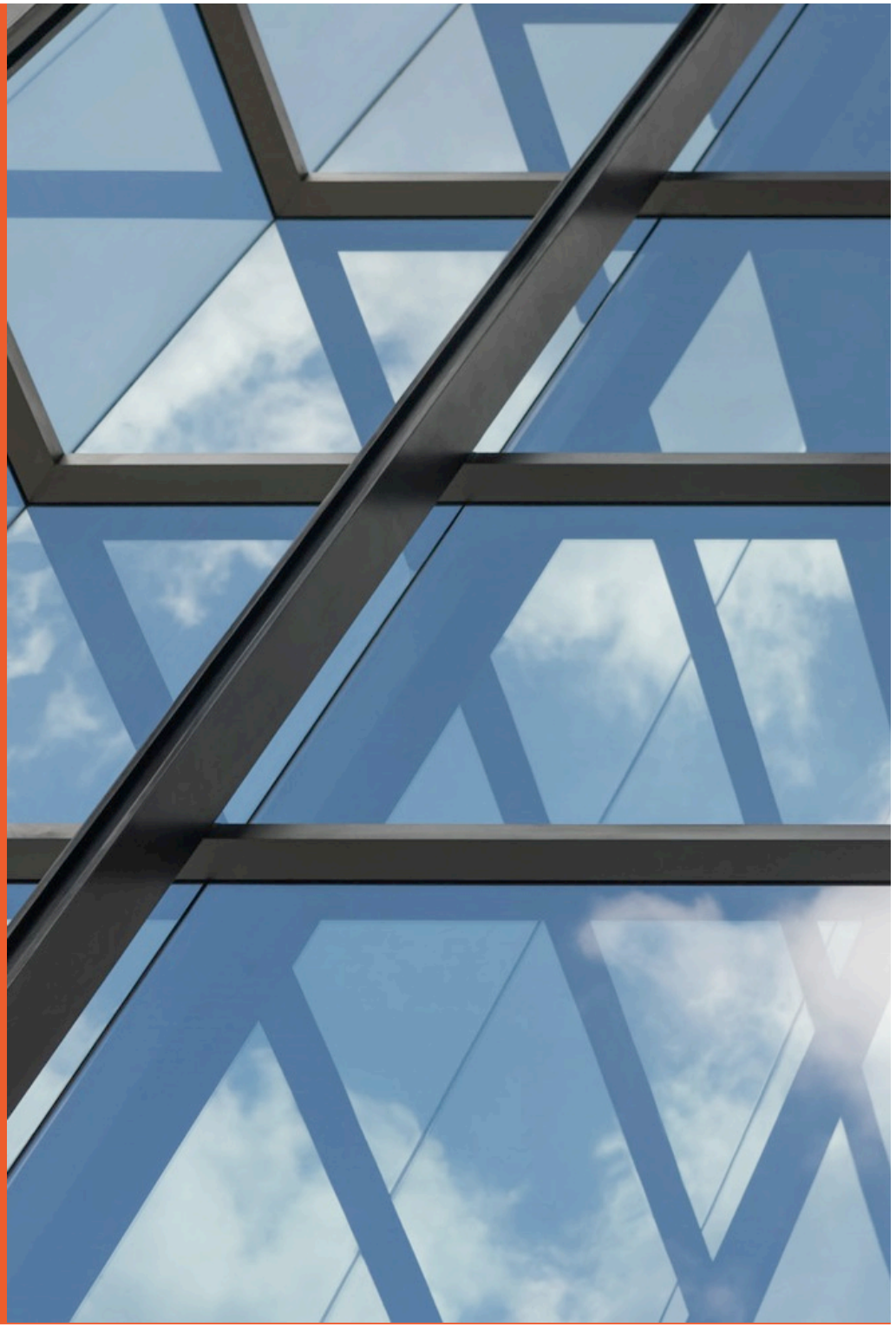
COMP3221: Distributed Systems

Synchronisation

Unit coordinator Dr. Vincent Gramoli
Lecturer Dr. Guillaume Jourjon
School of Information Technologies

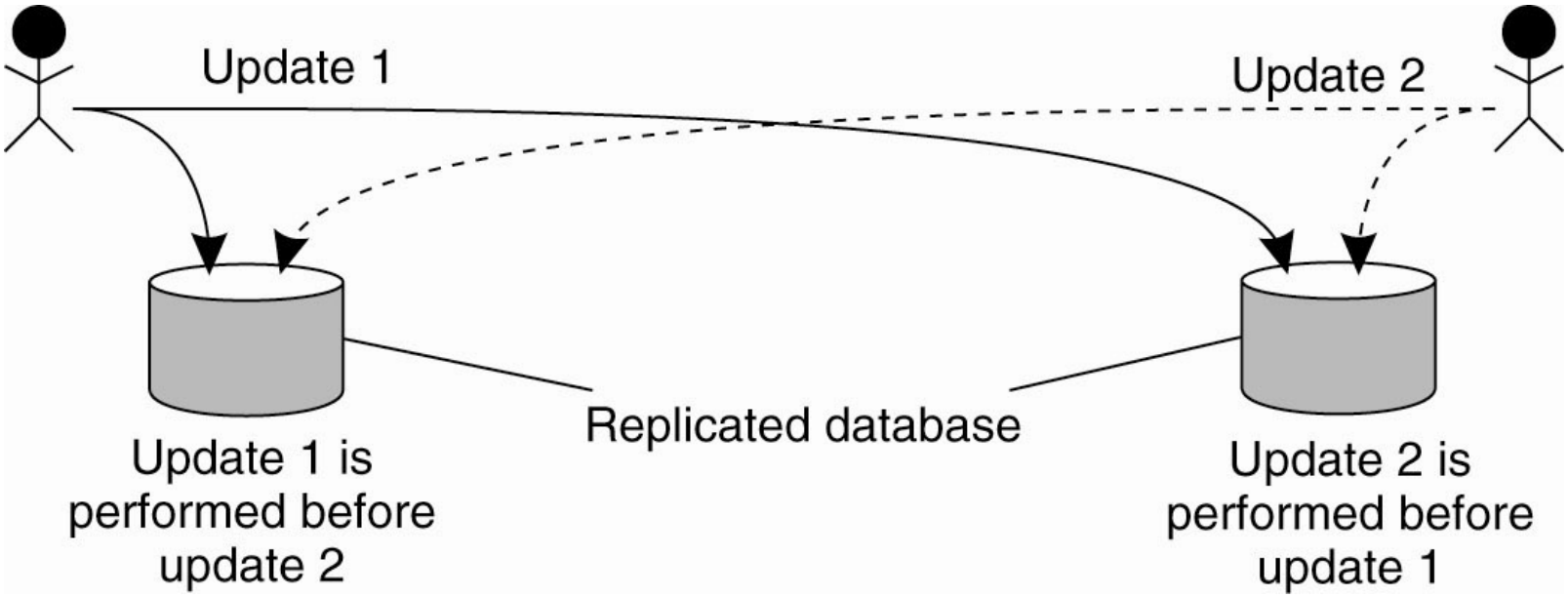


THE UNIVERSITY OF SYDNEY

Motivation

Why is time important in a distributed system?

