Distributed Systems: Logical Clocks

Anca Jurcut
E-mail: anca.jurcut@ucd.ie

School of Computer Science and Informatics
University College Dublin
Ireland

Logical Clocks

- A second approach to dealing with distributed synchronization is based on the concept of “relative time”.

- Based on the notion of ***event ordering***:
  - **Temporal Ordering**: the time of occurrence in real-time
    - induces a ‘happens-before’ relation
  - **Causal Ordering**: cause-effect relationship; implies temporal order

- Note that (with this mechanism) there is no requirement for “relative time” to have any relation to the “real time”.

- Such “clocks” are referred to as ***Logical Clocks***.

Lamport’s Logical Clocks

- Lamport made two observations about the nature of process interaction:
  - **First point**: if two processes do not interact, then their clocks do not need to be synchronized – they can operate concurrently without fear of interfering with each other.
  - **Second (critical) point**: it does not matter that two processes share a common notion of what the “real” current time is. What does matter is that the processes have some agreement on the order in which certain events occur.

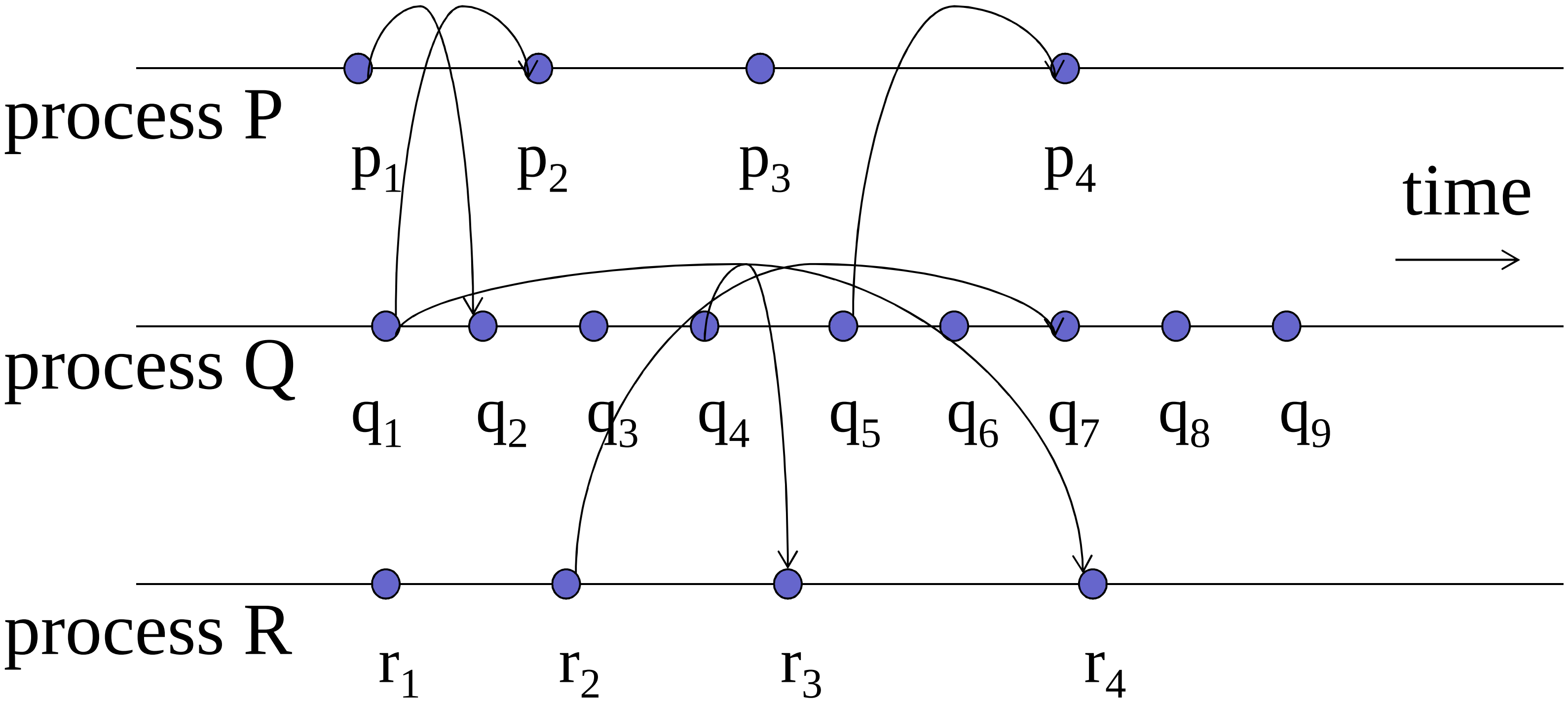
- Based upon these observations, he devised the “happens-before” relation (denoted “$\rightarrow$”).

The “Happens-Before” Relation

- Properties of “→”:
  - If A and B are events in the same process, and A occurs before B, then we can state that ***A “happens-before” B*** (denoted A → B)
  - If A is the event of sending a message, and B is the event of receiving the same message, then A → B.
  - If A → B and B → C, then A → C (**transitivity**).
  - If A and B are not related by →, then they occurred at the same time (i.e. they are ***concurrent*** events).

- We can make some deductions about the current clock time based on “→”:
  - If C(A) is the time when A occurred, then C(A) < C(B)

Example



happens before: $p_1 \rightarrow q_2$, $r_2 \rightarrow r_3$, $q_4 \rightarrow r_4$, $r_1 \rightarrow q_9$, …

concurrent: p_1 and q_1 ; p_3 and q_3 ,q_4, q_5 ; q_5 and r_3 , r_4 ; …

The “Happens-Before” Relation

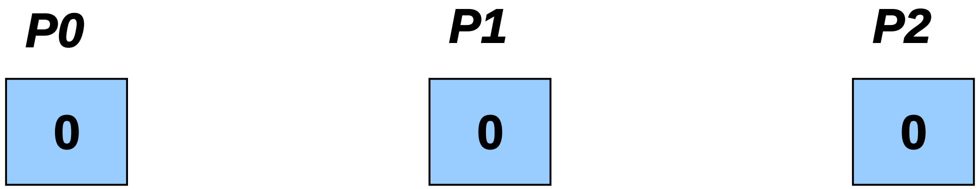
- Now, assume three processes are in a Distributed System: A, B and C.
  - All have their own physical clocks (which are running at differing rates due to “clock skew”, etc.).
  - A sends a message to B and includes a “timestamp”.
  - If this sending timestamp is less than the time of arrival at B, things are OK, as the “happens-before” relation still holds (i.e. A “happens-before” B is true).
  - However, if the timestamp is more than the time of arrival at B, things are NOT OK (as A “happens-before” B is not true, and this cannot be as the receipt of a message has to occur after it was sent).

The “Happens-Before” Relation

- The question to ask is:
  - How can some event that “happens-before” some other event possibly have occurred at a later time??

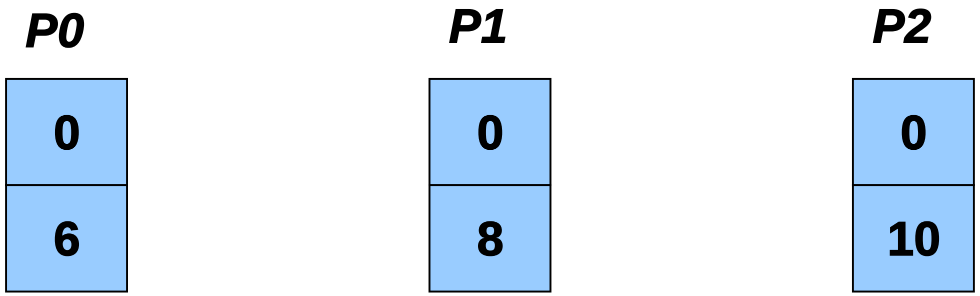
- The answer is: it can’t!
  - So, Lamport’s solution is to have the receiving process adjust its clock forward to one more than the sending timestamp value.
  - This allows the “happens-before” relation to hold, and also keeps all the clocks running in a synchronized state.
  - The clocks are all kept in sync relative to each other.

