جامعة الجزيرة الخاصة
ALJAZEERA PRIVATE UNIVERSITY

Coordination & Agreement in Distributed Systems

Coordination & Agreement

- Multicast Communication
- Distributed Mutual Exclusion
- Election Algorithms

Introduction

- DS = N processes **&** Comm. by messages **&** No Shared memory.
- DS processes need to coordinate their actions or to agree on one value or more.
  - No fixed relation master / slave ➔ allows to avoid problems of process failure.
- Reliable connections can be obtained using reliable communication protocol although network failures.
- Detectors of process failure are implemented using timeouts.

Multicast Communication

Multicast Communication

- IP multicast allows to **implement** coordination & agreement between system processes.
- IP multicast **needs** coordination & agreement to be
  - Reliable
  - Ordered

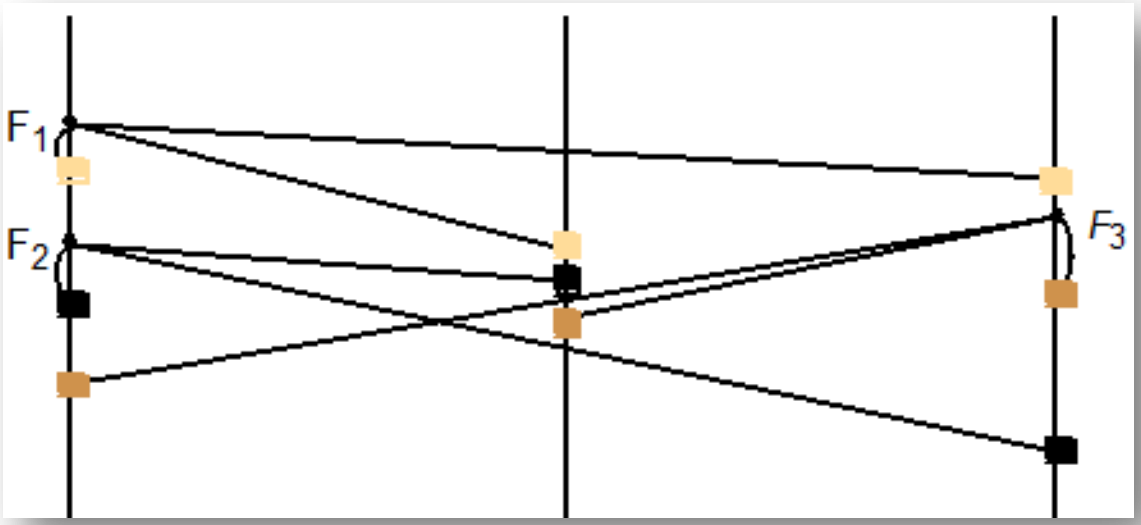
Reliable Multicast Communication

- if a process delivered a message before the multi-caster crashes.
- Then, all group processes will deliver this message.

Ordered Multicast Communication

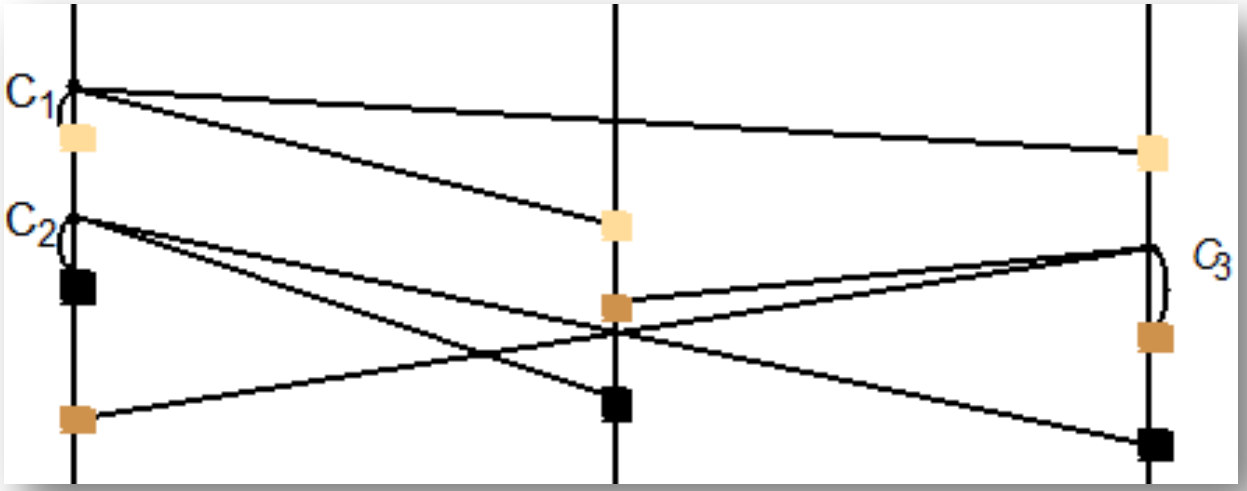
FIFO Ordering :

- process P multicasts ***m1*** and then ***m2***. Then, any group process will deliver ***m1*** before ***m2***. (= consistency)



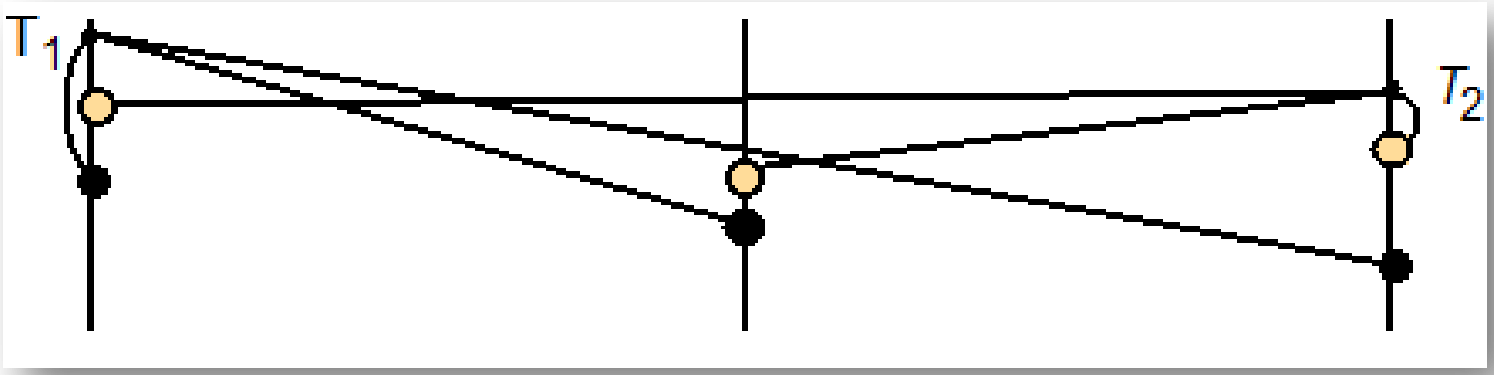
Ordered Multicast Communication

- **Causal Ordering :**
  - if multicast(***m1***) happened-before multicast(***m2***). Then, any group process will deliver ***m1*** before ***m2***.