- A bank replicates two copies of an account database in New York City (NYC) and San Francisco (SF) so that a query is always forwarded to the nearest city while updates must be carried out at both sites



- A client in SF adds $100 to his account which currently contains $1000 while the bank add 1% interest in NYC.
- Updating a replicated database and leaving it in an inconsistent state: $1111 in SF and $1110 in NYC
- The problem is that both sites perceive the updates in different order leading to inconsistencies

The solution is to use time to order requests

Outline

- Time
- Physical Time
- Logical Time
- Multicast
- Mutual exclusion
- Leader election

Time

THE UNIVERSITY OF SYDNEY

Time

The clock of a CPU

- Physical clock
  - Machine quartz crystal
  - Kept under tension: oscillates at a well-defined frequency
  - Each oscillation decrements a counter by 1
  - When the counter reaches 0, an interrupt is triggered and the counter is reset
  - Hence, we can obtain an interrupt 60 times per second, called a *clock tick*.
- Hardware clock
  - At each clock tick, one is added to the time stored in a battery-backed up CMOS RAM
  - When the CPU is turned off, the time in the CMOS RAM keep being incremented
- Software clock
  - It starts running at boot time and synchronize to the hardware clock
  - Used by the operating system to indicate a rough approximation of time

Time

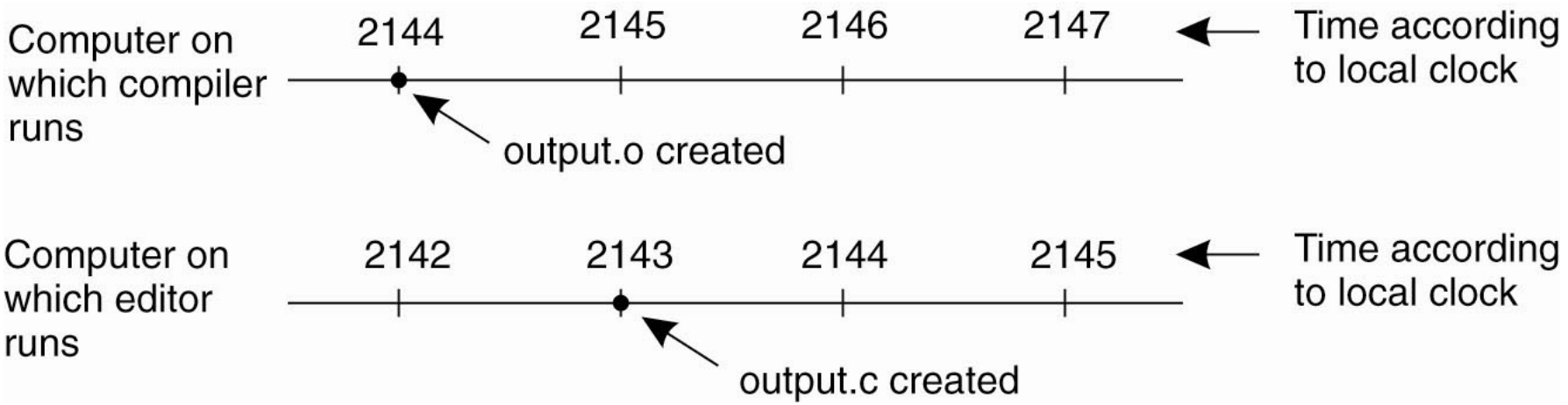
The notion of time in a distributed system

- Multiple CPUs imply multiple clocks
- The frequency of two crystals cannot be exactly the same
- Software clocks progressively get out of synch
- The difference returned by two clock values is the clock skew

Time

Example: Makefile

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



Time

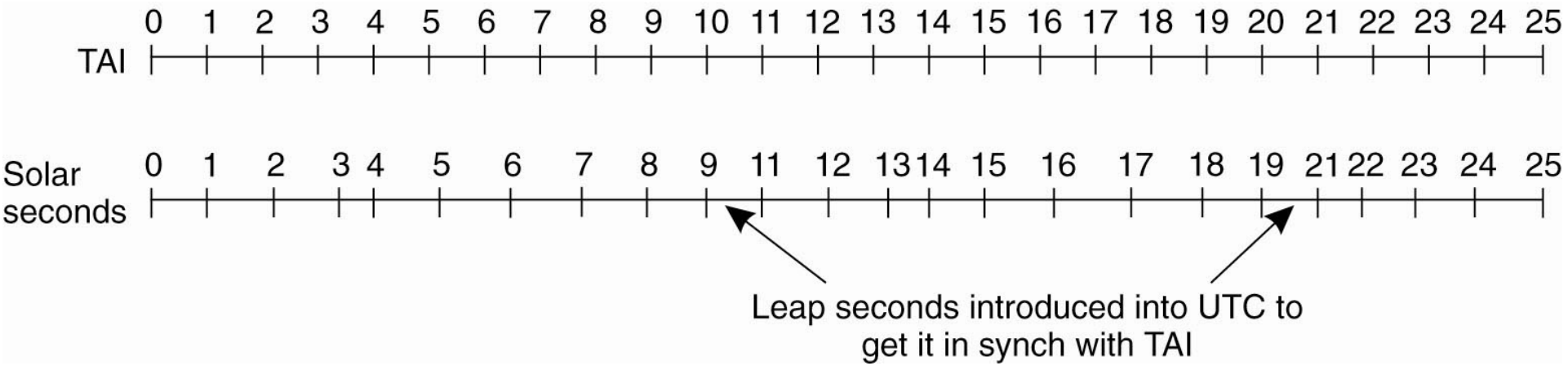
Leap second

- Astronomers defined the second to be mean solar second
  - Solar day becomes longer and longer
  - As solar second depends on the solar day, it also changes
- Physicist converted the solar second in 1948 into:
  - Time for the cesium 133 atom to make 9,192,631,770 transitions
  - This measure is still today the second used by the International Atomic Time (IAT)
- These two measures must be resynchronized by adding 1 sec to IAT sometimes
- This may introduce a bug in operating systems like in RedHat:
  - The timer interrupt handler holds xtime_lock
  - The lead second code does a printk to notify about the leap second
  - The printk code tries to wake up klogd (to prioritize kernel message)
  - The scheduler tries to get the current time if the system is busy,
  - Thus the scheduler tries to acquire xtime_lock

Time

Leap second (con't)

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



Time

Leap second (con't)

- This may introduce a bug in operating systems like in RedHat:
  - The timer interrupt handler holds xtime_lock
  - The lead second code does a printk to notify about the leap second
  - The printk code tries to wake up klogd (to prioritize kernel message)
  - The scheduler tries to get the current time if the system is busy,
  - Thus the scheduler tries to acquire xtime_lock

