Distributed Systems

a.y. 2023/2024

Distributed Systems:

Coordination

Coordination and agreement

- Action coordination
  - Mutual exclusion

- Agreement on shared values
  - Leader election

Coordination algorithms

- for resource sharing: concurrent updates of
  – records in a database (record locking)
  – files (file locks in stateless file servers)
  – a shared bulletin board

- to agree on actions: whether to
  – commit/abort database transaction
  – agree on a readings from a group of sensors

- to dynamically re-assign the role of master
  – choose primary time server after crash
  – choose co-ordinator after network reconfiguration

- Centralized solutions are not appropriate
  – communications bottleneck
- Fixed master-slave arrangements are not appropriate
  – processes crash
- Varying network topologies
  – ring, tree, …. connectivity problems

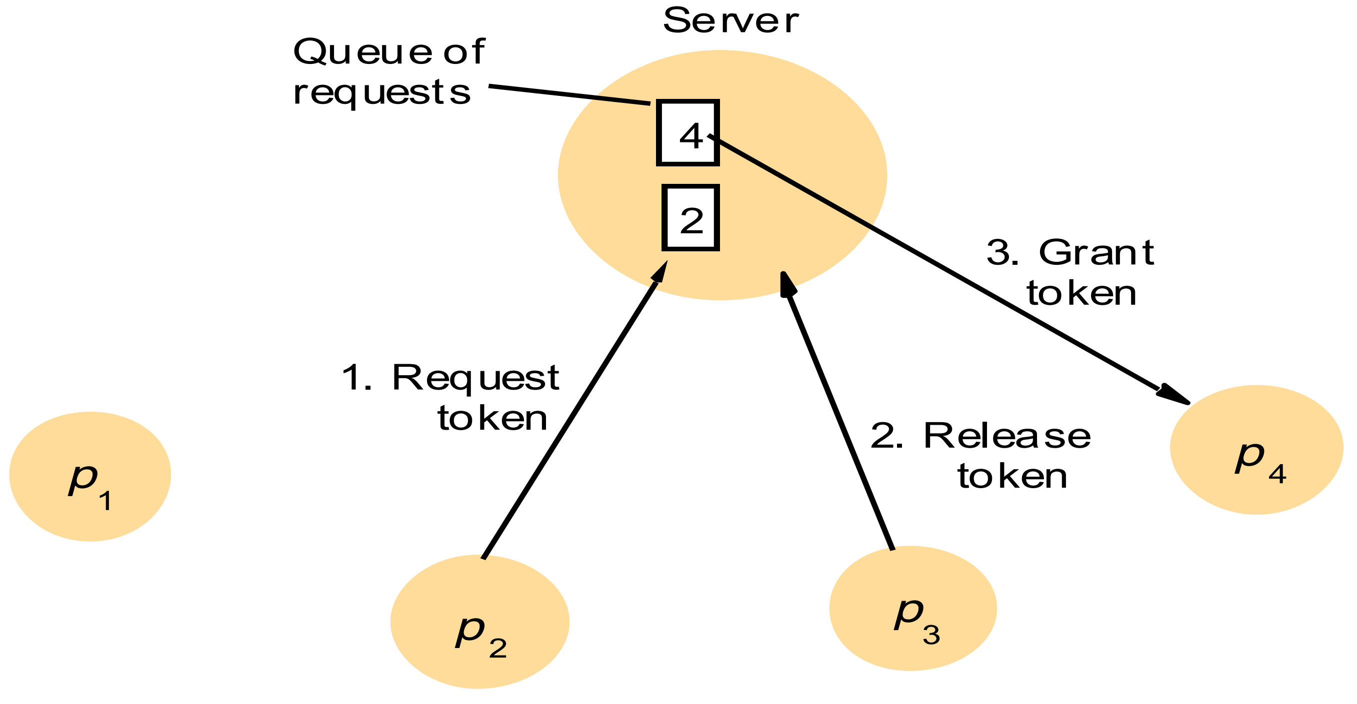
Requirements

(**MC1**) At most one process is in CS at the same time.

(**MC2**) Requests to **enter** and **exit** are eventually granted (no deadlock, no starvation).