Example



P0	P1	P2
0	0	0

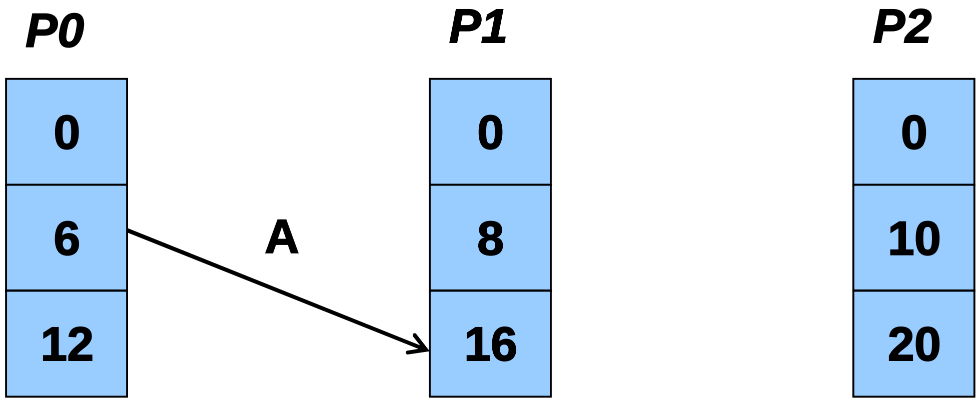
Example



P0	P1	P2
0	0	0
6	8	10

- P0 sends message A to P1

Example



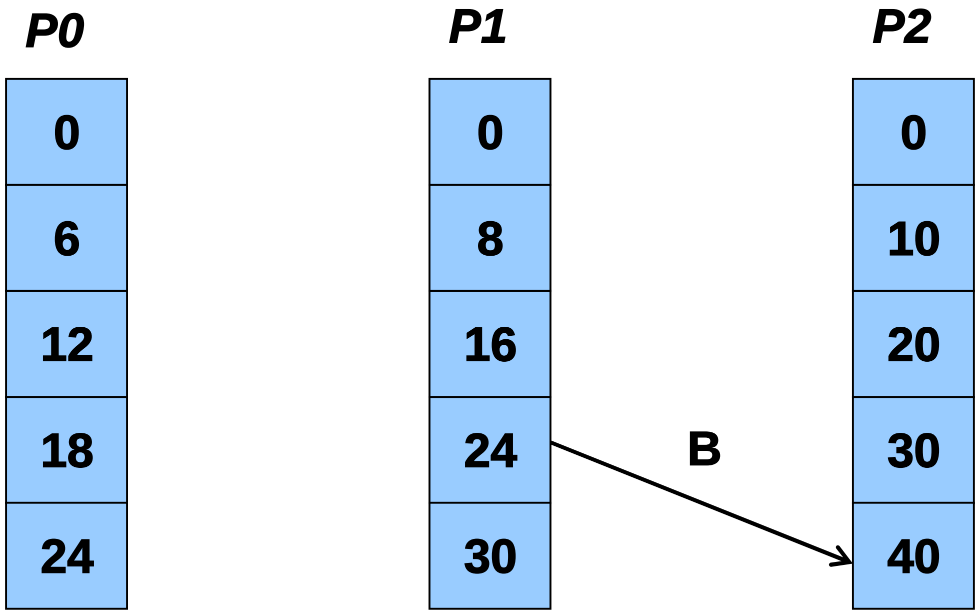
- P1 receives message A (everything is OK since 6 < 16)

Example

P0	*P1*	*P2*
0	0	0
6	8	10
12	16	20
18	24	30

- P1 sends message B to P2

Example



P0	*P1*	*P2*
0	0	0
6	8	10
12	16	20
18	24	30
24	30	40

- P2 receives message B (everything is OK since 24 < 40)

Example

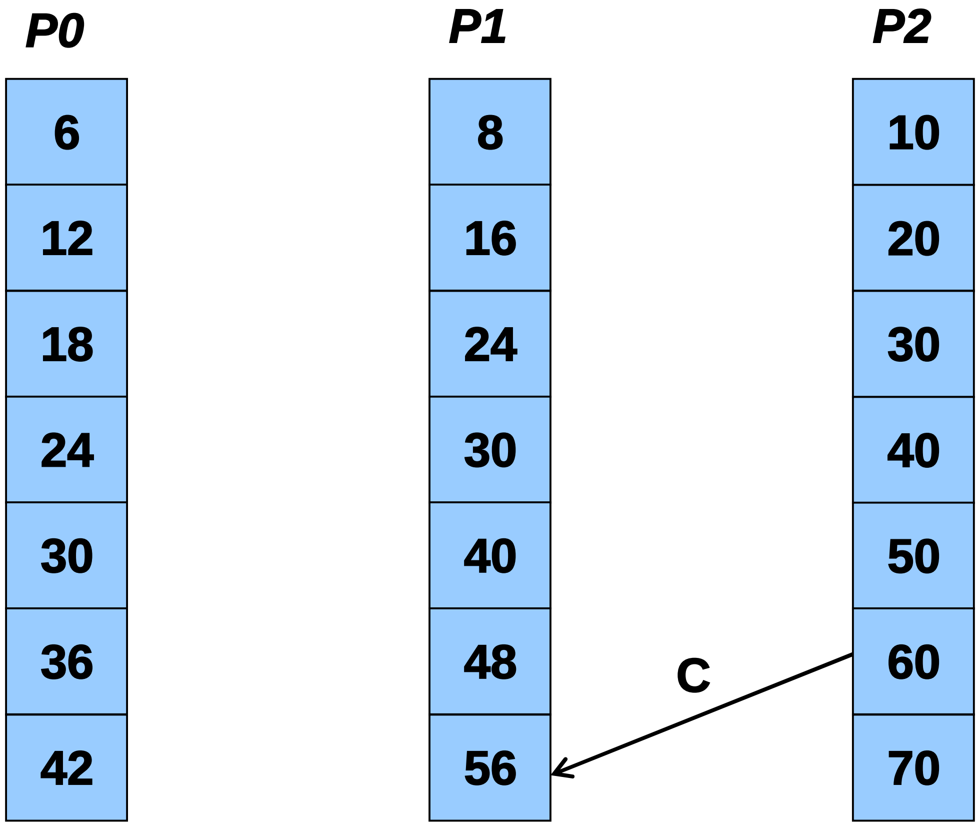
P0	*P1*	*P2*
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50