  $\Rightarrow$ Deadlock

  More details: https://lkml.org/lkml/2009/1/2/373

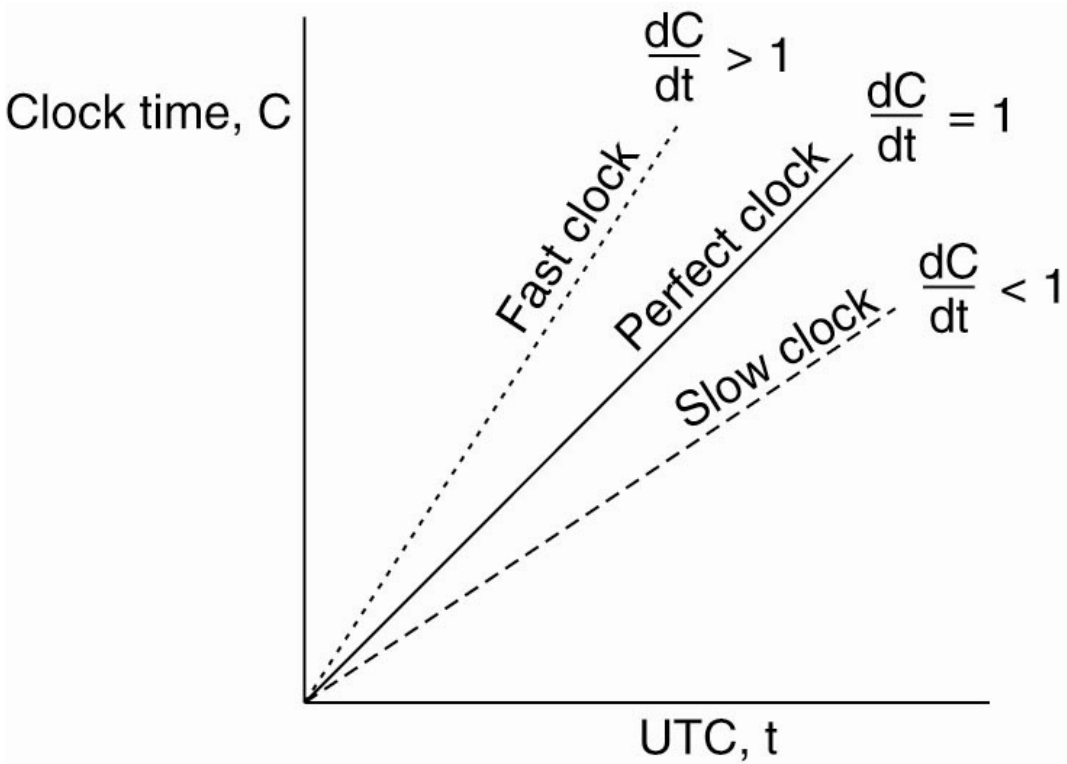
Physical Time

THE UNIVERSITY OF SYDNEY

Physical time

Different clock rate

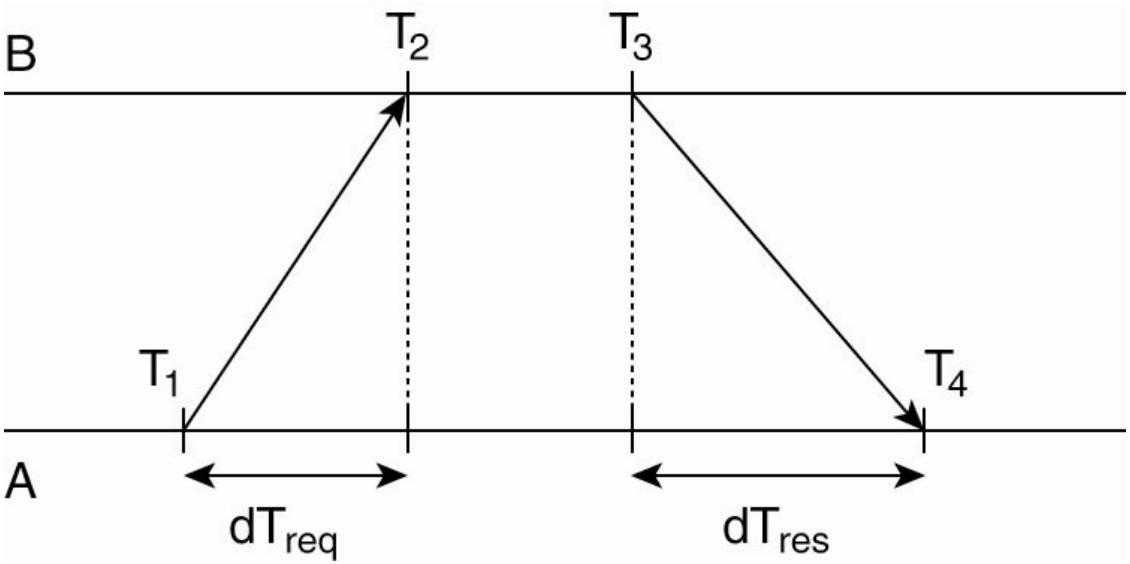
- The relation between clock time and UTC when clocks tick at different rates
- The *skew* of the clock is dC/dt-1



Physical time

Cristian's algorithm [1989]

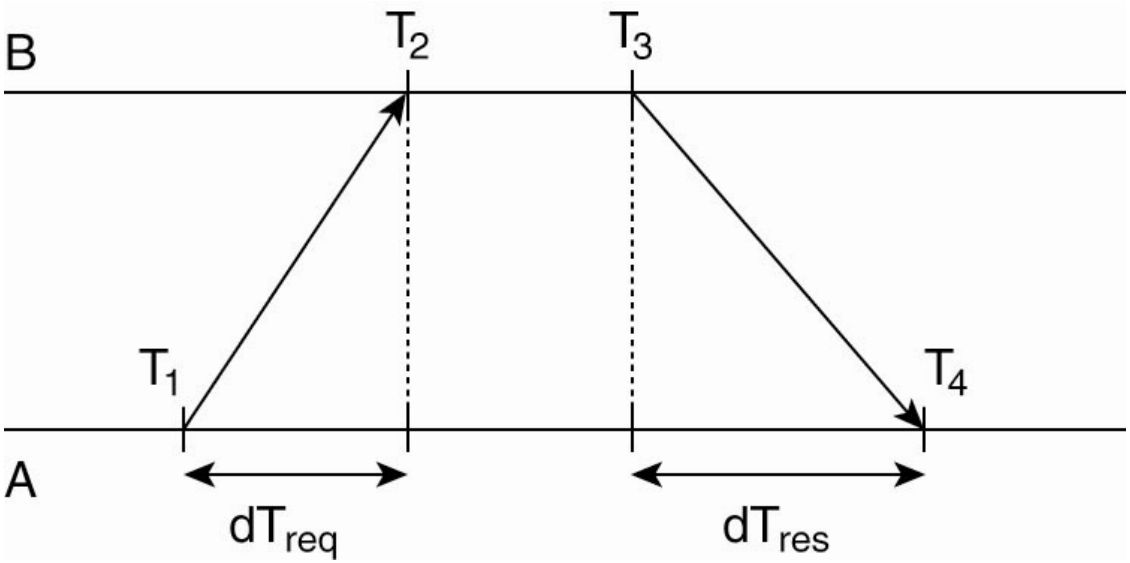
1. A sends a message to B with its timestamp T1 set to its local clock value
2. Upon reception, B records the current time T2 of its local clock
3. B returns a response message to A with T2 and the current time T3 of its clock
4. Upon reception, A records the current time T4 of its local clock



Physical time

Cristian's algorithm [1989] (con't)

- If clock rate increases monotonically (without even decreasing) and
- If both message transmissions take roughly the same time
- *Round-trip time* = $(T_2-T_1)+(T_4-T_3)$
- *Offset* of A relative to B = $T_4 - T_3 - RTT/2$



- Algorithm A can re-adjust its time to T3 + RTT/2
- This results is not accurate but gives an acceptable approximation

Physical time

Network Time Protocol (NTP)

- Servers are divided into *strata*
  - A server having an atomic clock operates at stratum 0
  - A server synchronized with a server at stratum j operates at stratum j+1

- This protocol is set up pair-wise between servers

1. Run Cristian's algorithm 8 times and records the 8 output pairs $<\text{Offset}, \text{RTT}/2>_i$ $(0 \leq i < 8)$
2. The pair with the minimum RTT/2 is chosen as the most precise output
3. The server with the higher stratum adjusts
   - its time using the Offset from the chosen pair
   - its stratum to the other server stratum + 1

- What if there is no server of stratum 0? Can we still relatively synchronize?

Physical time

The Berkeley Algorithm

- Server is active, it starts the algorithm periodically to resynchronize nodes

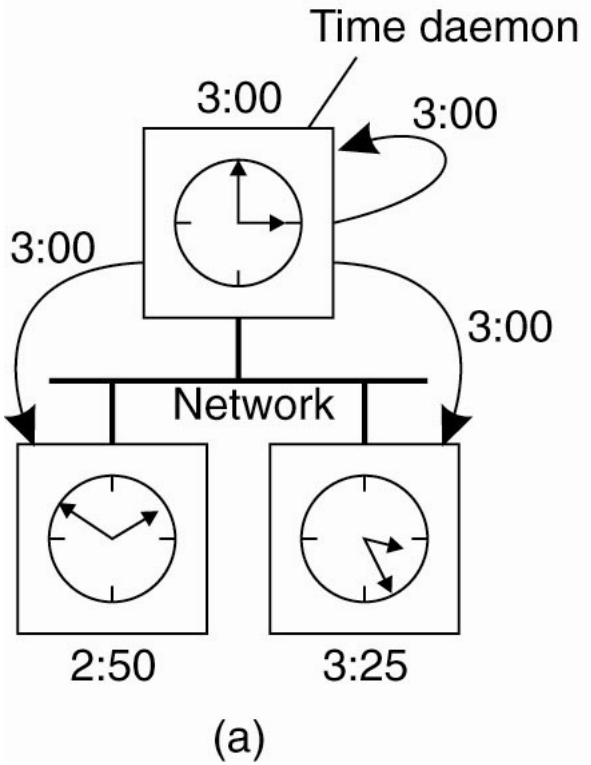
1. Time server (a.k.a., daemon) sends a “what time is it?” request to all nodes
2. The nodes answer with their local clock value
3. Once the server has received values from the nodes
   - It computes an average value
   - It sends it to the nodes for them to adjust