(**MC3** - Optional, stronger) Requests to **enter** granted according to causality order.

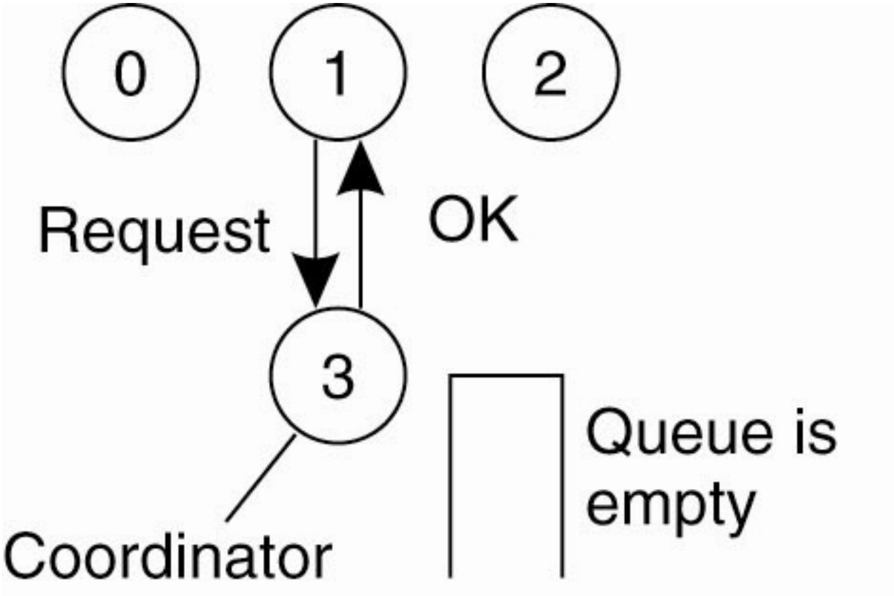
Centralized mutual exclusion



Centralized service

- Single server implements an imaginary token:
  – only the process holding the token can be in CS
  – server receives **request** for token
  – replies **grant**ing access if CS free; otherwise, the request is queued
  – when a process **release**s the token, the oldest request from the queue is granted
- it does not respect causality order of requests (MC3)

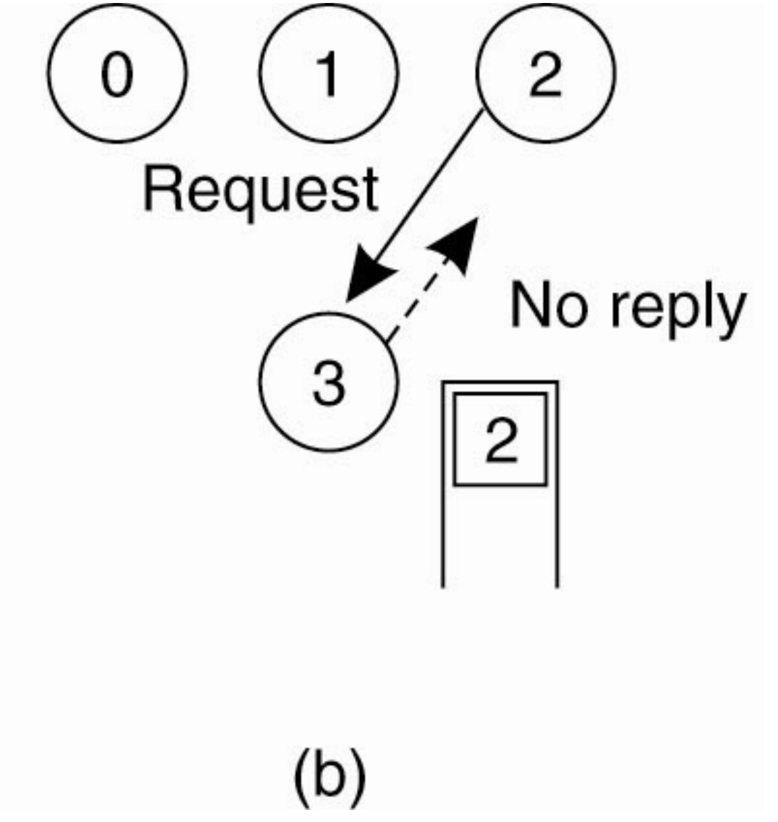
Mutual Exclusion a Centralized Algorithm