Example

P0	*P1*	*P2*
0	0	0
6	8	10
12	16	20
18	24	30
24	30	40
30	40	50
36	48	60

- P2 sends message C to P1

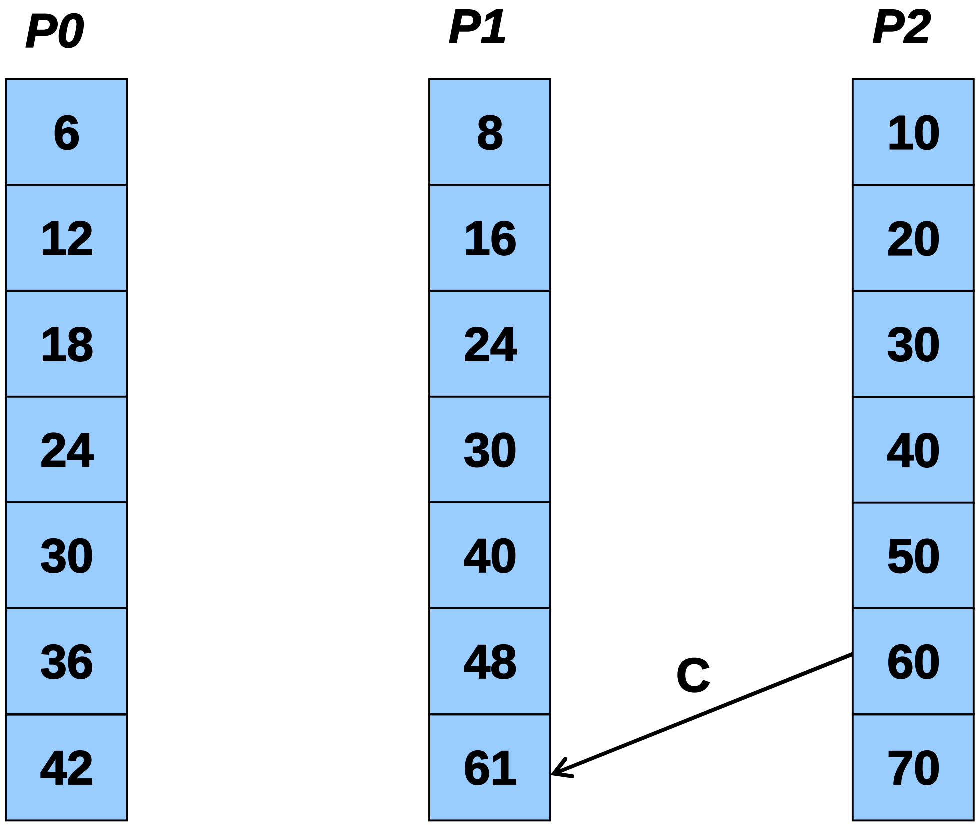
Example



P0	P1	P2
6	8	10
12	16	20
18	24	30
24	30	40
30	40	50
36	48	60
42	56	70

- P1 receives message C (Ouch! The message was sent at time 60 but received at time 56)

Example



P0	P1	P2
6	8	10
12	16	20
18	24	30
24	30	40
30	40	50
36	48	60
42	61	70

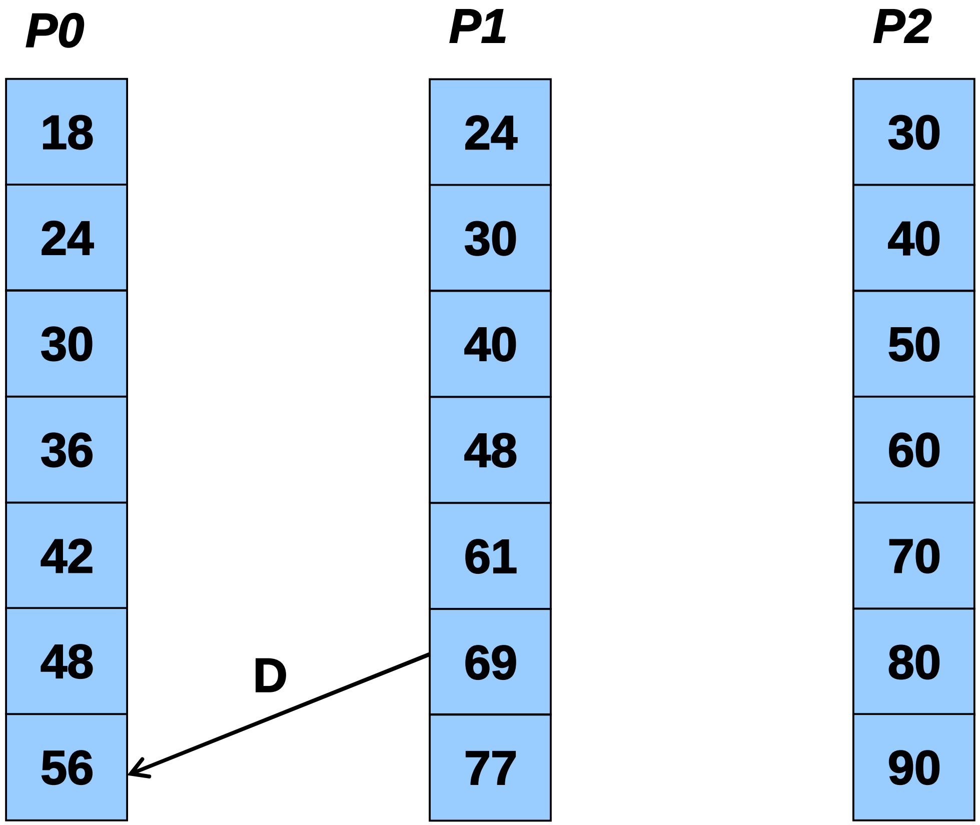
- Logical Time at P1 updated to be 1 greater than the time C was sent at.

Example

P0	*P1*	*P2*
12	16	20
18	24	30
24	30	40
30	40	50
36	48	60
42	61	70
48	69	80

- P1 sends message D to P0

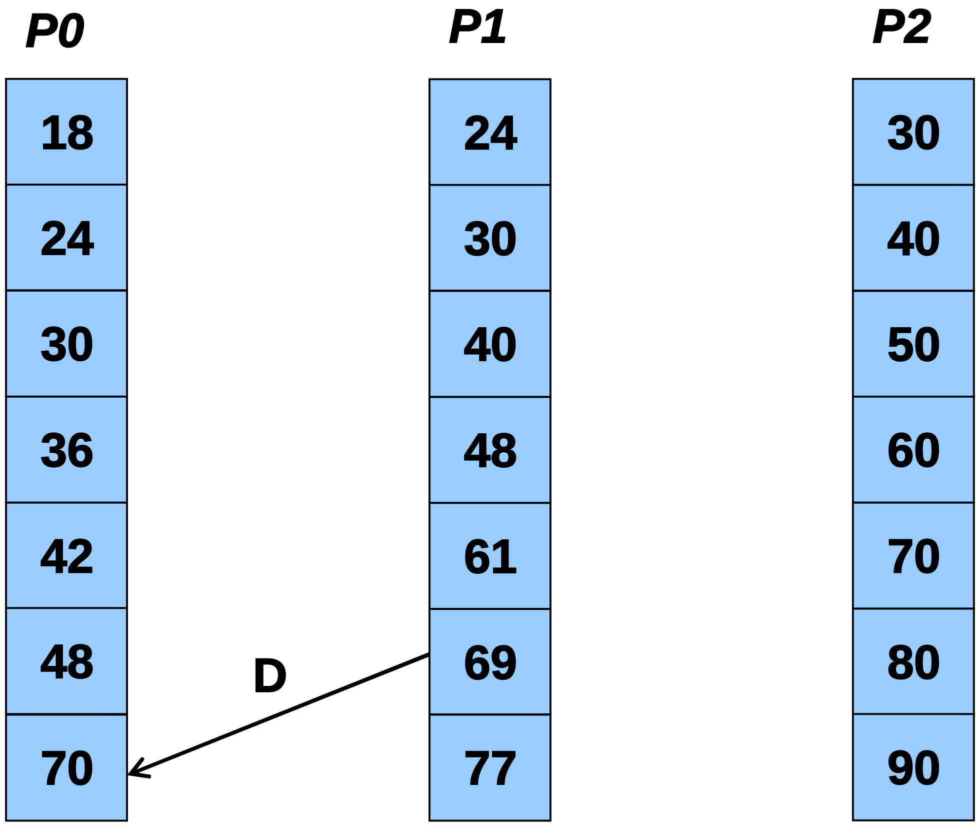
Example



P0	*P1*	*P2*
18	24	30
24	30	40
30	40	50
36	48	60
42	61	70
48	69	80
56	77	90

- P0 receives message D (Ouch! 56 < 69)