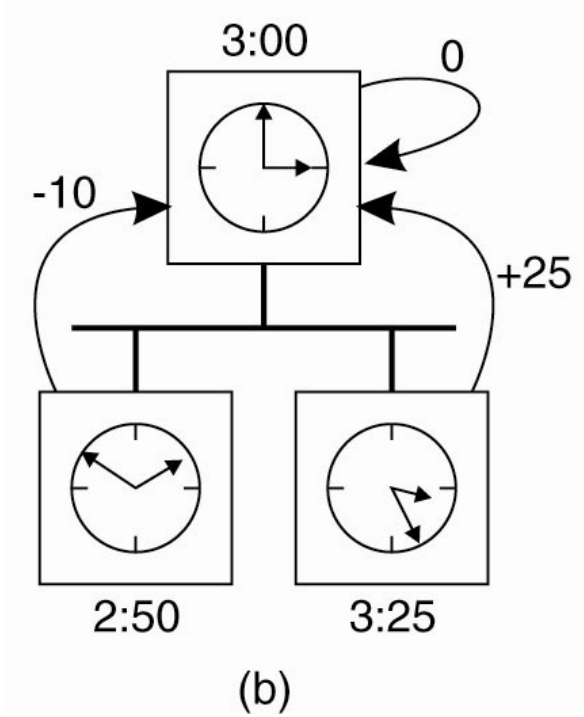
- The time can be far from the UTC, hence the time is not absolutely synchronize
- This relative synchrony is sufficient in many cases

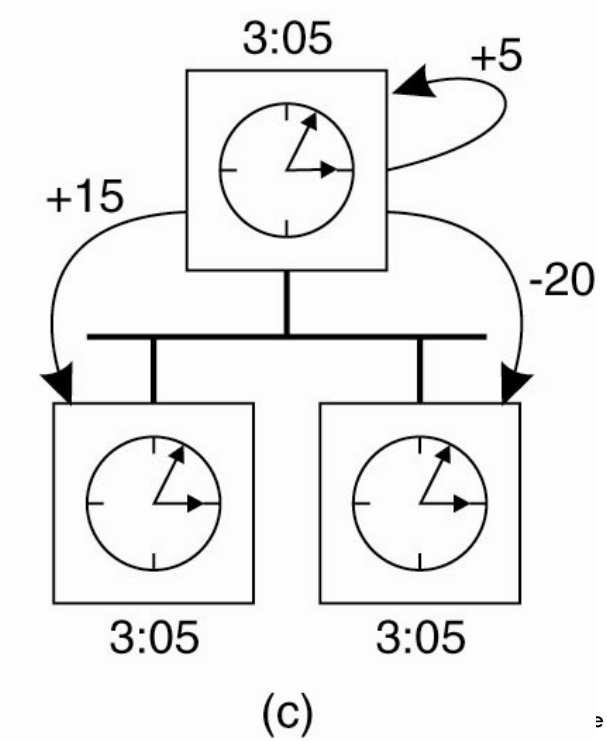
Physical time

The Berkeley Algorithm (con't)

- (a) the time daemon sends a request to all nodes
- (b) the nodes answer the time daemon about their local clock
- (c) the server computes the average and tells nodes to adjust their







Physical time

Pros and cons of physical time

- Pros: very powerful
  - Universal representation of the occurrence time of all distributed events
  - All events are made comparable (total order)

- Cons: hard to achieve
  - This physical time instruments are difficult to synchronize
    - They require the message transmission to have an upper-bounded duration
    - They sometimes require message duration to be similar (also lower bounded)
  - In large-scale system, the message delay is typically
    - High (in seconds) and the synchronization can be biased by message delays
    - Heterogeneous (LAN: milliseconds, WAN: seconds) 3 orders of magnitude far apart
    - Unpredictable, unbounded message delays and message can be delivered in disorder

- Let's look at logical time instead, where events may not always be comparable (partial order)

Logical Time

THE UNIVERSITY OF SYDNEY

Logical time

Definition: processes, events, histories

- A distributed system is modeled with multiple distinctly identified *processes*
- An *event* is simply a computational event executed locally by some process
- A *history* is a sequence of events, each indexed by its corresponding process

- Happens-before relation "a happens before b", denoted by a $\rightarrow$ b if:
  - a and b are events from the same process such that a occurs before b,
  - 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 **or**
  - it exists some event c such that a $\rightarrow$ c and c $\rightarrow$ b (i.e., transitive closure)
- Some events may not be comparable
  - It may happen that neither a $\rightarrow$ b nor b $\rightarrow$ a
  - In this case, we say that a and b are *concurrent*
- Let n be the number of processes present in the distributed system

Logical time

Logical clocks

- Goal: to make the happens-before relation visible to processes
- Key idea:
  - each process should keep track of the order in which events appear to take place locally
  - let's introduce a function logical clock C: events ↦ integers
- How does a process p_i maintain its logical clock C_i:

1. Each process p_i with identifier i ($0 \leq i < n$) keeps a variable C_i, initially 0
2. Upon sending, p_i increments C_i and sends it in its message
3. Upon reception of C_j from p_j, p_i increments C_i such that $C_i > C_j$

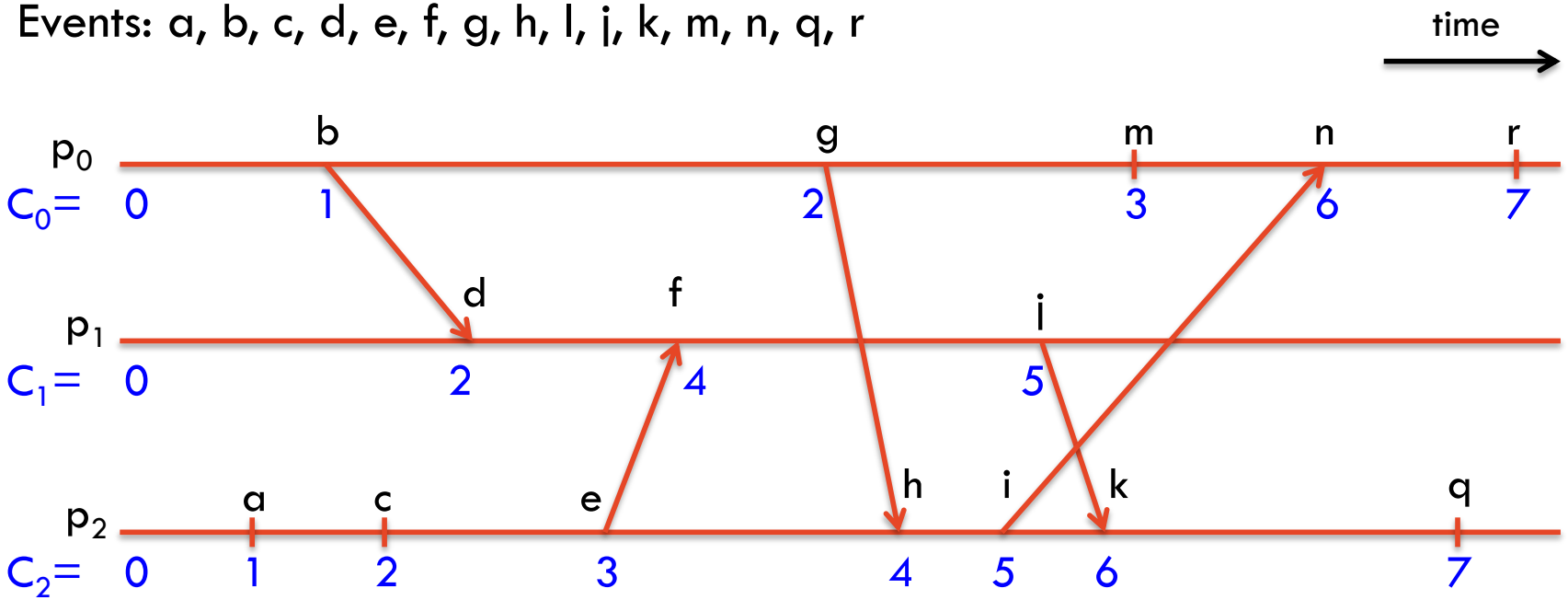
- Result: Let C(a) and C(b) be the value chosen for events a and b in the system, this algorithm guarantees that:

$$\text{If } a \rightarrow b \text{ then } C(a) < C(b)$$

Logical time

Logical clocks (con't)