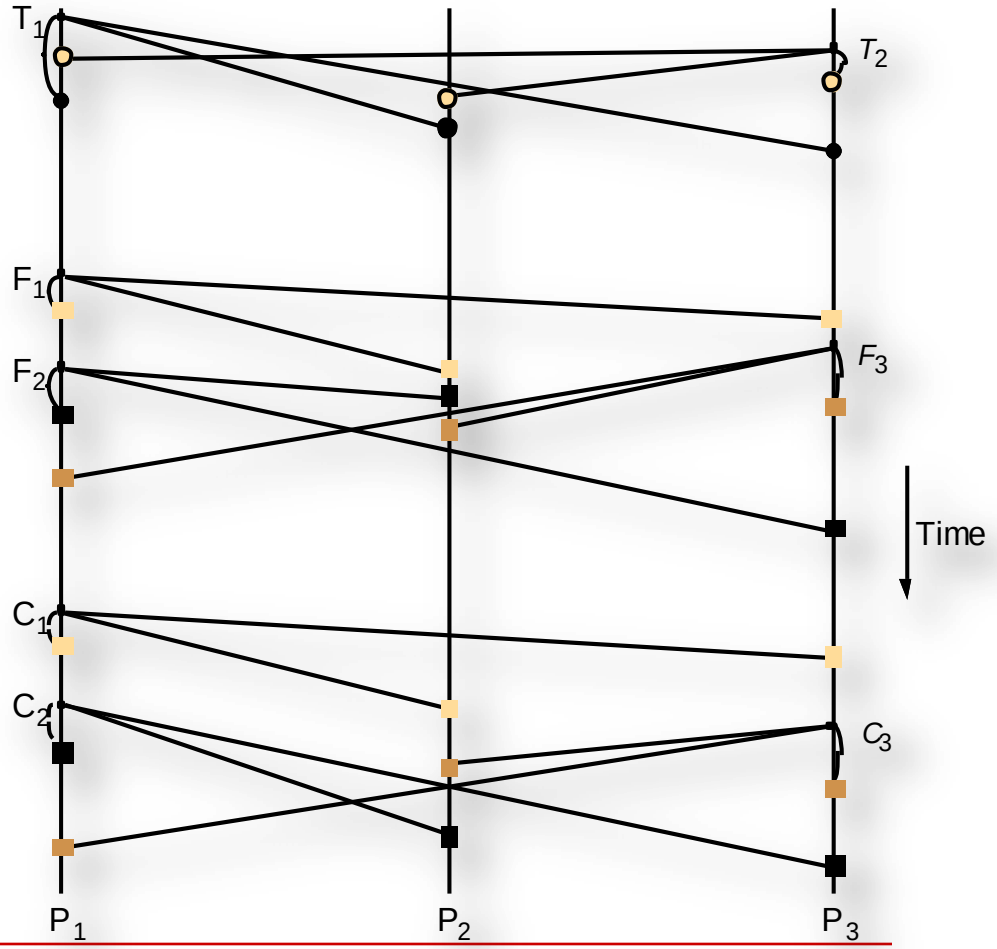
Ordered Multicast Communication

- **Total Ordering :**
  - if a process deliver ***m1*** before ***m2***. Then, any group process will deliver ***m1*** before ***m2***.

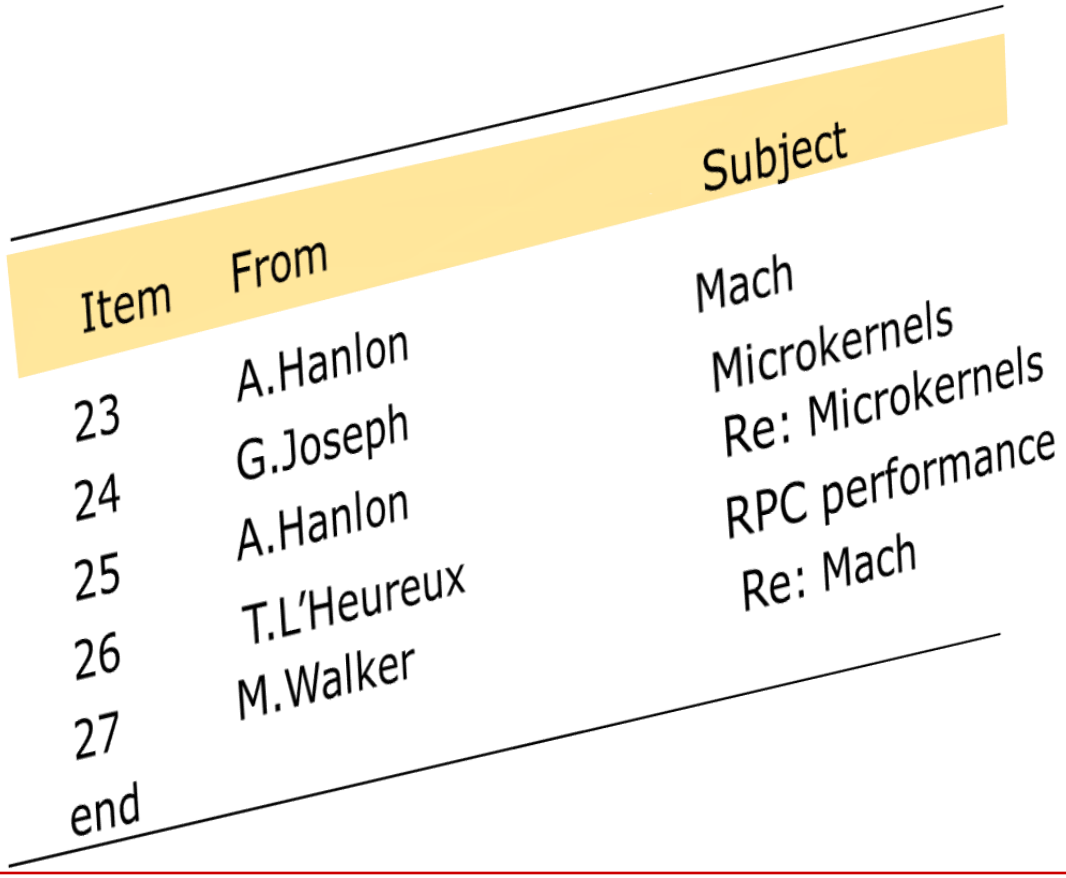


Multicast Communication (Ordering Ex)

- Totally ordered messages T_1 and T_2.
- FIFO-related messages F_1 and F_2
- Causally related messages C_1 and C_3 – and the otherwise arbitrary delivery ordering of messages.



Ordered Multicast Communication
(Ex : Chat Application)



Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

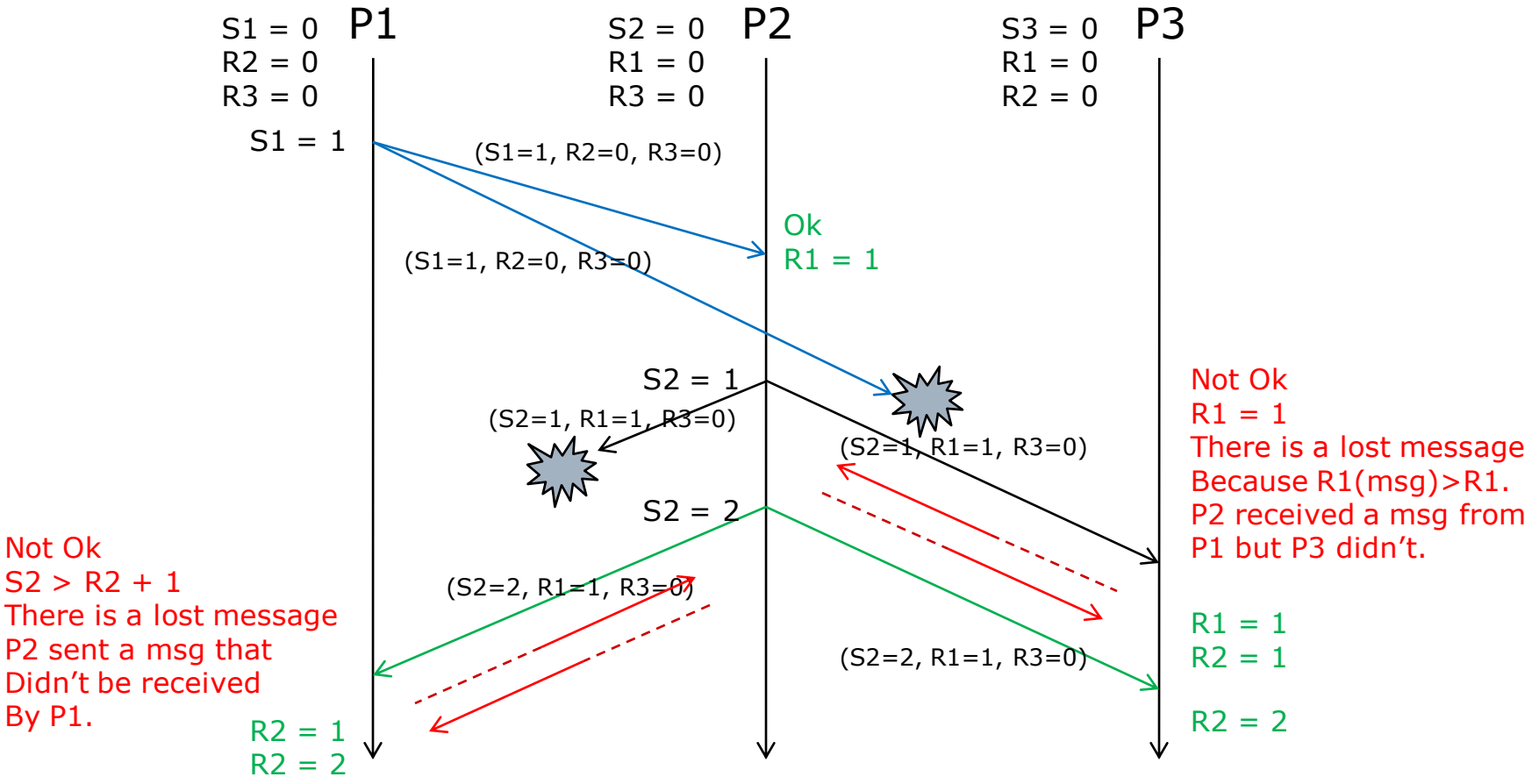
Reliable Multicast Communication

- Integrity (Safety) :
  - Correct process delivers a message **m** at most once.
- Validity (Liveness) :
  - If a correct process multi-casts **m**. Then, it will deliver **m**.
- Agreement (Atomicity) :
  - "All or Nothing" if correct process delivers message **m**. Then, all other correct processes in group will deliver it

Reliable Multicast Communication

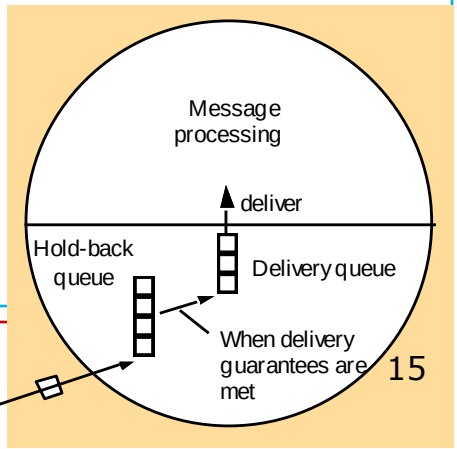
- Each Process P has several variables :
  - S_p = Nb messages sent by P
  - R_q = Nb messages sent by Q and delivered by P
- P multicast m ➔ S_p++ & multicast m, Sp, <q, R_q>, ..
- Q receives from P :
  - If S == R_p + 1 & R(@ msg) <= R_p ➔ R_p ++ & deliver m
  - If S <= R_p ➔ nothing, m has been delivered (duplication)
  - If S > R_p + 1 || R > R_p ➔
    - Wait for the missed messages from P
    - Demand the missed messages from P or from the original Process.

Reliable Multicast Example

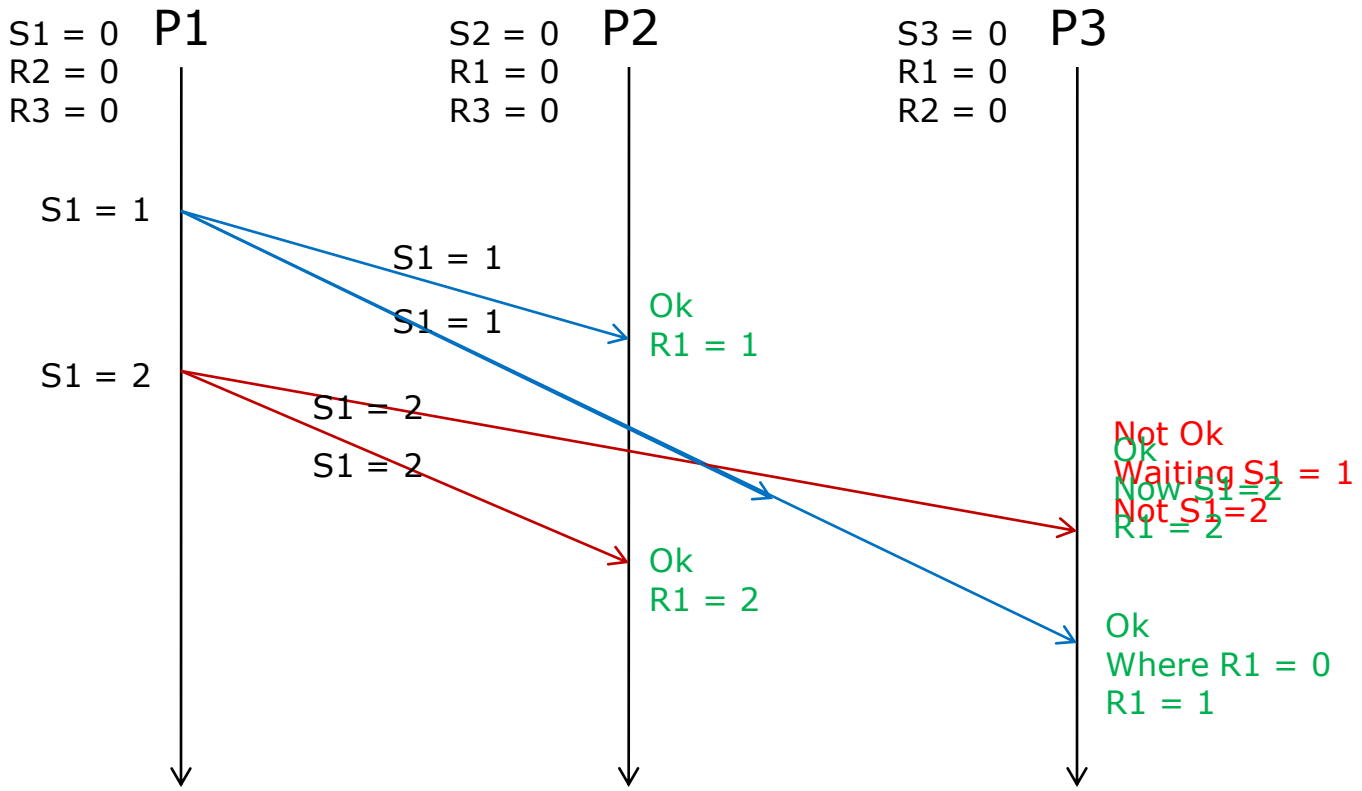


FIFO Ordering Multicast

- Each process P has 2 variables :
  - S_p = Nb of messages sent by ***P***
  - R_q = Nb of messages delivered by ***p*** from ***q***
- P multicasts message. So, S_p++ and message is sent with S_p
- P receives message from q ➔
  - If S == R_q + 1. Then, delivery the message & Rq++
  - If S > R_q + 1. Then, put the message in the hold-back until the reception of missed messages

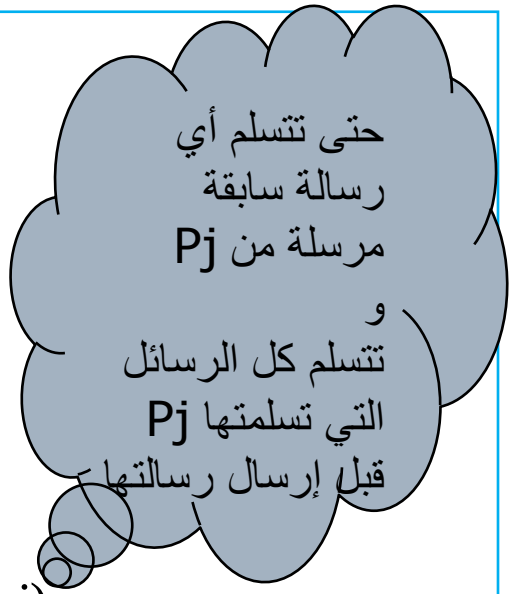


FIFO Multicast Example

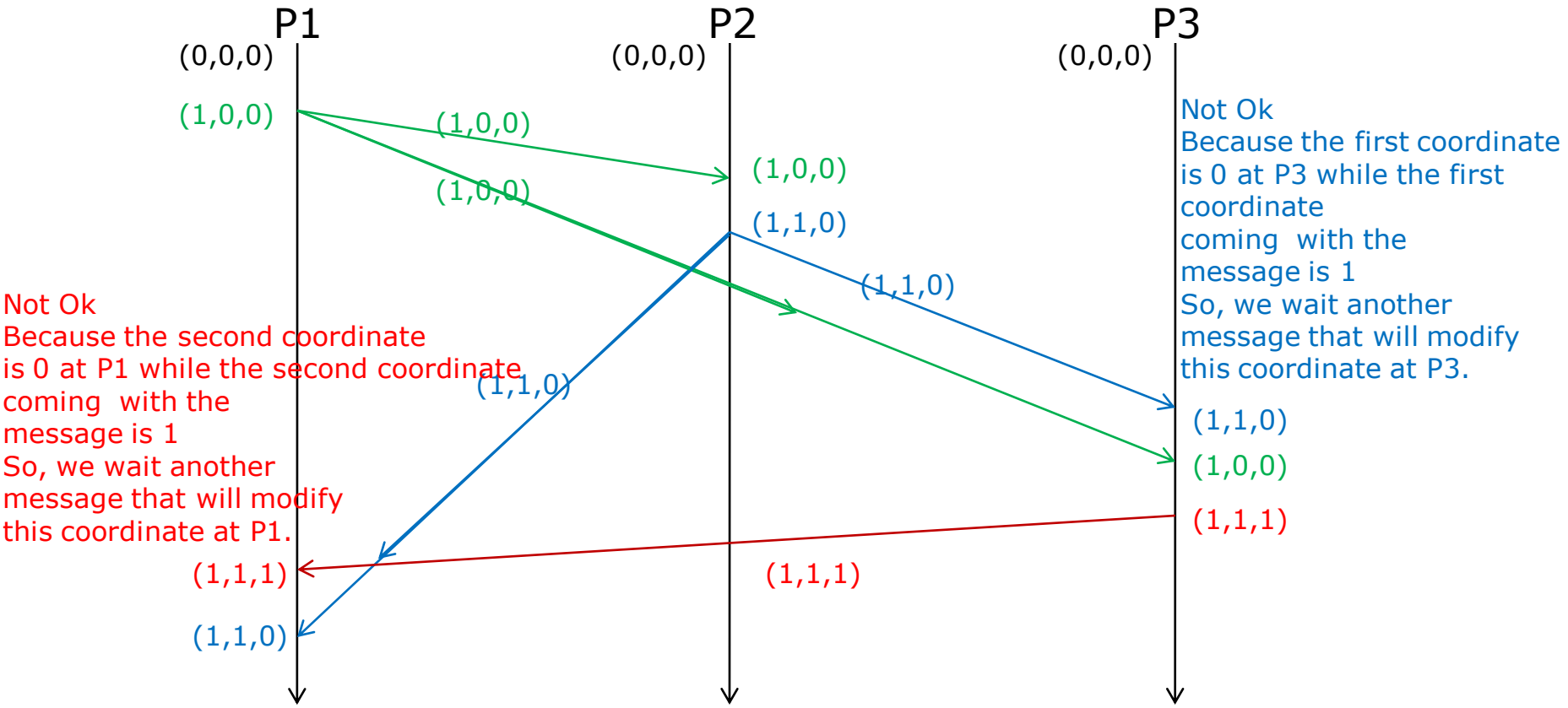


Causal Ordering Multicast

- Initialization V[j] = 0 (j= 1,2, ….N).
- Pi will multicast message m
  - Vi[i] = Vi[i] + 1
  - Multicast (Vi, m)
- Pi receives message m from Pj
  - Put <Vj, m> in hold-back queue
  - Until Vj[j] = Vi[j] + 1 **&** Vj[k] <= Vi[k] (k != j)
  - Deliver m
  - Vi[j] = Vi[j] + 1



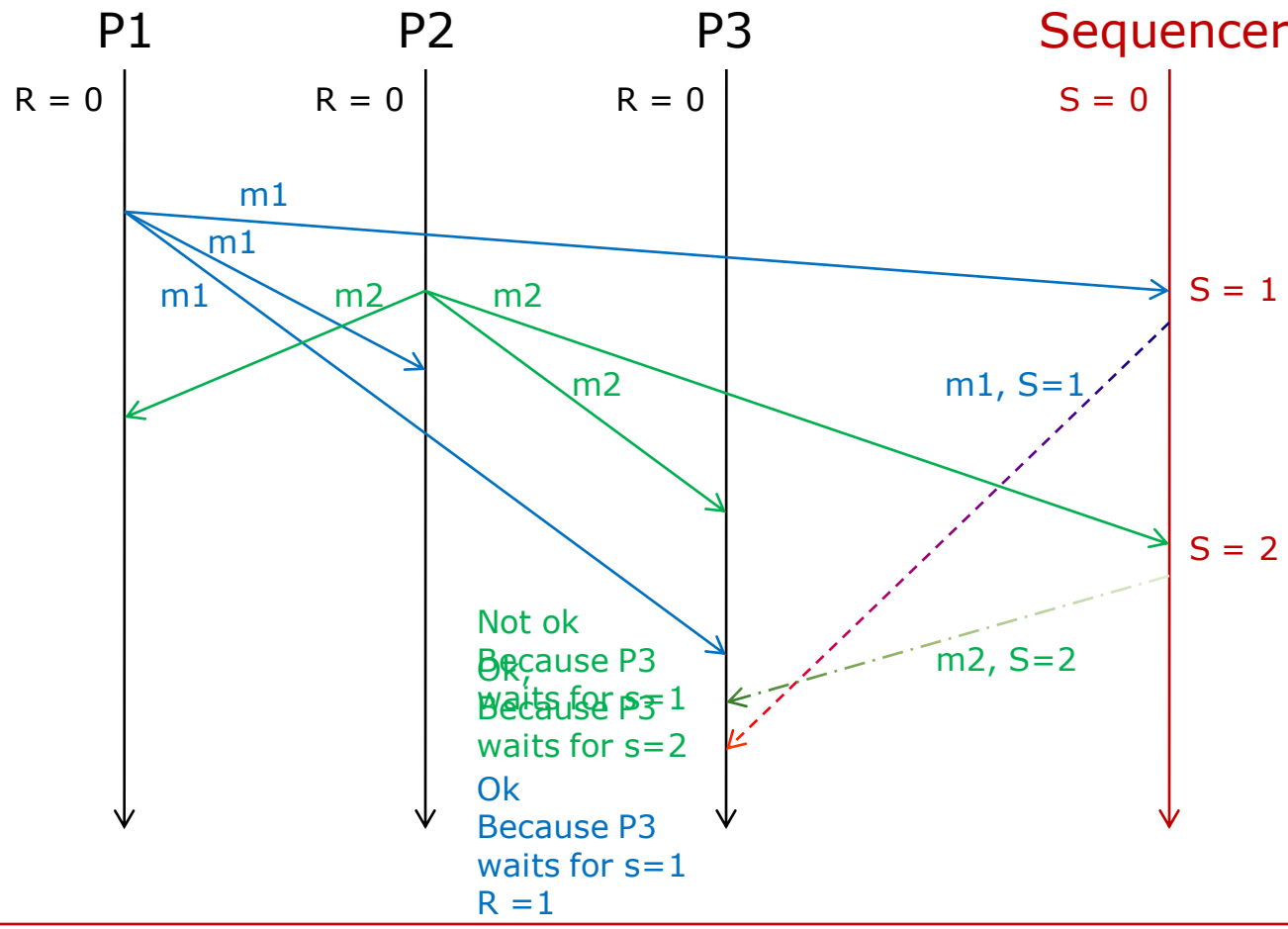
Causal Multicast Example



Total Ordering Multicast (Sequencer)

- Sequencer is a process that holds a variable S.
- Any process P holds variable R.
- Process P multicasts message <m, i> to all processes + sequencer
- P receives <m, i> ➔ put <m, i> in hold-back queue
- Sequencer receives <m, i>
  - S = S +1
  - Multicast <'order', i, S>
- P receives <'order', i, S>
  - Wait until <m, i> in hold-back & S == R + 1
  - Deliver m
  - R = S

Total Multicast Example



Total Ordering Multicast (Global agreement)

- Aq = largest observed agreed sequence.
- Pq = own largest proposed sequence Nb.

- Process P multicasts message <m, i>
- All processes Q send propositions to P
  - Pq = max(Aq, Pq) + 1
- P collects the propositions and selects the largest Nb = a.
  - Multicasts (i, a)
- All processes Q modify Aq = max (Aq, a)

Distributed Mutual Exclusion

Distributed Mutual Exclusion

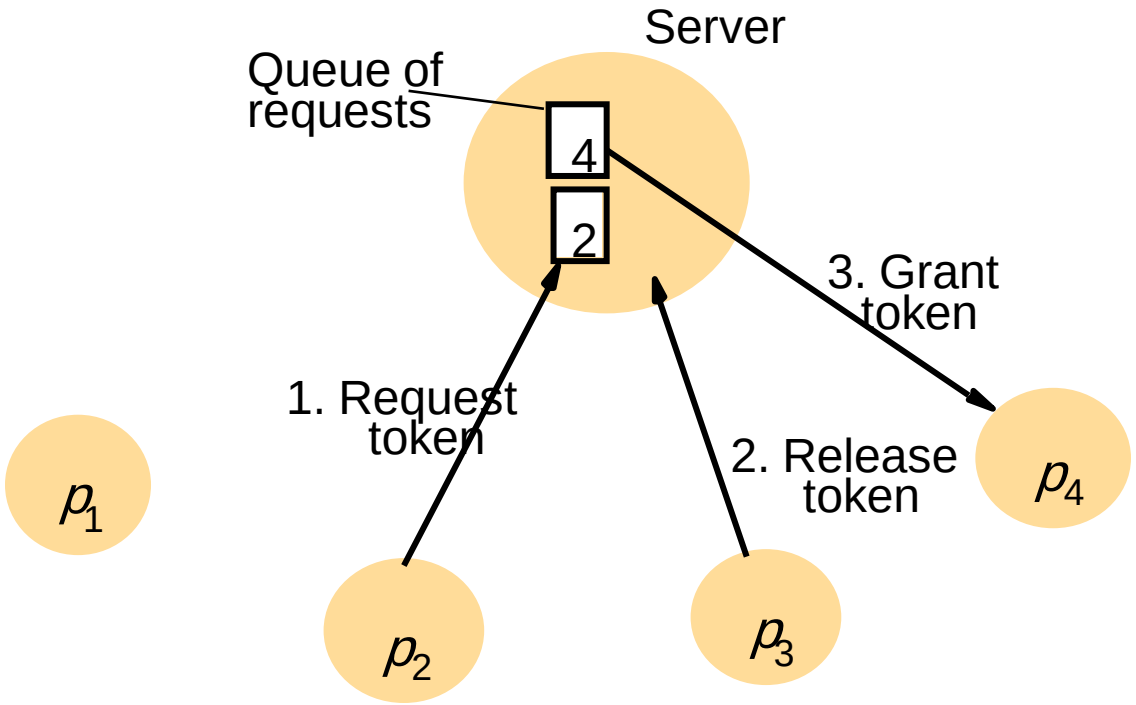
- Processes need to coordinate their actions.
  - Processes may access shared data structure.
- **Critical Sections** are used to avoid the interference (keep shared data consistent) = one process can execute the critical section at any given time.
  - Operations = Enter(), Exit(), AccessResource()
- No Shared memory & No kernel facilities in Distributed Systems. So, **Semaphores** cannot be used
- **Messages passing** still the sole solution to realize distributed mutual exclusion.

Mutual Exclusion Requirement

- **Safety** : At most one process may execute the critical section at a time
- **Liveness** : Requests to enter / exit critical section eventually succeed.
  - No deadlock & No starvation
- **Ordering** : Request 1 to enter the critical section happened-before Request 2. Then, Request 1 enters the CS before Request 2.
- **Performance evaluation** :
  - Bandwidth & delay of client.

Centralized Algorithm

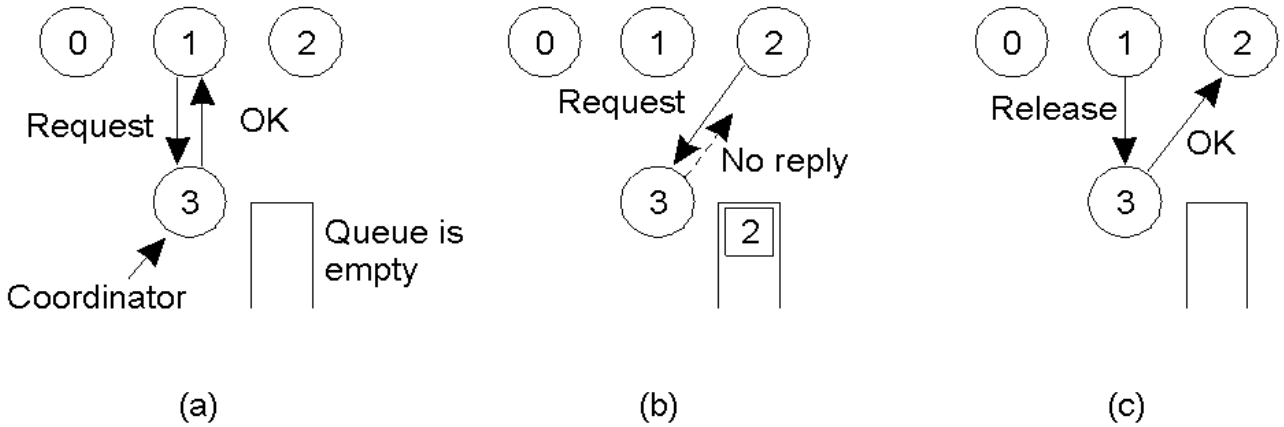
- One server process (coordinator) grants accesses to the critical section



Centralized Algorithm

- **Enter() = Request Message =** Process asks the coordinator to enter a critical section.
  - If no process is in the critical section. Then, coordinator accords the permission & replies by **(Grant Message)**.
  - Else = (another process is in the critical section) the coordinator does not reply and **queues** the request.
- **Exit() = Release Message** is sent to the coordinator by the process that exits from the critical section.
  - The coordinator removes & sends **(Grant Message)** to the first process in the queue.

Centralized Algorithm



a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

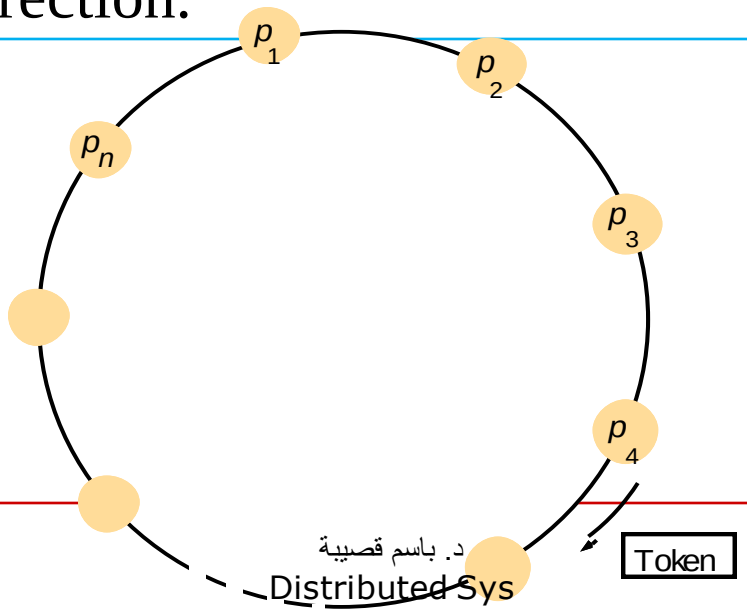
c) When process 1 exits the critical region, it tells the coordinator, then, it replies to 2

Centralized Algorithm

- **Results :**
  - Safety = Ok. At most one process in the critical section
  - Liveness = No process waits for ever.
  - Ordering = ?
  - Performance evaluation =
    - Bandwidth used to enter/exit = 3 messages
    - Delay of client = 2 messages
    - Delay of synchronization = 2 messages
- **Problems :**
  - Coordinator = point of failure.
  - Coordinator = bottleneck on busy systems.

Token Ring Algorithm

- No need for an additional process.
- Arrange processes in a logical ring.
- p_i has a communication channel to $p_{(i+1) \bmod (n+1)}$.
- Token may be passed by messages from p_i to $p_{(i+1) \bmod (n+1)}$ in a single direction.



Token Ring Algorithm

- Only the process that holds the token can enter the critical section.
  - **Enter()** = process waits for the token.
  - **Exit()** = process sends the token onto its neighbor
- If a process does not want to enter the CS when it receives the token, it forwards the token to the next neighbor.

Token Ring Algorithm

- **Results :**
  - Safety = Ok. Only one process holds the token at any instant. Then, only one process can enter the critical section.
  - Liveness = Every process will have the token at some instant.
  - Ordering = ?
  - Performance evaluation =
    - Bandwidth used to enter/exit = 1 → ∞ messages
    - Delay of client = 0 → (N-1) messages
    - Synchronization delay = 1 to N messages
- **Problems :**
  - Process failure :
    - Lose of the token (acknowledgement of token receipt)
    - Broken ring (……)

Algorithm of multicast & logical clocks

- **Enter()** = process multicasts a request message. It enters the critical section if all other processes have replied (positively).
- **Exit()** = reply to all queued requests.
- Message form = <T, P_i>
  - T = Sender's timestamp
  - Pi = Sender identity
- Process states :
  - **RELEASED** = outside the critical section
  - **WANTED** = wants to enter the critical section
  - **HELD** = inside the critical section

Algorithm of multicast & logical clocks

- When a process receives a request for a critical section. It answers according to its state :
  - RELEASED : It replies immediately & positively
  - HELD : No reply & it queues the request
  - WANTED : It compares the timestamps of the request and that of its request
    - Its req timestamp < the request's timestamp ➔ No reply & it queues the request
    - Its req timestamp > the request's timestamp ➔ It replies positively
- When a process leaves the critical section. It replies positively all queued requests

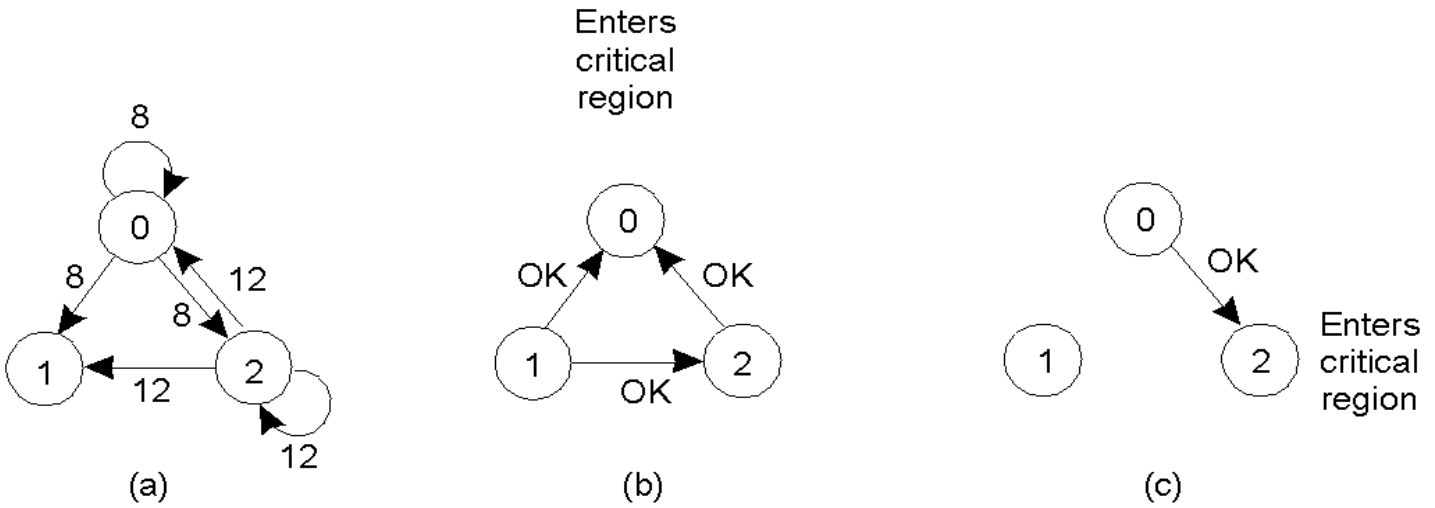
Algorithm of multicast & logical clocks

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    T := request's timestamp;
    Multicast *request* to all processes;
    Wait until (number of replies received = (*N* – 1));
    state := HELD;

On receipt of a request $<T_i, p_i>$ at p_j ($i \neq j$)
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if

To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Algorithm of multicast & logical clocks



a) Two processes want to enter the same critical region at the same moment.

b) Process 0 has the lowest timestamp, so it wins.

c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

Algorithm of multicast & logical clocks

- **Results :**
  - Safety & Liveness = Ok. Processes use timestamps to agree about one decision in case of conflict.
  - Ordering = processes enter the critical section according to request's timestamps.
  - Performance evaluation =
    - Bandwidth used to enter/exit = 2 (N-1) messages
    - Delay of client = N message (2 message times)
    - Synchronization delay = 1 message
- **Problems :**
  - N points of failure. Silence is interpreted (incorrectly) as denial of permission
    - Solution = using replies & timeouts

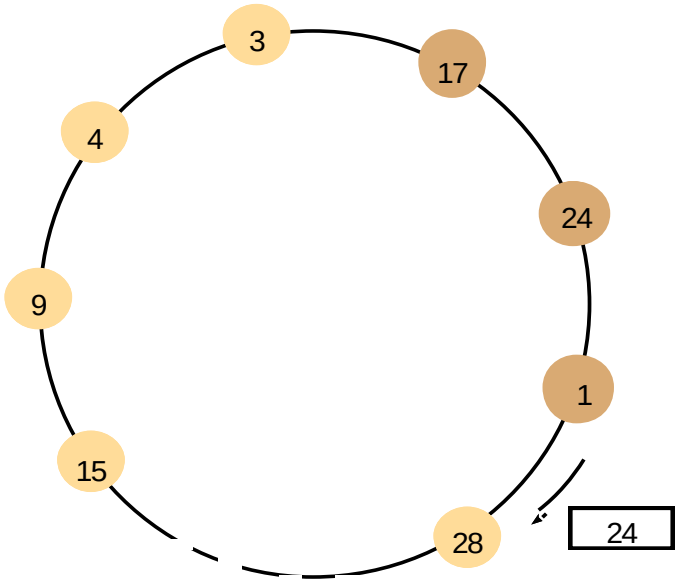
Election Algorithms

Election Algorithms

- Some processes can call the election when they detect the coordinator crash.
- **Goal** = agree on an unique coordinator among N processes.
  - Selection condition = process whose the largest ID (Ex: the network address)
- Participant processes = engaged in some run of the election
- Requirements :
  - **Safety** = Participant process has $elected_i = \perp$ or P. where P = largest ID non-crashed.
  - **Liveness** = all processes P_i participate eventually set $elected_i \neq \perp$ or crash

Ring-based Election Algorithm

- Arrange processes in a logical ring.
- p_i has a communication channel to $p_{(i+1) \bmod (n+1)}$.
- Token may be passed by messages from p_i to $p_{(i+1) \bmod (n+1)}$ in a single direction.



Ring-based Election Algorithm

- Initially, all processes are marked non-participant
- Any process can call the election
  - It is marked participant
  - It passes its ID as token <***election*** msg> to its neighbor
- Process receives the election msg
  - If msg ID > its ID ➔ forward the msg to neighbor & process is marked participant
  - If msg ID < its ID && process is non-participant ➔ forward its ID & process is marked participant
  - If msg ID < its ID && process is participant ➔ no forward (<u>useful in case of conflicts</u>)
  - If msg ID == its ID ➔ this process is the new coordinator & process is marked non-participant & sends ***elected*** msg to its neighbor

Ring-based Election Algorithm

- When process receives the ***elected*** msg
  - It is marked as non-participant
  - It sets ***elected*** variable to the msg ID
  - If the process is not the new coordinator ➔ it forwards the ***elected*** msg to neighbor
    - Else ➔ No forward

Ring-based Election Algorithm

- Results :
  - Safety = all IDs compared
  - Liveness = no two elections simultaneously allowed
  - Bandwidth used in the worst case = (3N -1) msg
    - Worst case = the largest ID process is the previous process to that calls the election
    - (N – 1) msg to arrive to the new coordinator
    - N msg to ensure that the new coordinator has the largest ID
    - N elected msg
- Problems :
  - No tolerance to failures.

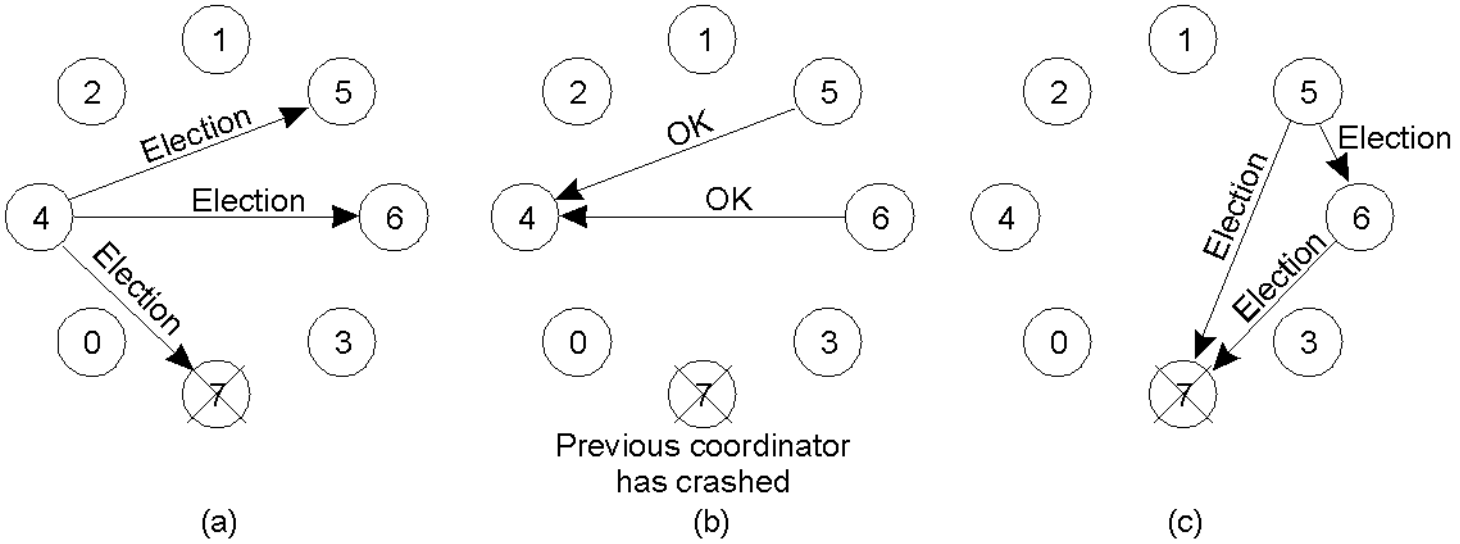
Bully Election Algorithm

- This algorithm allows processes to crash during elections.
- This algorithm uses timeouts to detect process failure.
  - Several processes can detect concurrently the failure of coordinator
- Process knows the processes of higher ID.
- Process of highest ID can directly elect itself.
- Message types :
  - *Election* = announce new election
  - *Answer* = tell the sender to stop
  - *Coordinator* = announce the new coordinator ID

Bully Election Algorithm

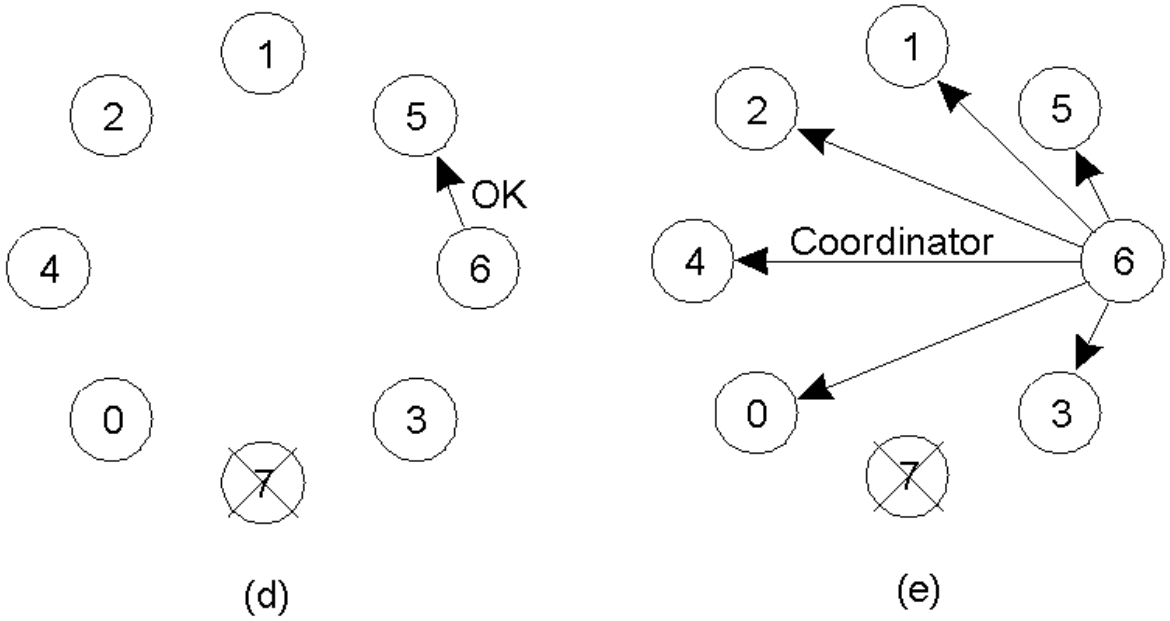
- When a process detects the coordinator failure.
  - It sends an ***election*** msg to processes of higher ID.
  - It waits answers for T
    - No answers arrive. Then, it is the new coordinator & sends ***coordinator*** msg to the processes of the lower ID.
    - Otherwise (answers arrive). Then, it waits T′ for ***coordinator*** msg.
      - No ***coordinator*** msg arrive. Then, it re-sends ***election*** msg
- When a process receives a ***coordinator*** msg. Then, it sets its *elected* variable to the new coordinator ID.
- When a process receives an ***election*** msg. Then, it replies by ***answer*** msg & it sends (if did not yet do) ***election*** msg to processes of higher ID

Bully Algorithm / Example



A. Process 4 holds an election
B. Process 5 and 6 respond, telling 4 to stop
C. Now 5 and 6 each hold an election

Bully Algorithm / Example



d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone.

Bully Algorithm / (Example with crash)

Election of

P2

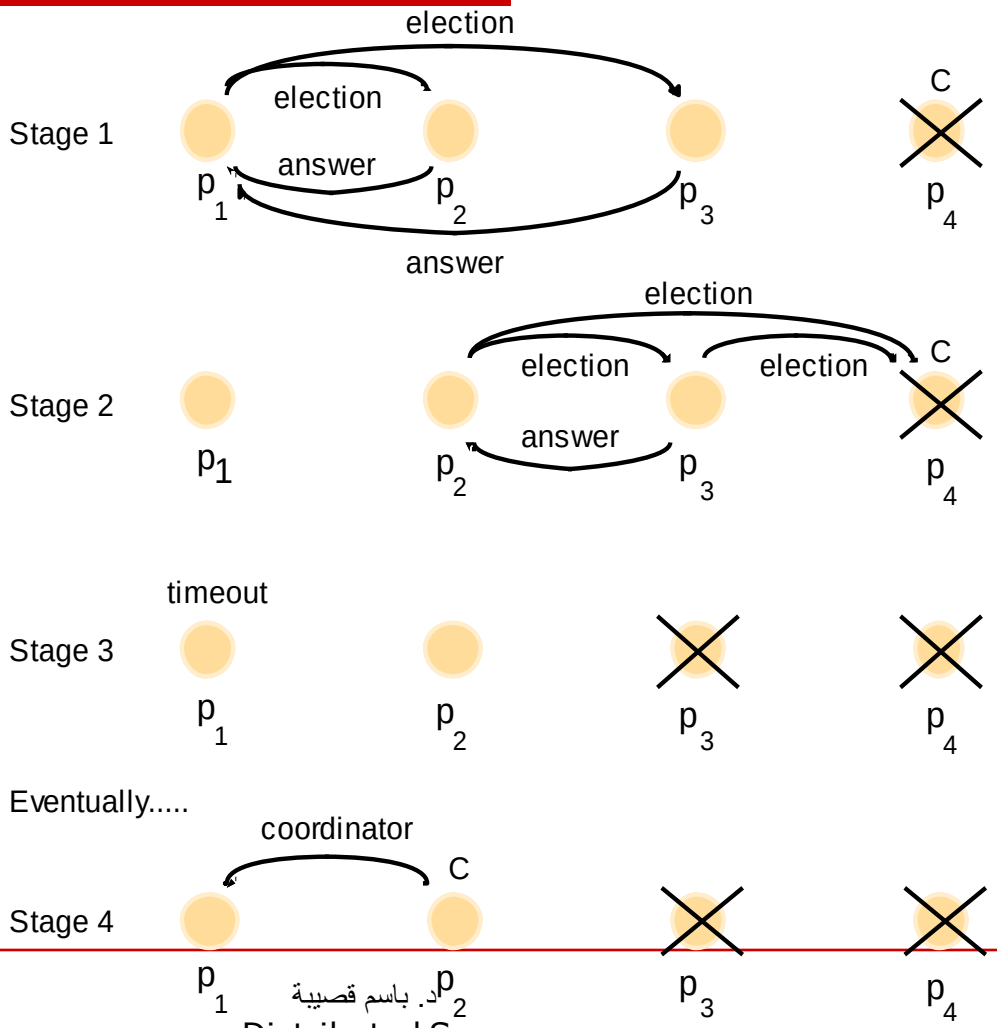
after failure

of

P4

and then

P3



Bully Election Algorithm

- Results :
  - **Safety** = all IDs compared.
    - But if timeout turns out. So, not all IDs compared
  - **Liveness** = No two processes can be coordinator because the lower ID process stop when it detects larger ID processes.
  - Bandwidth used :
    - Best case = the highest ID process detect the coordinator failure and sends N-2 coordinator messages
    - Worst case = the lowest ID process detect the coordinator failure and sends N-1 election messages and each process repeats that. So, $O(N^2)$
- Problems :
  - Need reliable message delivery & reliable failure detector.

Questions ?