Example



P0	P1	P2
18	24	30
24	30	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90

- Logical Time at P0 updated

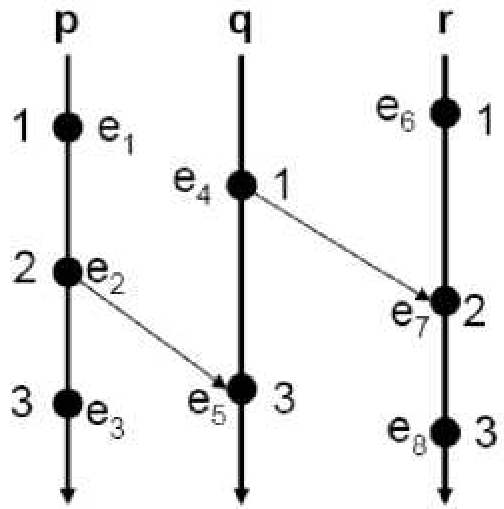
Example

P0	*P1*	*P2*
24	30	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

- End of Run

Limitation of Lamport's algorithm

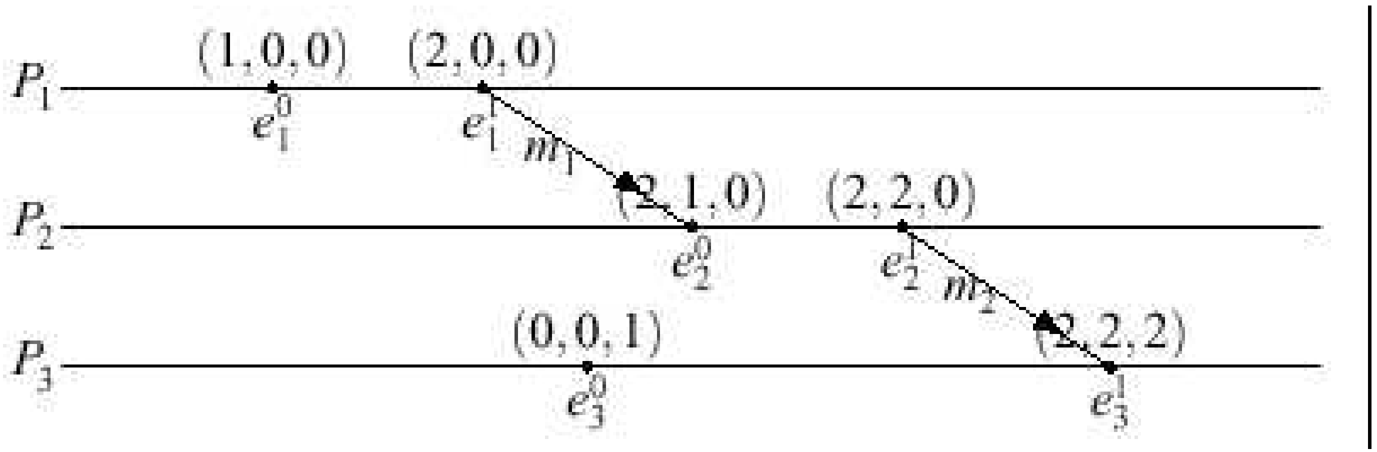
- If e1 → e2 then e1.TS < e2.TS (TS = "Time Stamp")
- But it is not necessarily the case that if e1.TS < e2.TS then e1 → e2



e1.TS < e8.TS, but is e1 → e8 true?

Given two events, we cannot say whether they are causally related from their timestamps.

Vector Time



- Alternative to Lamport’s Algorithm in which a global timestamp is used:

- To each event *e* we now want to associate timestamps V(e) such that

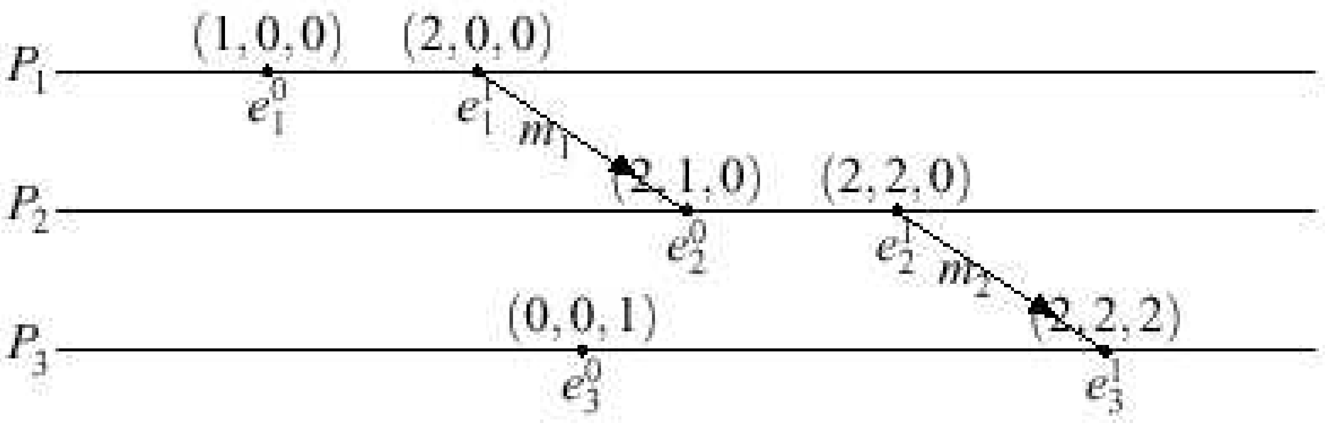
  e ì *e’* ę *V(e)* < *V(e’)*

- We let V(e) be a vector $(t_1, . . . , t_N)$. Intuitively, for each $1 \leq i \leq N$ the latest event from process *i* which could have influenced *e* is $e_i^{t_i}$

- We define:

  $(t_1 , . . . , t_N) < (t'_1 , . . . t'_N)$ ę $t_1 < t'_1 , . . . , t_N < t'_N$

Vector Time



- The timestamps are formally defined as:

$$
\begin{aligned}
V(e_i^{-1}) &= (\overbrace{0,\ldots,0}^{N}) \\
V(e_i^j) &= (\overbrace{0,\ldots,0}^{i-1},1,\overbrace{0,\ldots,0}^{N-i}) \\
&\quad + \begin{cases} V(e_i^{j-1}) & \text{if } e_i^j \neq recv(m) \\ \max(V(e_i^{j-1}),V(send(m))) & \text{if } e_i^j = recv(m) \end{cases}
\end{aligned}
$$

for any m, j > 0 and 1 ≤ i ≤ N. Here max and + are coordinate wise.

Distributed Systems:
Global State

Introduction

- Sometimes we need to capture the global state of a distributed system.
  - E.g. Deadlock Detection, Termination Detection, Distributed Debugging, …

- Capturing the state of the individual processes is relatively easy:
  - We simply write the state of each process to the hard disk.

- How can we synchronise this write activity to capture the state of all the processes simultaneously?