- Processes: p_0, p_1, p_2
- Events: a, b, c, d, e, f, g, h, I, j, k, m, n, q, r

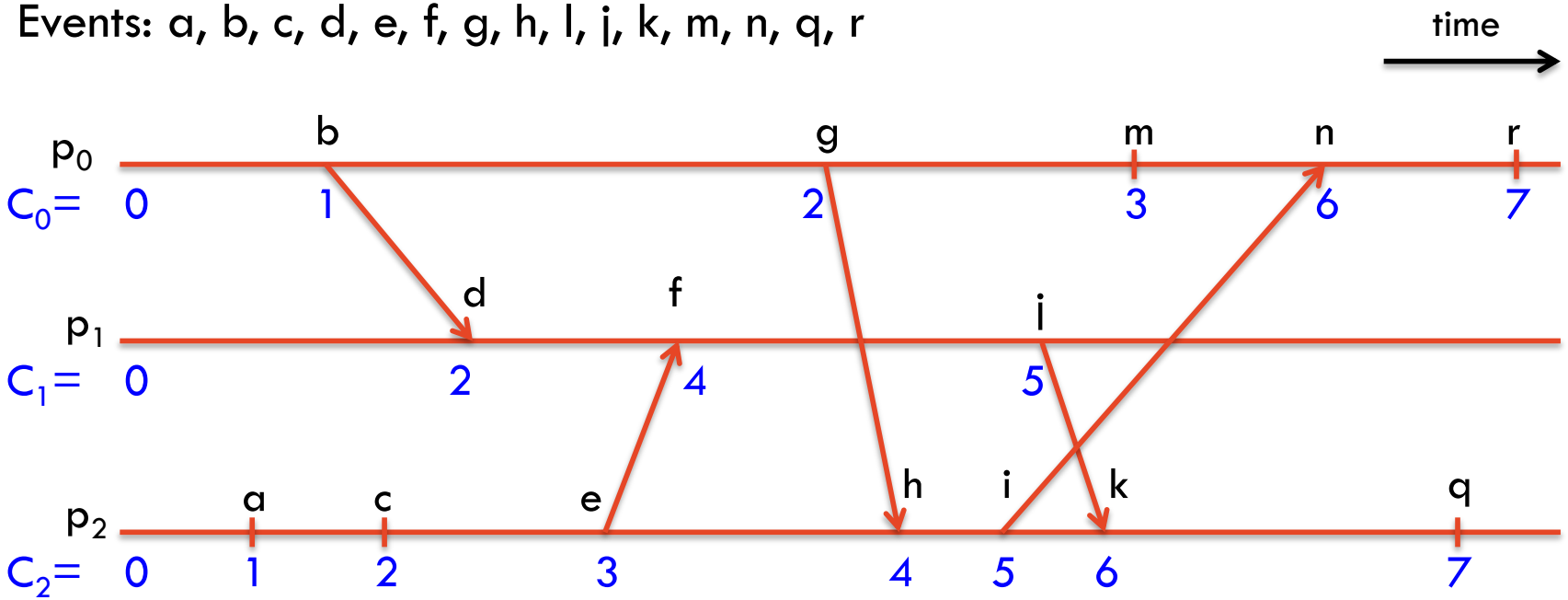


- c ⟶ n, j ⟶ k, e ⟶ h, etc.
- a and b are concurrent and a and d are concurrent but b ⟶ d

Logical time

Logical clocks (con't)

- Processes: p_0, p_1, p_2
- Events: a, b, c, d, e, f, g, h, l, j, k, m, n, q, r



- $c \rightarrow n$, $j \rightarrow k$, $e \rightarrow h$, etc.
- a and b are concurrent and a and d are concurrent but $b \rightarrow d$

Even though $C(a) < C(d)$, we have that $a \not\rightarrow d$

Logical time

Vector clocks

- Goal: to make also the did-not-happen-before relation visible
- Key idea:
  - each process should also maintain an approximation of the clock of other processes
  - let's introduce a function vector clock VC: events $\mapsto$ integers $\times$ processes
- How does a process p_i maintain its vector clock VC_i:

1. Each process p_i with identifier i ($0 \leq i < n$) keeps a vector VC_i of integers, initially 0
2. Upon sending, p_i increments $VC_i[i]$ and then sends a message m with timestamp $ts(m) = VC_i$
3. Upon reception of ts(m) from p_j, p_i sets $VC_i[k]=\max(VC_i[k], ts(m)[k])$ for each k and $VC_i[i] = VC_i[i]+1$

Logical time

Vector clocks (con't)

- Result:

  Let VC(a) and VC(b) be the values chosen for events a and b,

  We order vector clocks by comparing their coordinates

  - One is lower than another, if each of its coordinates is lower than the corresponding coordinate of the other
    $VC(a) \leq VC(b)$ if for all i $(0 \leq i < n)$ $VC(a)[i] \leq VC(b)[i]$
  - One is strictly lower than another, if it is lower and different from the other
    $VC(a) < VC(b)$ if $VC(a) \leq VC(b)$ and $VC(a) \neq VC(b)$

  This algorithm guarantees that:

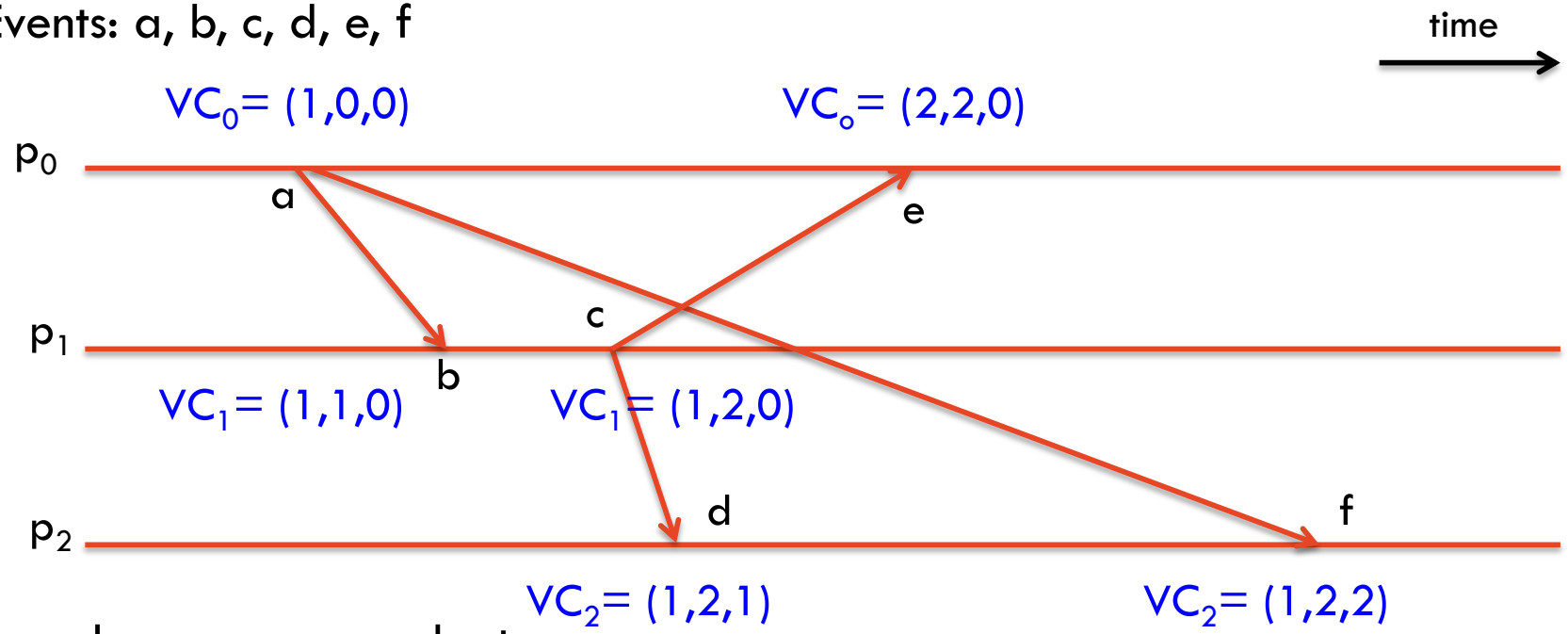
  - If $VC(a) < VC(b)$ then $a \rightarrow b$
  - If $a \rightarrow b$ then $VC(a) < VC(b)$

$$VC(a) < VC(b) \text{ if and only if } a \rightarrow b$$

Logical time

Vector clocks (con't)

- Processes: p_0, p_1, p_2
- Events: a, b, c, d, e, f



- $a \rightarrow b$, $a \rightarrow c$, $a \rightarrow d$, etc.
- $e \not\rightarrow f$ and $f \not\rightarrow e$