(a)

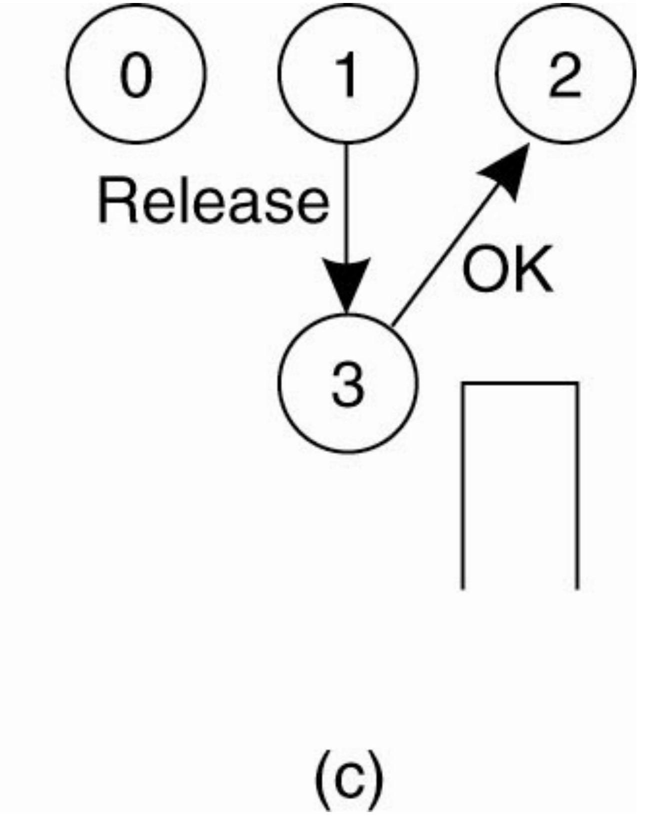
- Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

Mutual Exclusion a Centralized Algorithm



- Process 2 then asks permission to access the same resource. The coordinator does not reply.

Mutual Exclusion a Centralized Algorithm



- When process 1 releases the resource, it tells the coordinator, which then replies to 2.

Distributed mutual exclusion

• peer processes

• Working hypthesis:

– N asynchronous processes, for simplicity no failures
– guaranteed message delivery (reliable links)
– to execute critical section (CS), each process calls:

  - **enter**()
  - **resourceAccess**()
  - **exit**()

A Distributed Algorithm

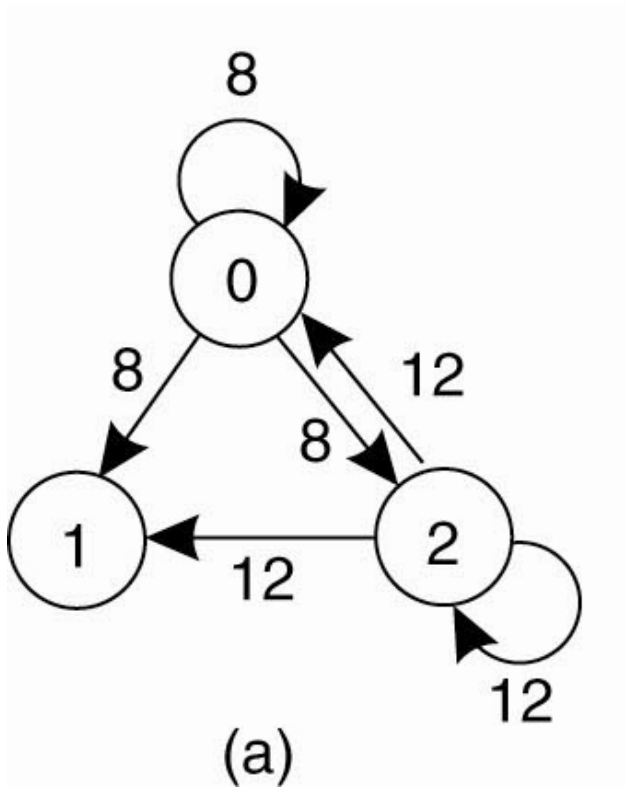
1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.