Introduction

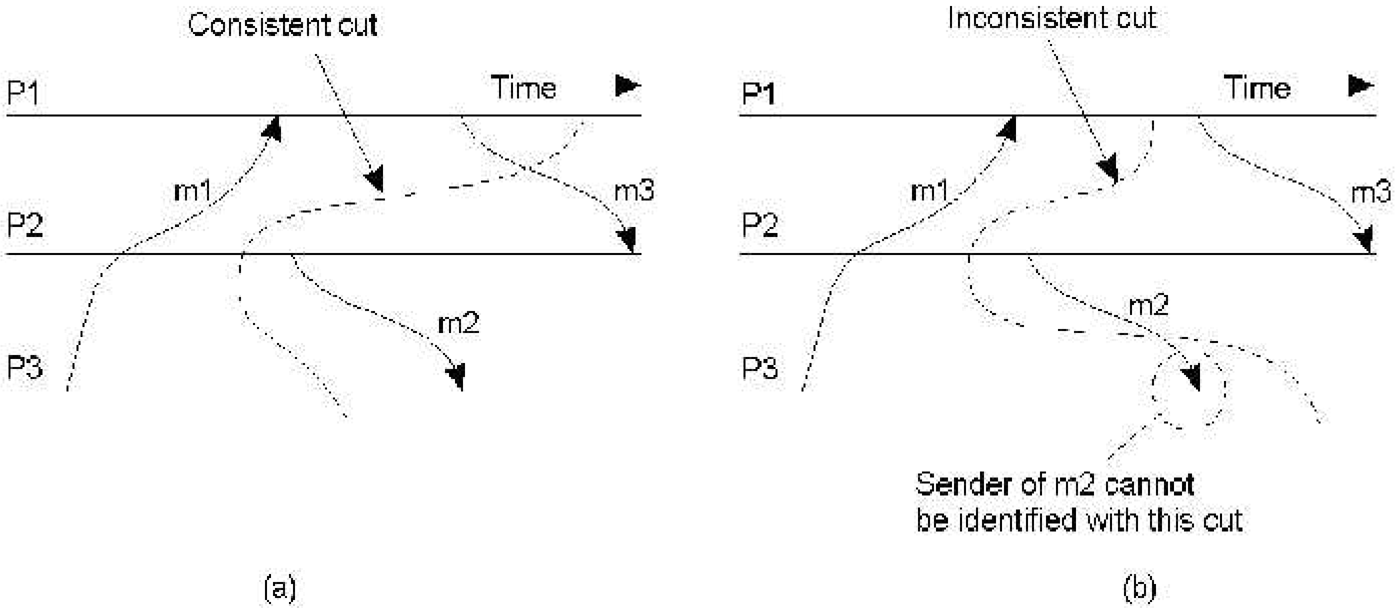
- Naïve Approach: We send a global “capture state message”

- Upon receipt of this message, each process writes it state to disk.

- But, in large systems this may take time:
  - This can lead to “Captured” processes continuing to receive messages from “Uncaptured” processes

- How can we ensure that the state we capture is **consistent**?

Consistent Global State



- A consistent cut (a ***possible*** distributed state)
- An inconsistent cut (it is impossible for the system to ever have been in this state)

Snapshot algorithm

- Developed by Chandy and Lamport in 1985
  - The paper is provided on moodle– please read

- Assumptions:
  - neither channels nor processes fail and communication is reliable so that every message send is eventually received intact, exactly once
  - channels are unidirectional and provide FIFO-ordered message delivery
  - the graph of processes and channels is strongly connected (there is a path between any two processes)
  - any process may initiate a global snapshot at any time
  - the processes may continue their execution and send and receive normal messages while the snapshot takes place

Snapshot algorithm

- Requires the implementation of two “rules” that together ensure a correct snapshot is taken:

- A “Marker” message is sent around the system to instruct processes to take snapshots, and to get them to record the state of communication channels.

- Marker sending rule for process P

- After P has recorded its state, for each outgoing channel, C:
  - P sends one marker message over C (before it sends any other message)

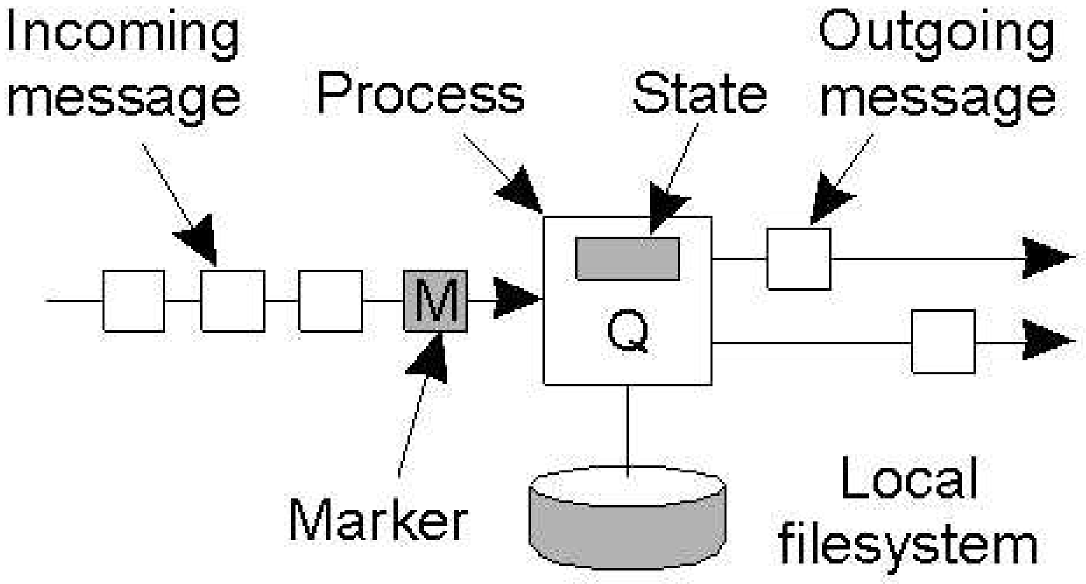
Snapshot algorithm

- Requires the implementation of two “rules” that together ensure a correct snapshot is taken:

- Marker-receiving rule for process P

- On P’s receipt of a marker message over channel C
  - If P has not yet recorded its state it:
    - Records its process state now
    - Records the state of C as the empty set
    - Turns on recording of messages that arrive over other incoming channels
  - Else
    - P records the state of C as the set of messages it has received over C since it saved its state
  - Endif

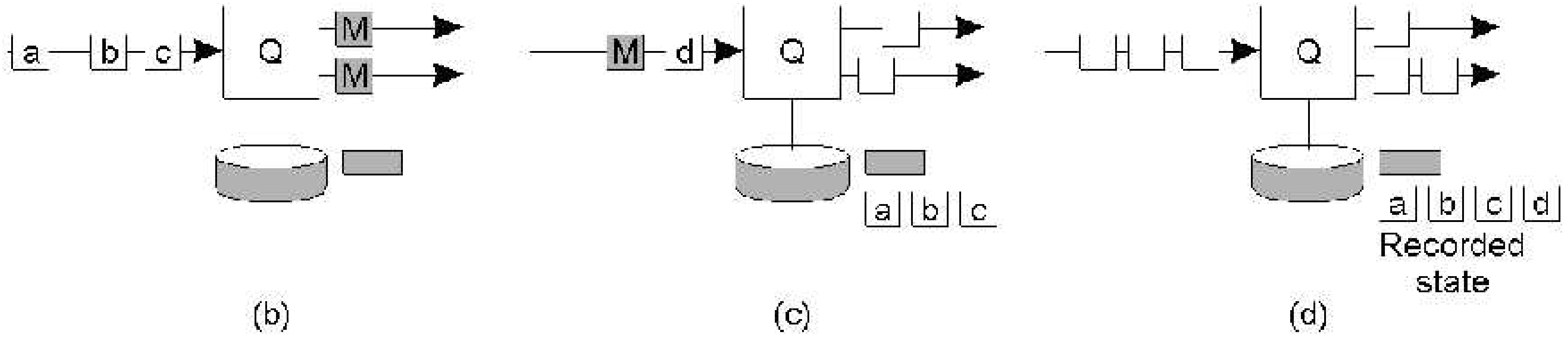
Snapshot algorithm



(a)

(a) Process Q receives a marker for the first time and records its local state

Snapshot algorithm



b) Q records its own state and sends a marker down all outgoing communication channels.

c) Q records all incoming messages.

d) Q receives a second marker on an incoming channel and finishes recording the state of that channel.