$VC(i) < VC(j)$, if and only if $i \rightarrow j$

Logical time

Logical vs vector clocks

- Clocks give an indication on the ordering of event by using logic instead of physical instrument

- Intuitively, if some event occurred before another then it should be marked by a lower (logical or vector) clock value

- If some process observes that an event a has a strictly lower logical clock than another event b
  - then the process can conclude *a cannot happen after b*
  - the process cannot distinguish:
    - Whether a happened before b
    - Or a and b are concurrent

- If some process observes that an event a has a strictly lower vector clock than another event b
  - then the process can conclude a cannot have happened after b
  - And more particularly, that *a happened before b*

Multicast

THE UNIVERSITY OF SYDNEY

Multicast

Related terminology

- Broadcast: one-to-all communication
  - Action of sending a message to all nodes of the system
  - Typically used for relatively small systems, like IP broadcast as part of Ethernet in LANs
- Unicast: one-to-one communication
  - In contrast with broadcast, describe a special form of distribution to a single receiver
  - Used generally in the context of multimedia or streaming applications
- Anycast: one-to-random-one communication
  - Send a message to an IP address range from to obtain a response from any potential receiver, whose IP address belongs to this range
  - Used with UDP and TCP
- Geocast: one-to-geographical-neighbors communication
  - Send a message to the closest nodes
  - Used in mobile ad-hoc networks where communicating with further nodes may be impossible (e.g., limited transmission range, limited energy)
- Multicast: one-to-many communication
  - Action of sending a message to multiple nodes of the system (not necessarily all nodes)
  - This term is used in many contexts (network, algorithm)
- Note: we will use the general term multicast in the following

Multicast

Definitions

- Ordering of messages:
  - *Totally ordered*: for all messages m1 and m2 and all processors pi and pj, if pi delivers m1 before it delivers m2, then m2 is not delivered at pj before m1 is
  - *Causally ordered*: for all messages m1 and m2 and every processor pi, if m1 happens before m2, then m2 is not delivered at pi before m1 is
- Reliability:
  - *Integrity*: every message received was previously sent; no message is received "out of thin air"
  - *No duplicates*: no message is received more than once at any single process
  - *Liveness*: all messages broadcast by a processor are eventually received by all processors
- In the following we assume reliability and we ensure ordering of messages

Multicast

First-in first-out (FIFO) multicast

- Goal: broadcasting messages that got delivered in the same order at all nodes

- Each process maintains a counter and a priority queue of messages (waiting to be delivered to the top layer)

1. A processor p_i multicasts a message with a sequence number corresponding to its current counter value and then increments this counter by 1
2. The recipient of a message from p_i with sequence number T puts the message in the queue according to number T and waits to perform the FIFO delivery until it has done so for all messages from p_i with sequence numbers less than T

Multicast

Totally ordered multicast

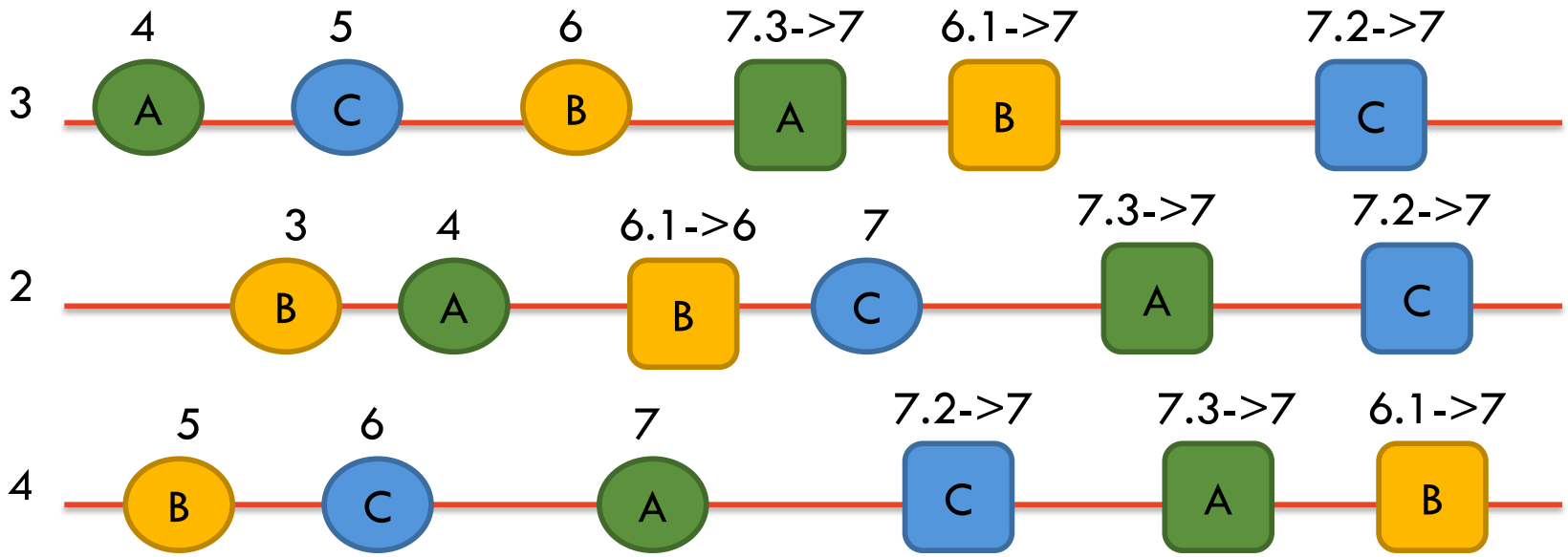
- Goal: broadcasting messages that got delivered in the same order at all nodes

- Each process maintains a logical clock and a priority queue of messages (waiting to be delivered to the top layer)
  1. A processor p_i increments its logical clock and fifo-multicasts a message with the current logical clock as the timestamp message
  2. Any incoming message is queued in a priority queue according to its temporary timestamp, and acknowledged using a fifo-multicast message to every other processes (including the sender)
  3. Once all processes have acknowledged, the sender sets the definitive clock as the current clock it has for this message
  4. A message in the queue is delivered once there is a message, acknowledged by all, from each process with a larger timestamp in the queue

Multicast

Totally ordered broadcast algorithm (con't)

- Initial clocks are 3, 2, 4 at resp. processes 1, 2, 3; messages are A, B, C
- Circles (resp. square) represent msg reception (resp. last ack reception)
- 7.3 represents clock 7 received from 3, max(local, received) is taken



- Messages are delivered in the order of their clocks, B, C, A, everywhere

Multicast

Causal ordered multicast

- Goal: broadcasting messages that got delivered only if all causally preceding messages have already been delivered

- Each process maintains a vector clock (similar as before) and a priority queue of messages (waiting to be delivered to the top layer)