2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.

A Distributed Algorithm

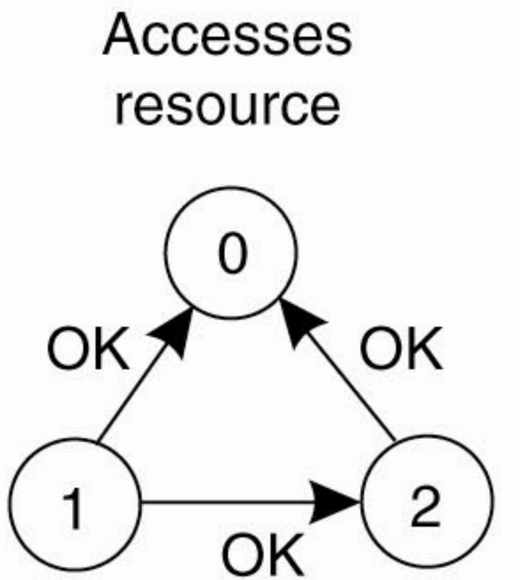
3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.

A Distributed Algorithm



- Two processes want to access a shared resource at the same moment.

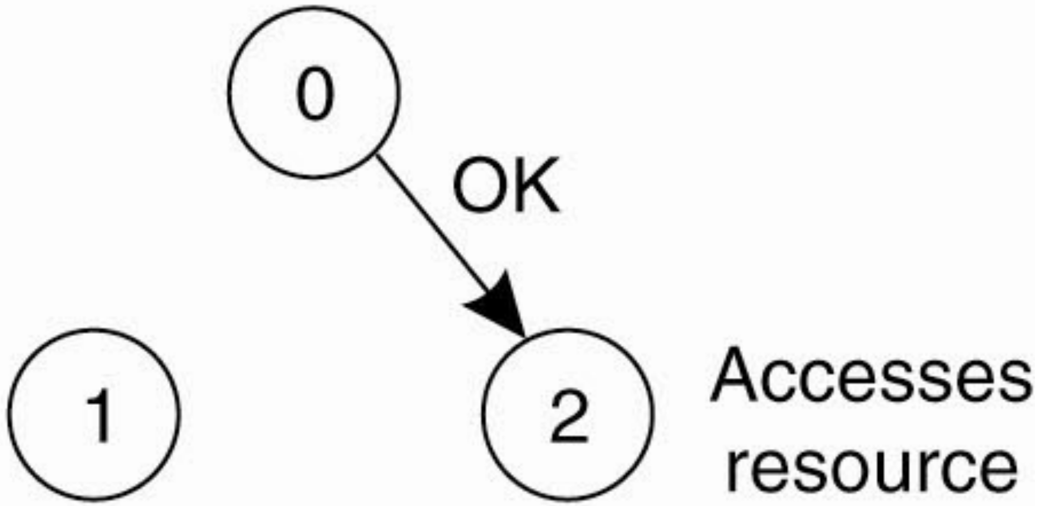
A Distributed Algorithm



(b)

- Process 0 has the lowest timestamp, so it wins.

A Distributed Algorithm



(c)

- When process 0 is done, it sends an OK also, so 2 can now go ahead.

Ricart&Agrawala algorithm

It is based on multicast communication.

- N inter-connected asynchronous processes, each with
  - unique id
  - Lamport's logical clock
- processes multicast request to enter:
  - timestamped with Lamport's clock and process id
- entry granted
  - when all other processes replied
  - simultaneous requests resolved with the timestamp

How it works:

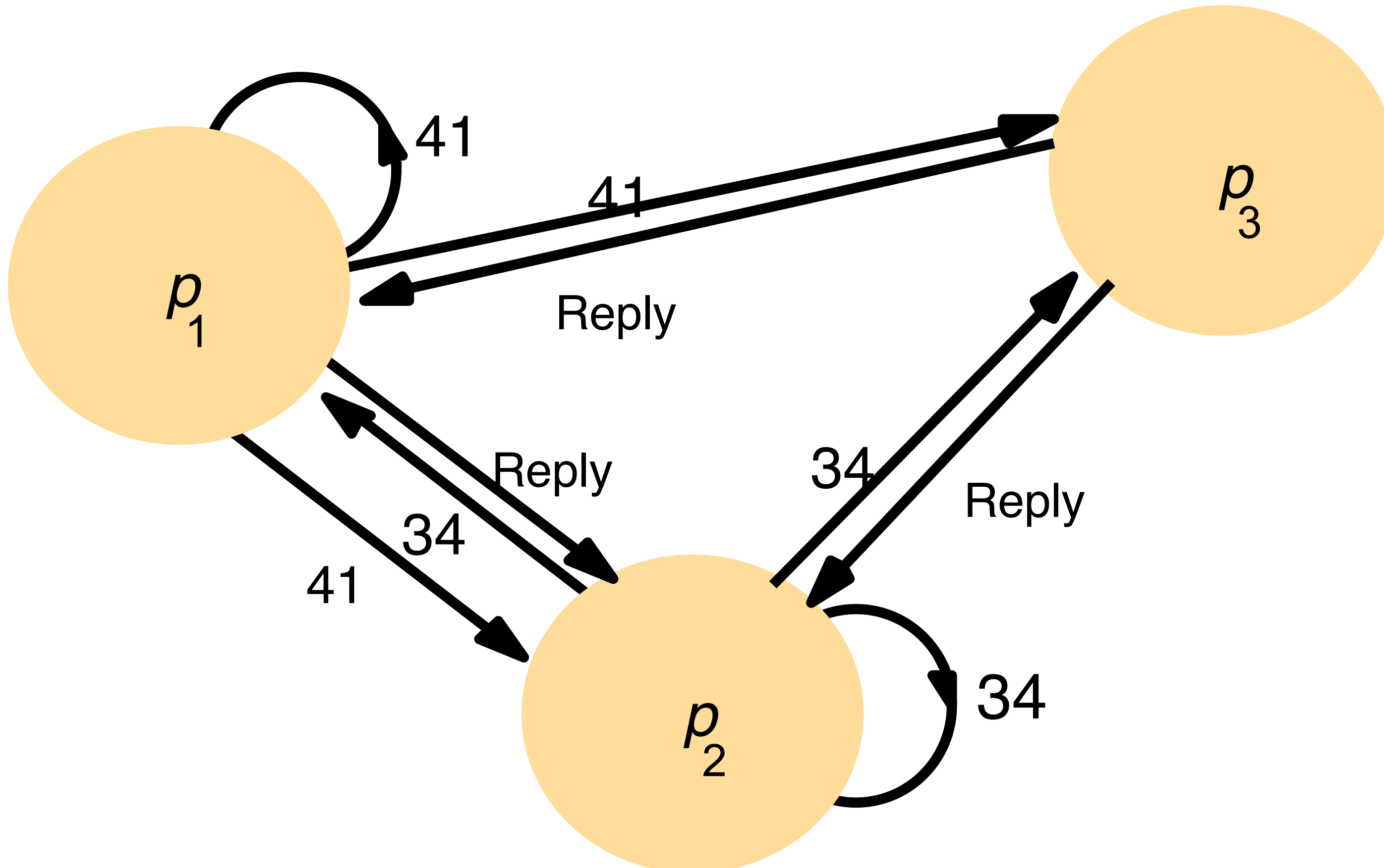
- satisfies the stronger property (MC3)
- if hardware support for multicast, there is only one message to enter