Termination of Snapshot

- We assume that:
  - a process that has received a marker message records its state within a finite time, and
  - Sends marker messages over each outgoing channel within a finite time.

- If there is a path of communication channels and processes from a process p_i to a process p_j ($j \neq i$), then it is clear the p_j will record its state a finite time after p_i recorded its state.

- Since we are assuming the graph of processes and channels to be strongly connected:
  - It follows that all processes will have recorded their states and the states of incoming channels a finite time after some process initially records its state.

To Do:

- Read Paper entitled:

  “Distributed Snapshots: Determining Global States of Distributed Systems”

  K. Mani Chandy and Leslie Lamport

  ACM Transactions on Computer Systems, Vol. 3, No. 1, February 1985, Pages 63-75.

Distributed Systems
Coordination: Mutual Exclusion

Mutual Exclusion

- It is often necessary to protect a shared resource within a Distributed System using “mutual exclusion”
  - For example, it might be necessary to ensure that no other process changes a shared resource while another process is working with it.

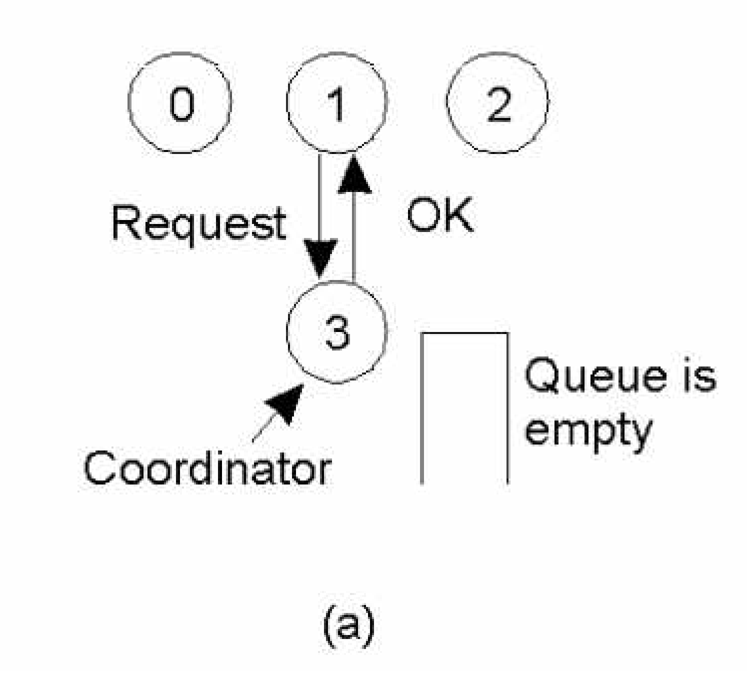
- In non-distributed, uniprocessor systems, we can implement “critical regions” using techniques such as semaphores, monitors and similar constructs – thus achieving mutual exclusion.

- These techniques have been adapted to Distributed Systems …

Mutual Exclusion: Techniques

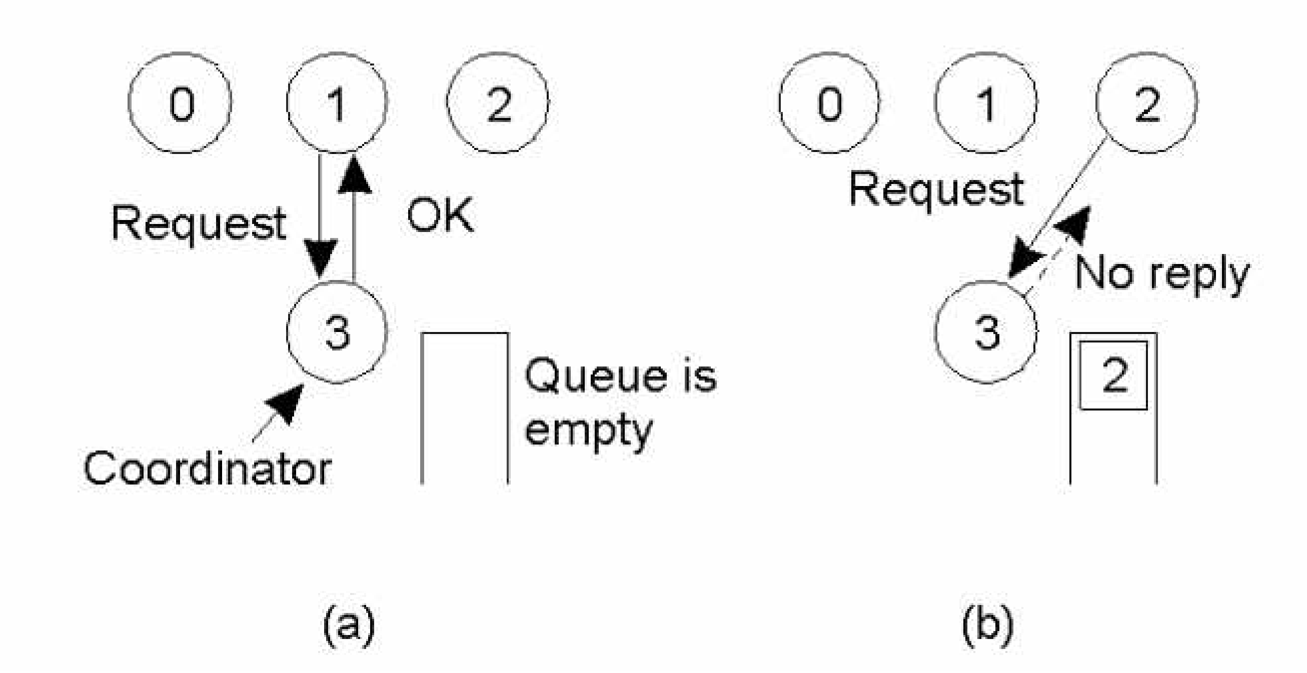
- In this course we will consider three techniques:
  - **Centralized**: a single coordinator controls whether a process can enter a critical region.
  - **Distributed**: the group confers to determine whether or not it is safe for a process to enter a critical region.
  - **Token Ring**: processes are organised into a logical loop and use a token to determine when to enter a critical region.
- We will look at each technique over the coming slides…

Centralized Algorithm



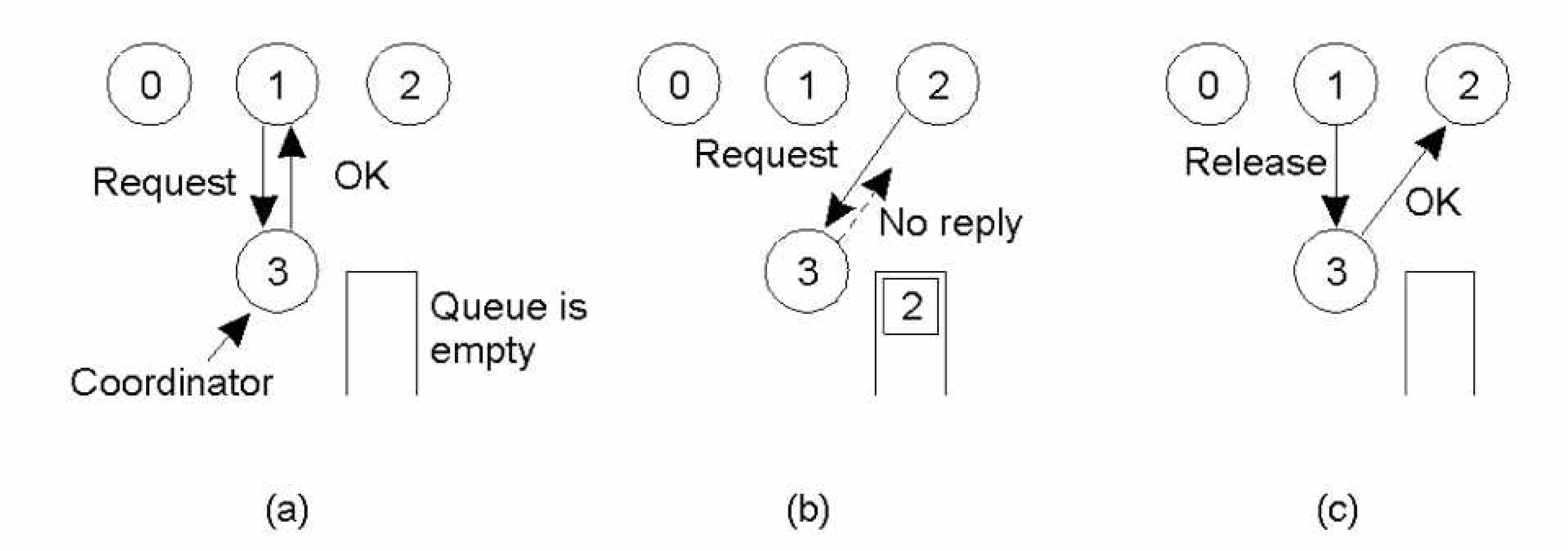
a)Process 1 asks the coordinator for permission to enter a critical region. Permission is granted by an OK message (assuming it is, of course, OK).

Centralized Algorithm



a)Process 2 then asks permission to enter the same critical region. The coordinator does not reply (but adds 2 to a queue of processes waiting to enter the critical region). No reply is interpreted as a “busy state” for the critical region.

Centralized Algorithm



a)When process 1 exits the critical region, it tells the coordinator, which then replies to 2 with an OK message.

The Centralized Algorithm

- Advantages:
  - It works.
  - It is fair.
  - There’s no process starvation.
  - Easy to implement.

- Disadvantages:
  - There’s a single point of failure!
  - The coordinator is a bottleneck on busy systems.

- **Critical Question**: When there is no reply, does this mean that the coordinator is “dead” or just busy?

Distributed Mutual Exclusion

- Based on work by Ricart and Agrawala (1981).

- Requirement of their solution: total ordering of all events in the distributed system (which is achievable with Lamport's timestamps).

- Note that messages in their system contain three pieces of information:
  - The critical region ID.
  - The requesting process ID.
  - The current time.

Distributed Algorithm

- When a process (the “requesting process”) decides to enter a critical region, a message is sent to all processes in the Distributed System (including itself).

- What happens at each process depends on the “state” of the critical region.
  - If not in the critical region (and not waiting to enter it), a process sends back an OK to the requesting process.
  - If in the critical region, a process will queue the request and send back no reply to the requesting process.

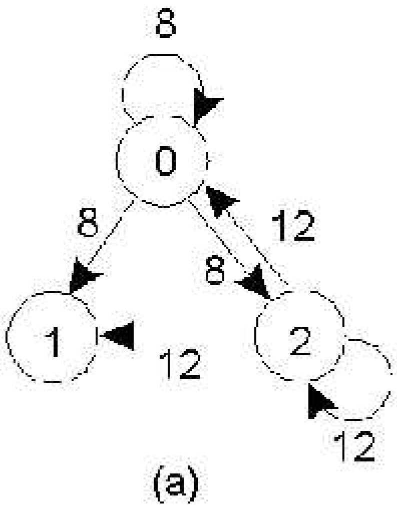
Distributed Algorithm

- If waiting to enter the critical region, a process will:
  - Compare the timestamp of the new message with that in its queue (note that the lowest timestamp wins).
  - If the received timestamp wins, an OK is sent back, otherwise the request is queued (and no reply is sent back).

- When all the processes send OK, the requesting process can safely enter the critical region.

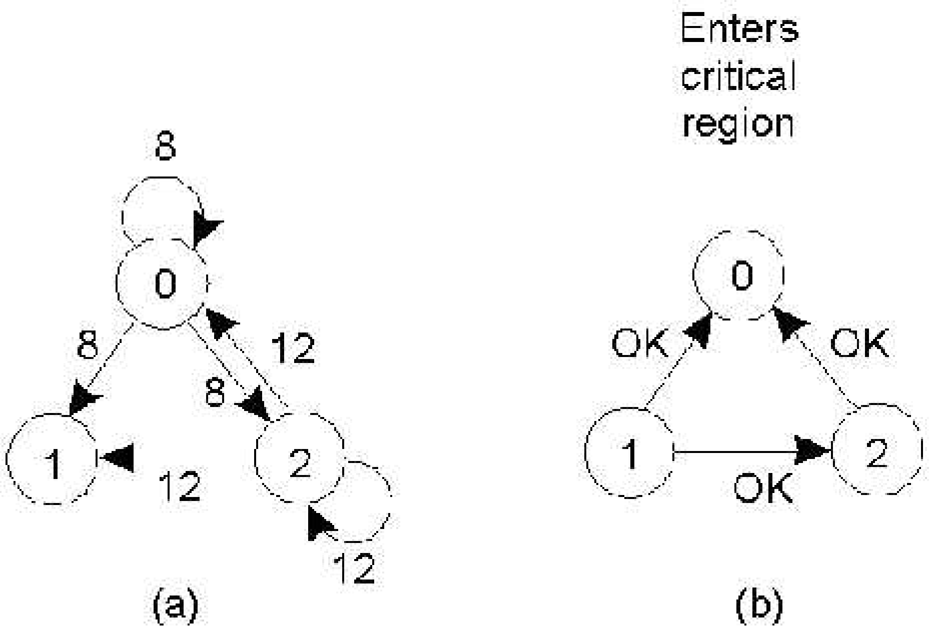
- When the requesting process leaves the critical region, it sends an OK to all the process in its queue, then empties its queue.

Distributed Algorithm Example



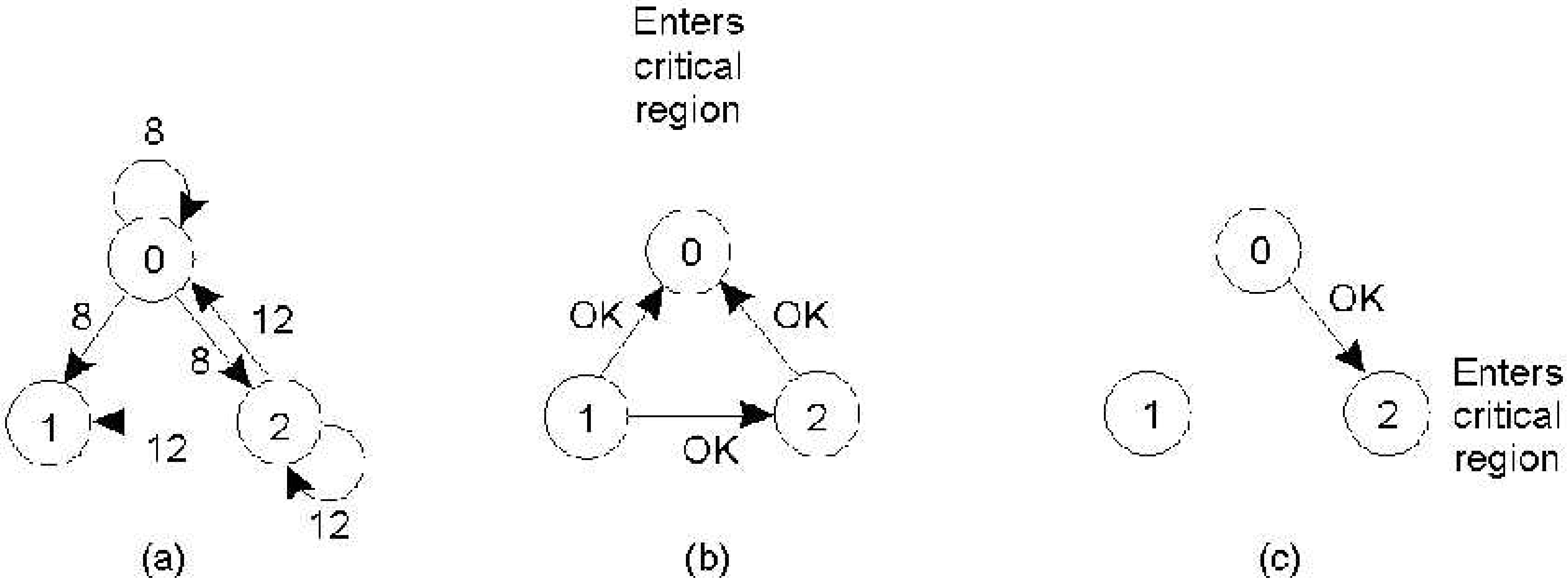
a) Processes 0 and 2 wish to enter the critical region “at the same time”.

Distributed Algorithm Example



b) Process 0 wins as it’s timestamp is lower than that of process 2.

Distributed Algorithm Example



c) When process 0 leaves the critical region, it sends an OK to 2.

The Distributed Algorithm

- The algorithm works because in the case of a conflict, the lowest timestamp wins as everyone agrees on the total ordering of the events in the distributed system.

- Advantages:
  - It works.
  - There is no single point of failure

- Disadvantages:
  - We now have multiple points of failure!!!
  - A “crash” is interpreted as a denial of entry to a critical region.
  - (A patch to the algorithm requires all messages to be ACKed).

The Distributed Algorithm

- Worse still - all processes must maintain a list of the current processes in the group (and this can be tricky)

- Worse still is that one overworked process in the system can become a bottleneck to the entire system – so, everyone slows down.

Which Just Goes To Show …

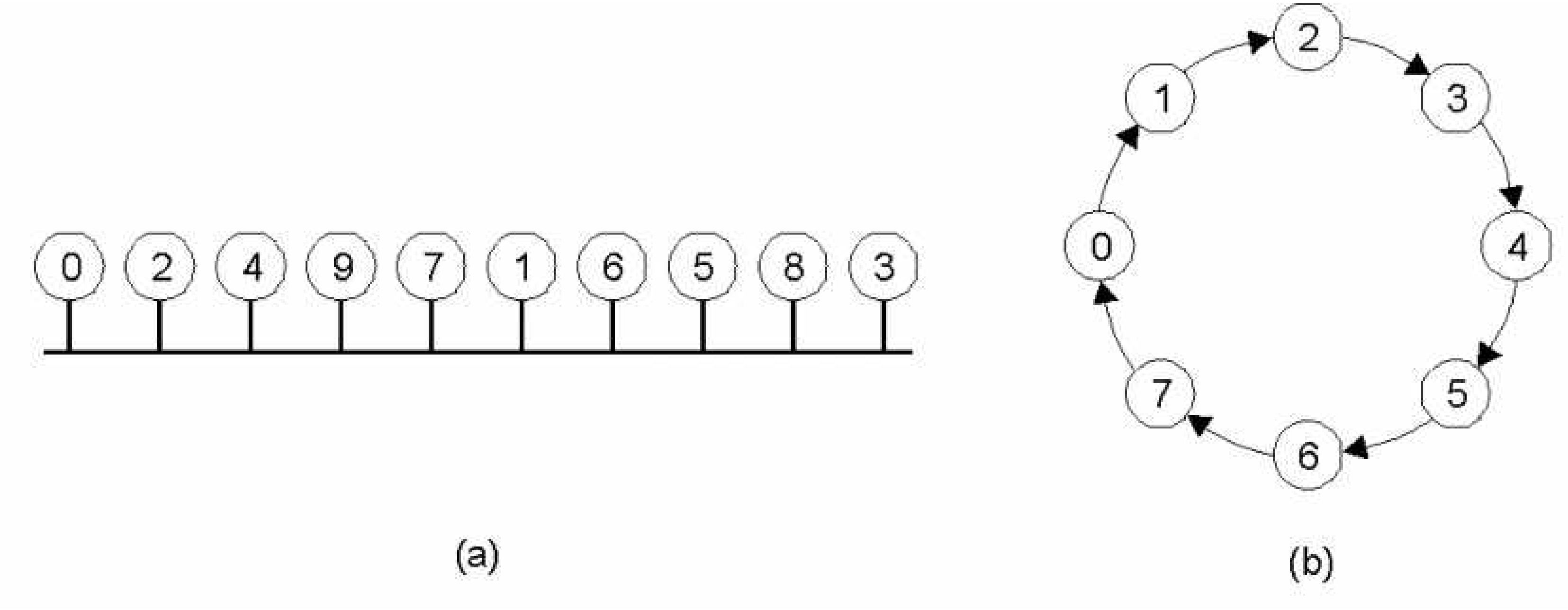
- It isn’t always best to implement a distributed algorithm when a reasonably good centralized solution exists.

- Also, what’s good in theory (or on paper) may not be so good in practice.

- Finally, think of all the message traffic this distributed algorithm is generating (especially with all those ACKs).

- Remember: every process is involved in the decision to enter the critical region, whether they have an interest in it or not. (Oh dear …)

Token-Ring Algorithm



a)An unordered group of processes on a network. Note that each process knows the process that is next in order on the ring after itself.

b)A logical ring is constructed in software, around which a token can circulate – a critical region can only be entered when the token in held. When the critical region is exited, the token is released.

Token-Ring Algorithm

- When the ring is initialized, process 0 is given a *token*.
- The token circulates around the ring.
- It is passed from process *k* to process *k+1.*
- When a process acquires the token from its neighbor it checks to see if it is attempting to enter a critical region.
- If so, the process enters the region, does all the work it needs to and leaves the region.
- Token is passed to the next process in the ring.

Token-Ring Algorithm

- Advantages:
  - It works (as there’s only one token, so mutual exclusion is guaranteed).
  - It’s fair – everyone gets a shot at grabbing the token at some stage.

- Disadvantages:
  - Lost token! How is the loss detected (it is in use or is it lost)? How is the token regenerated?
  - Process failure can cause problems – a broken ring!
  - Every process is required to maintain the current logical ring in memory – not easy.

Comparison: Mutual Exclusion

Algorithm	**Messages per entry/exit**	**Delay before entry (in message times)**	**Problems**
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token-Ring	1 to ∞	0 to n – 1	Lost token, process crash

- *None are perfect – they all have their problems!*

Comparison: Mutual Exclusion

- The “Centralized” algorithm is simple and efficient, but suffers from a single point-of-failure.
- The “Distributed” algorithm has nothing going for it – it is slow, complicated, inefficient of network bandwidth, and not very robust. It “sucks”!
- The “Token-Ring” algorithm suffers from the fact that it can sometimes take a long time to reenter a critical region having just exited it.
- All perform poorly when a process crashes, and they are all generally poorer technologies than their non-distributed counterparts.
  - Only in situations where crashes are very infrequent should any of these techniques be considered.

Election Algorithms

- Some Distributed Systems requires that one of the processes play a particular role.
  - For example, selecting a process to play the role of “central-server” in a variant of the Centralized Mutual Exclusion algorithm.

- In such situations, we need to employ a mechanism for selecting the “leader” process.
  - This mechanism must allow all relevant processes to participate in the choice.
  - It must also produce a single choice that is accepted by all the processes.
  - Once chosen the “leader” performs the assigned role until either they “retire” or fail.

- We term such a mechanism an **Election Algorithm**.