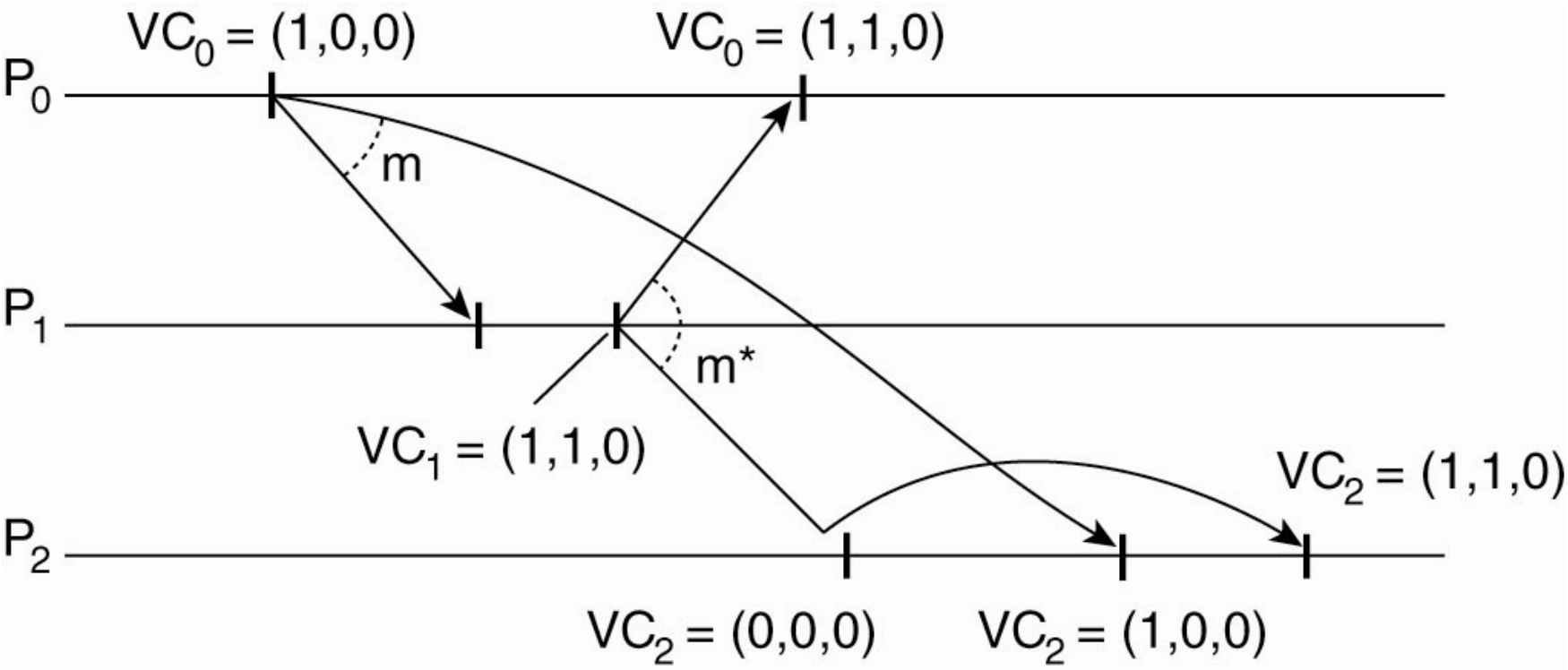
Processor p_i increments $VC_i[i]$ only when sending a message (not when receiving) and p_i adjusts VC_i when receiving a message.

p_i postpones delivery of m from p_i until both conditions are met:

1. $ts(m)[i] = VCj[i] + 1$
2. $ts(m)[k] \leq VCj[k]$ for $k \neq i$

Multicast

Causal ordered multicast (con't)



- Take $VC_2 = [0,2,2]$, $ts(m) = [1,3,0]$ from p_0. What information does p_2 have, and what will it do when receiving m (from p_0)?

Mutual Exclusion

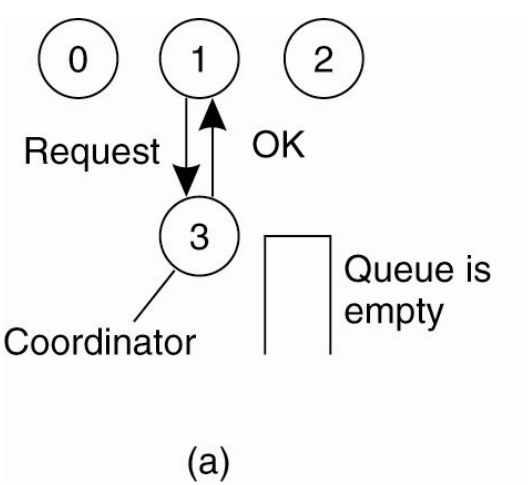
THE UNIVERSITY OF SYDNEY

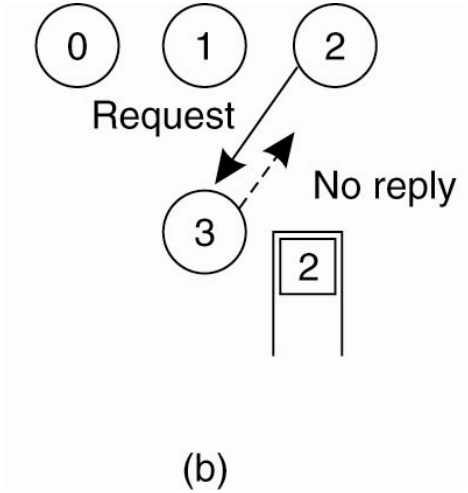
Mutual exclusion

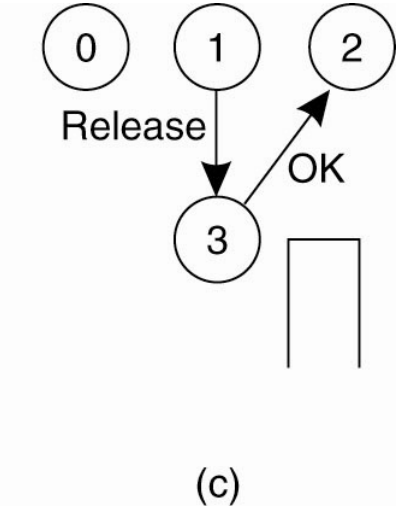
- Problem: Multiple processes in a distributed system want exclusive access to some resource

Mutual exclusion

Centralized algorithm







- (a) Process 1 asks the coordinator for permission to access a shared resource; permission is granted
- (b) Process 2 then asks permission to access the same resource. The coordinator does not reply
- (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2

Mutual exclusion

Decentralized algorithm

- Principle. Assume every resource is replicated n times, with each replica having its own coordinator ⇒ access requires a majority vote from m > n/2 coordinators. A coordinator always responds immediately to a request.

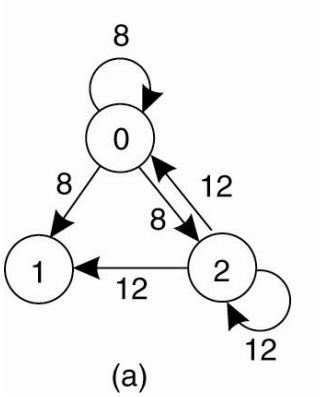
- Assumption: When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.

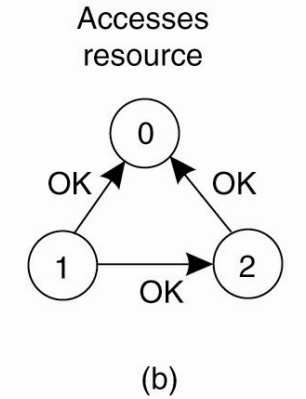
- Problem: A coordinator may grant permission to some node whereas it already did it to another node

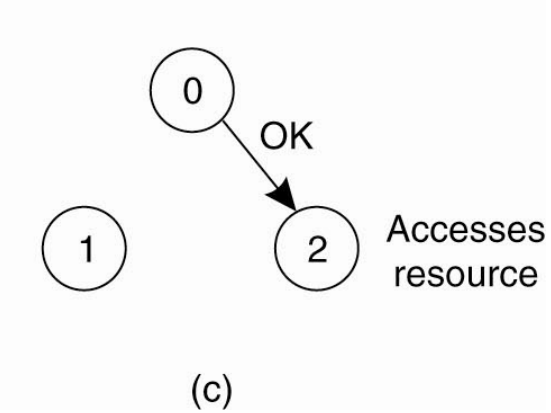
Mutual exclusion

Ricart and Agrawala algorithm

- Principle. The same as before except that acknowledgments are not sent. Instead, replies (i.e., grants) are sent only when
  - The receiver has no interest in the shared resource or
  - The receiver is waiting for the resource, but has lower priority (known through comparison of timestamps)
  - In all other cases, reply is deferred, implying some more local administration