Ricart&Agrawala algorithm

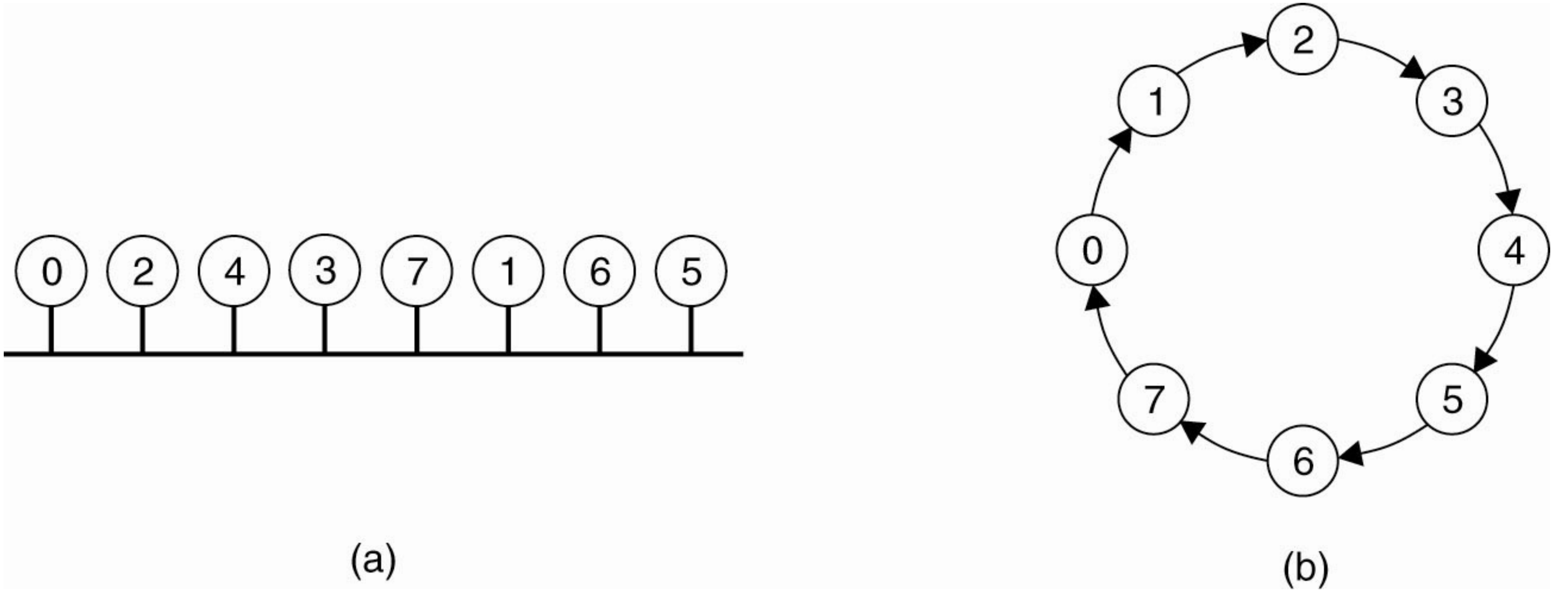
On initialization
state := RELEASED;
To enter the section
state := WANTED;
Multicast *request* to all processes; — request processing deferred here
T := request's timestamp;
Wait until (number of replies received = $(N - 1)$);
state := HELD;

On receipt of a request $\langle T_i, p_i \rangle$ *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;

Multicast mutual exclusion



A Token Ring Algorithm



- (a) An unordered group of processes on a network.
  (b) A logical ring constructed in software.

Ring-based algorithm

- No server bottleneck, no master
- Processes:
  - continually pass token around the ring, in one direction
  - if do not require access to CS, pass on to neighbour
  - otherwise, wait for token and retain it while in CS
  - to exit, pass to neighbour

How it works

– continuous use of network bandwith
– delay to enter depends on the size of ring
– causality order of requests not respected (MC3)

Maekawa's algorithm – part 1

On initialization

    state := RELEASED;
    voted := FALSE;

For p_i to enter the critical section

    state := WANTED;
    Multicast *request* to all processes in $V_i - \{p_i\}$;
    Wait until (number of replies received = $(K - 1)$);
    state := HELD;

On receipt of a request from p_i at p_j $(i \neq j)$

    if (*state* = HELD *or voted* = TRUE)
    then
        queue *request* from p_i without replying;
    else
        send *reply* to p_i;
        voted := TRUE;
    end if

Maekawa's algorithm – part 2

For p_i to exit the critical section

state := RELEASED;

Multicast *release* to all processes in $V_i - \{p_i\}$;

On receipt of a release from p_i at p_j $(i \neq j)$

if (queue of requests is non-empty)

then

remove head of queue – from p_k, say;

send *reply* to p_k;

voted := TRUE;

else

voted := FALSE;

end if

A Comparison of the algorithms

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

- A comparison of mutual exclusion algorithms.

End of lecture