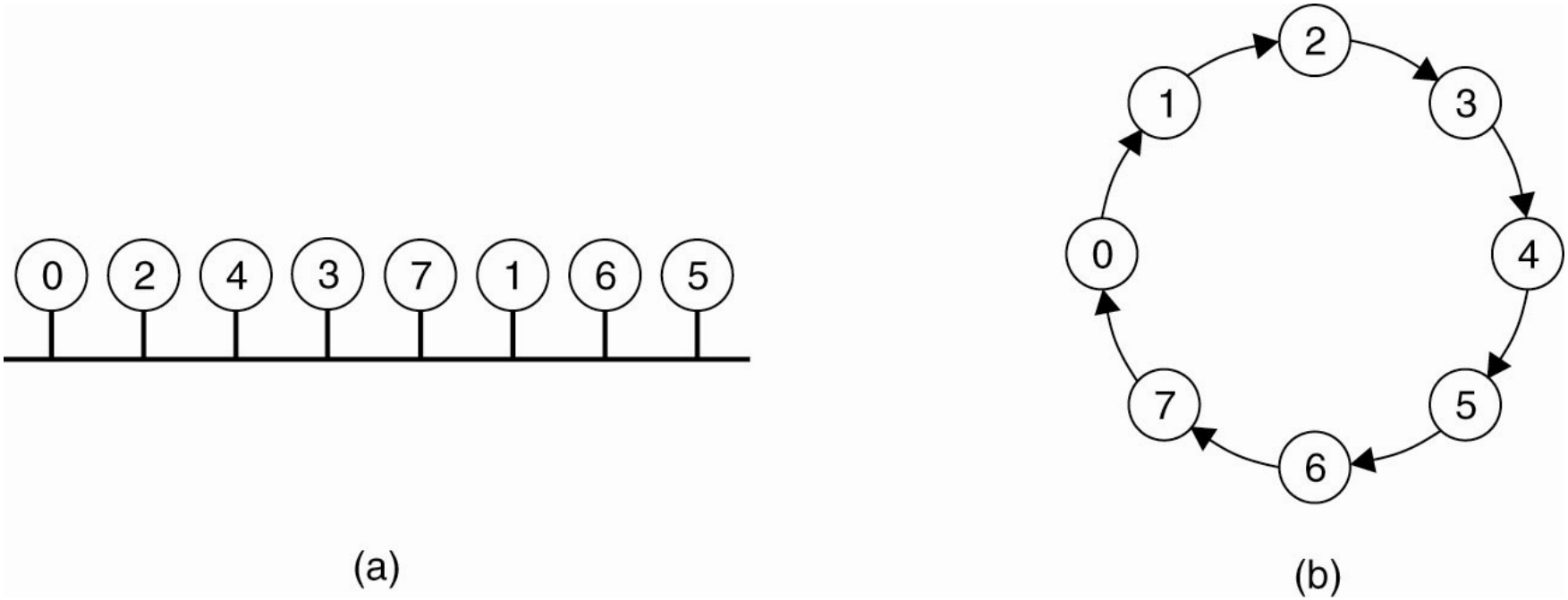




  - Two processes want to access a shared resource at the same moment
  - Process 0 has the lowest timestamp, so it wins.
  - When process 0 is done, it sends an OK also, so 2 can now go ahead.

Mutual exclusion

Token ring algorithm



- Principle. Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to accessing the resource (if it wants to).

Mutual exclusion

Comparison of the last four solutions

- Messages and delay are necessary to get granted the permission to access the resource (this means entering the critical section) or to release the permission (exiting it)

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	3mk, k = 1,2,...	2 m	Starvation, low efficiency
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Leader Election

THE UNIVERSITY OF SYDNEY

Leader election

- Problem: An algorithm requires that some process acts as a coordinator. The question is how to select this special process

- Note. In many systems the coordinator is chosen once for all (e.g., file servers). This leads to centralized solutions ⇒ single point of failure

- A fully distributed solution one without a coordinator is not always more robust than any centralized coordinated solution
  - E.g., rotating leader vs. fixed majority system (with a fixed intersection)

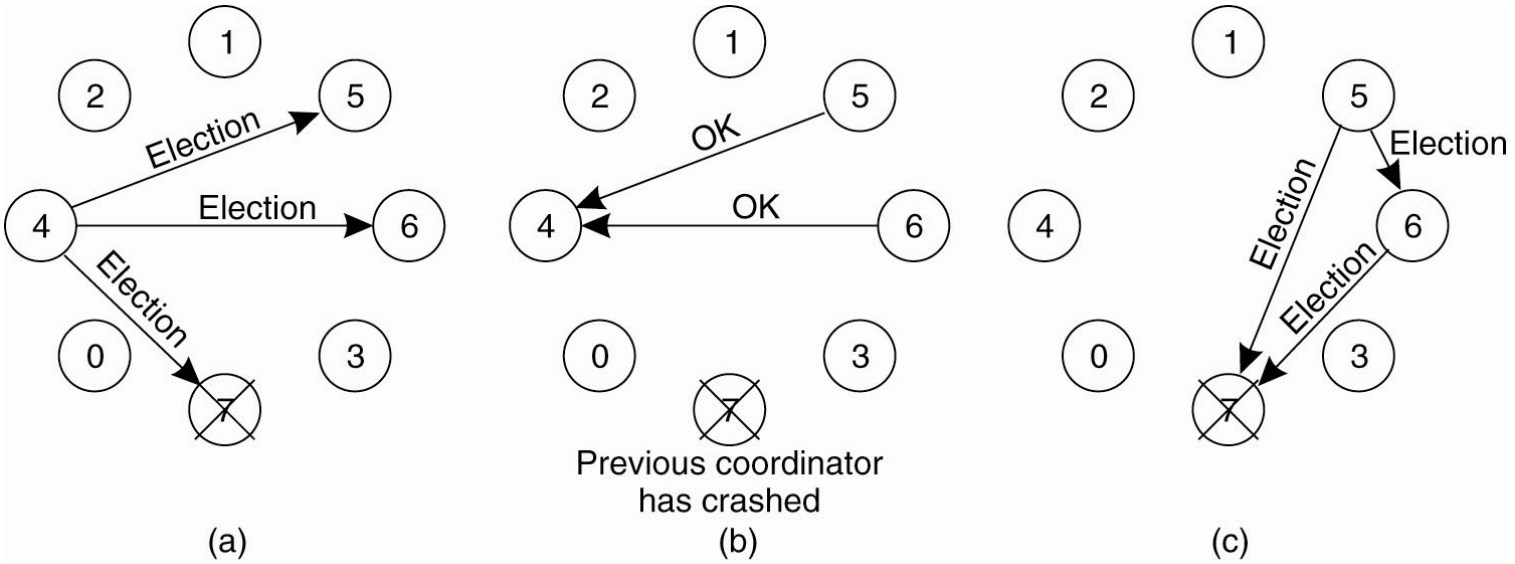
Leader election

The Bully Algorithm

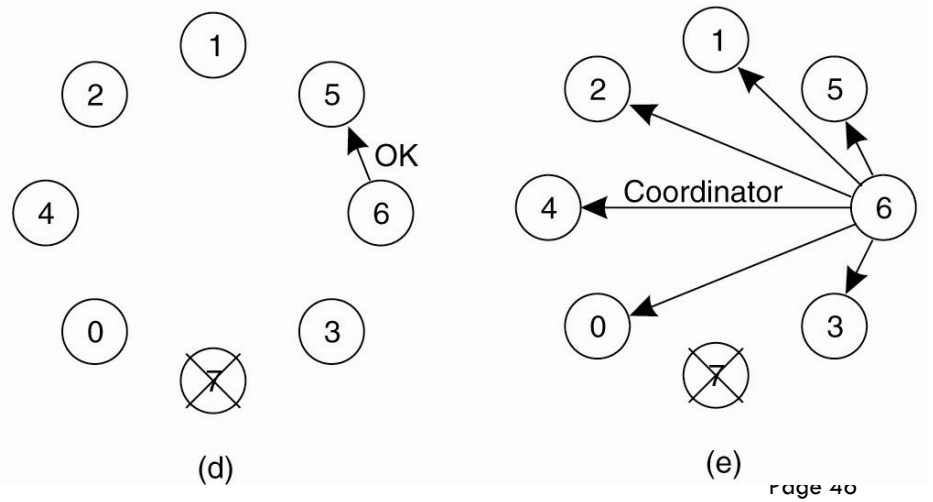
1. p sends an *ELECTION* message to all processes with higher numbers.
2. If no one responds, p wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. *P*'s job is done.

Leader election

The Bully Algorithm (con't)



- (a) Process 4 holds an election
- (b) Processes 5 and 6 respond, telling 4 to stop
- (c) Now 5 and 6 each hold an election
- (d) Process 6 tells 5 to stop
- (e) Process 6 wins and tells everyone



Leader election

Election in a ring

- Principle. Process priority is obtained by organizing process into a (logical) ring. Process with the highest priority should be elected as coordinator.
  - Any process can start an election by sending an election messages to its successor. If a successor is down, the message is passed on to the next successor.
  - If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known
  - The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator