Distributed Systems

Coordination
(Mutual exclusion & Election)

Thoai Nam

High Performance Computing Lab (HPC Lab)

Faculty of Computer Science and Engineering

HCMC University of Technology

Mutual exclusion

Mutual exclusion

- Problem
  - A number of processes in a distributed system want exclusive access to some resource
- Goal (avoiding)
  - Starvation: not every process gets a chance to access the resource
  - Deadlocks: several processes are waiting for each other to proceed
- Basic solutions
  - Permission-based: A process wanting to enter its critical section, or access a resource, needs permission from other processes
  - Token-based: A token is passed between processes. The one who has the token may proceed in its critical section, or pass it on when not interested.

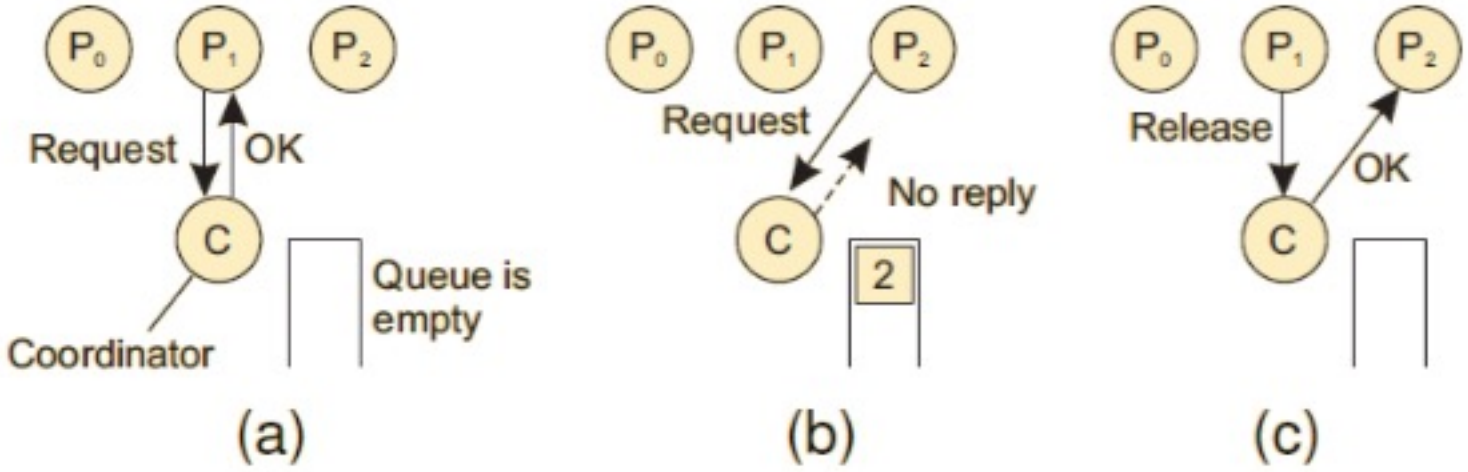
Critical section

- Critical section
  - Considering a system of N processes p_i (i = 1, 2,..., N), that do not share variables. The processes access common resources, but they do so in a critical section
- The application-level protocol for executing a critical section is as follows:
  - *enter():* enter critical section – block if necessary
  - *resourceAccesses():* access shared resources in critical section
  - *exit():* leave critical section – other processes may now enter
- Mutual exclusion requirements

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: (Optional, stronger)	Requests to **enter** granted according to causality order

Permission-based, centralized (1)

- Simply use of a coordinator
- One process is elected as the coordinator, which only lets one process at a time to access the resource:
  a) Process P_1 asks the coordinator for permission to access a shared resource. Permission is granted.
  b) Process P_2 then asks permission to access the same resource. The coordinator does not reply.
  c) When P_1 releases the resource, it tells the coordinator, which then replies to P_2.



Permission-based, centralized (2)

- Pros
  - Fairness
  - No starvation
  - Simplicity, only three messages (request, grant, release).
- Cons
  - Coordinator is a single point of failure and performance bottleneck
  - Distinguishing crashed coordinator from permission denied.

A distributed algorithm

- Return a response to a request only when:
  - The receiving process has no interest in the shared resource; or
  - The receiving process 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.

Multicast

- FIFO ordering: If a correct process issues multicast(*g*,*m*) and then multicast(*g*,*m'*), then every correct process that delivers *m'* will have already delivered *m*
- Causal ordering: If multicast(*g*,*m*) → multicast(*g*,*m'*) then any correct process that delivers *m'* will have already delivered *m*
  - Note that → counts multicast messages delivered to the application, rather than all network messages
- Total ordering: If a correct process delivers message *m* before *m'*, then any other correct process that delivers *m'* will have already delivered *m*.

A distributed algorithm: Ricart & Agrawala (1981)

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 $<T_i, p_i>$ *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

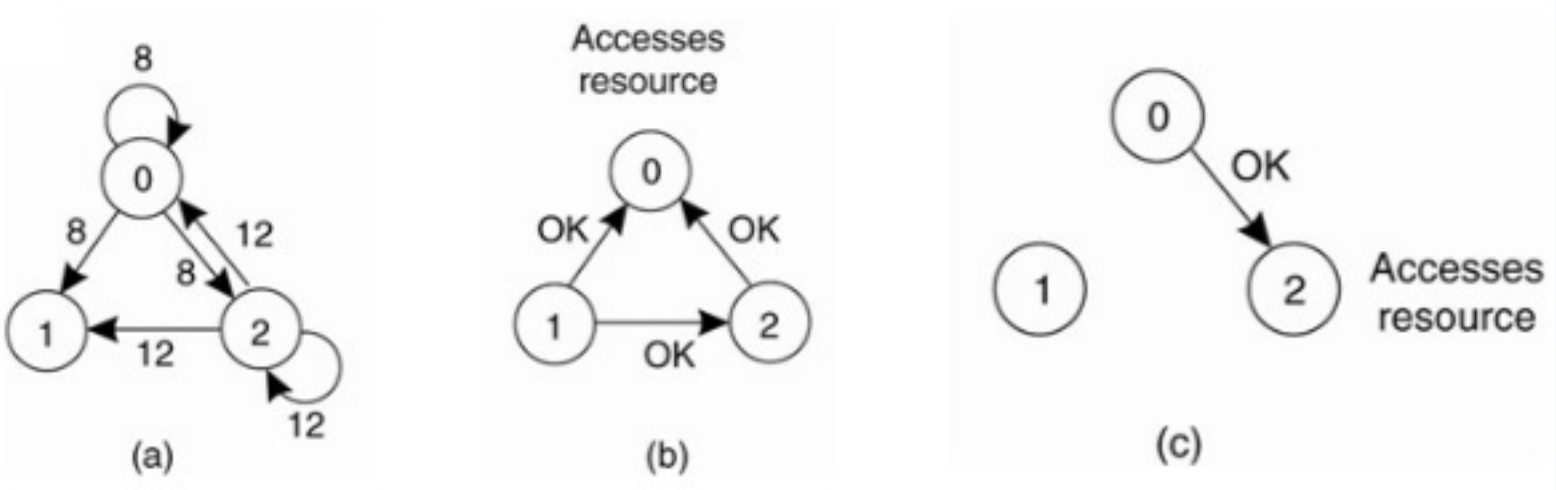
To exit the critical section

state := RELEASED;
reply to any queued requests;

Ricart & Agrawala (1981) (1)

- Based on total ordering of all events in the system
- A process wanting to access a resource builds a message containing the name of the resource, its process number and the current (logical) time
- The process sends the message to all other processes
- A process receiving the request has three alternatives:
    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
    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. If the incoming message has a lower timestamp, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing
- The process waits until everyone has given permission.

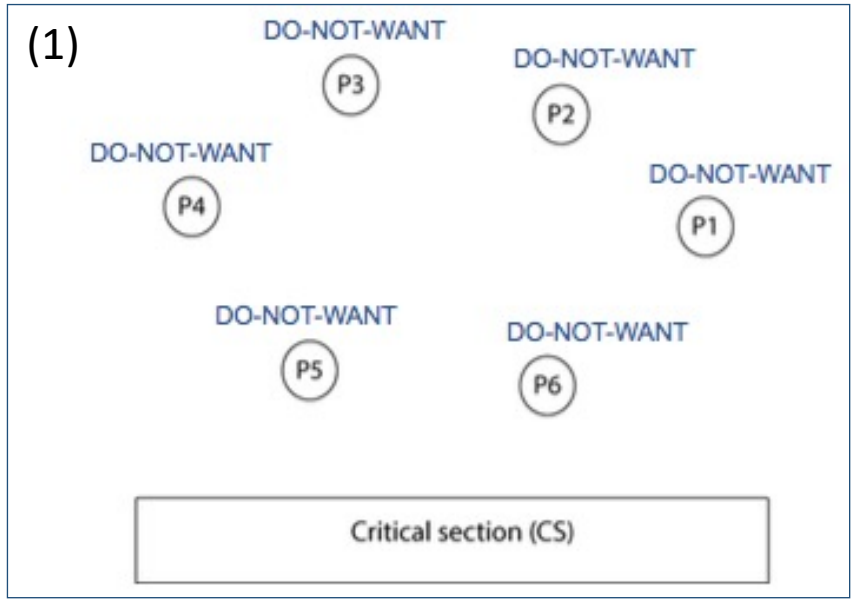
Ricart & Agrawala (1981): an example

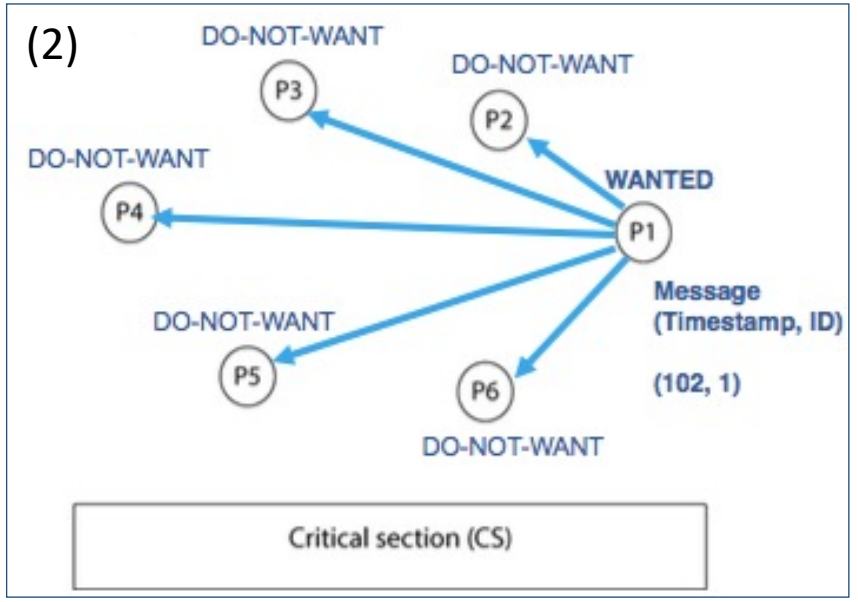


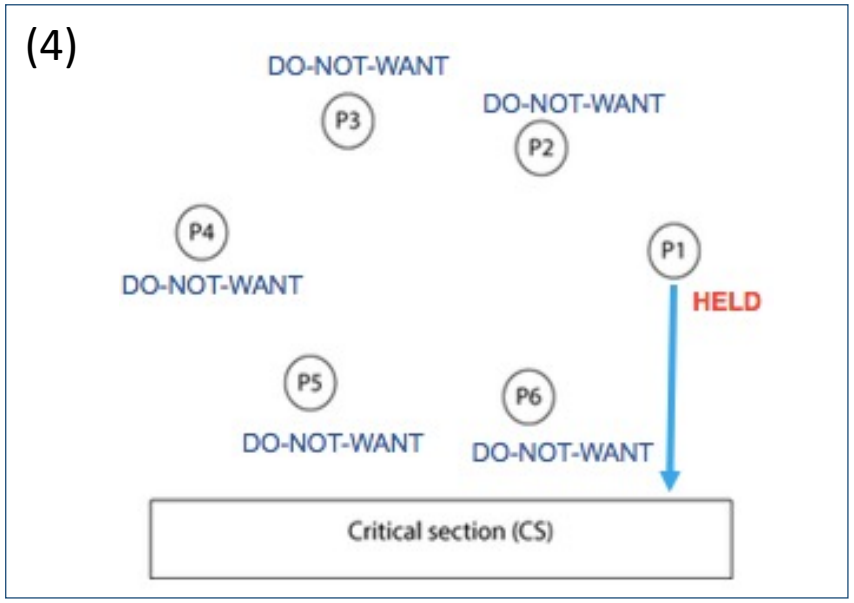
① Two processes want to access a shared resource at the same moment

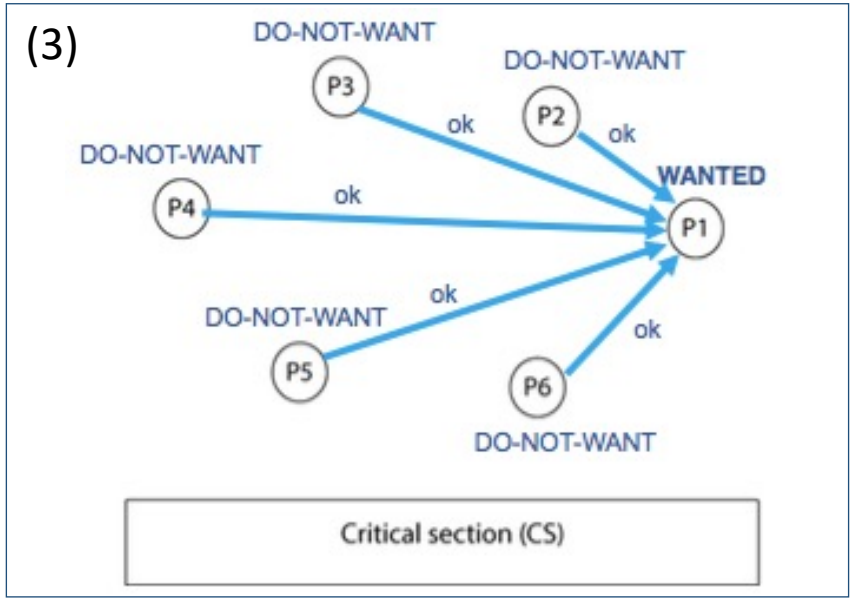
② P_0 has the lowest timestamp, so it wins

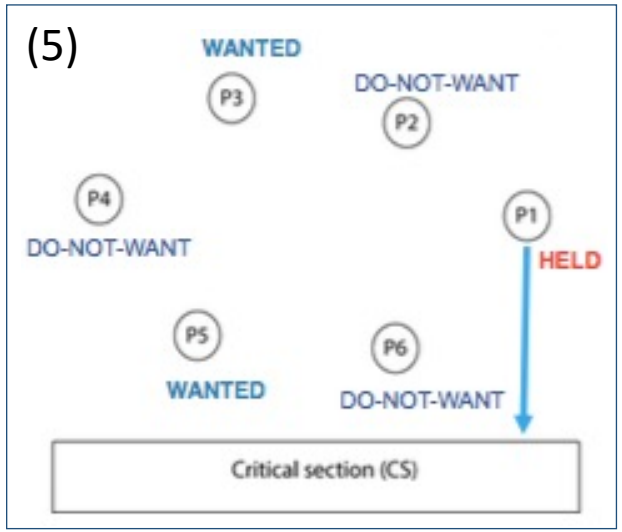
③ When process P_0 is done, it sends an OK also, so P_2 can now go ahead.

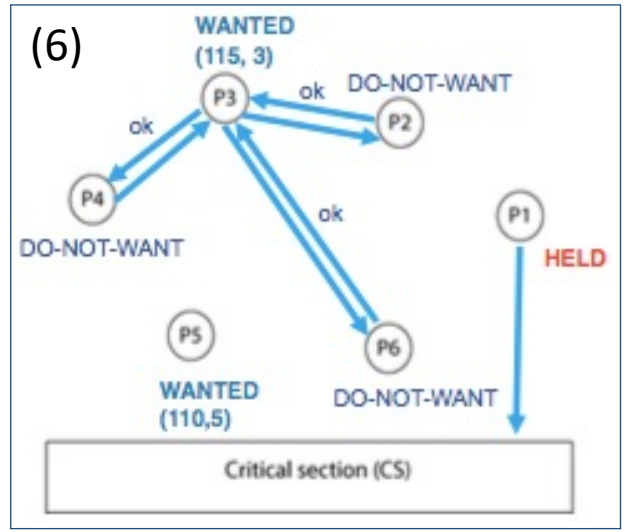

(1)
DO-NOT-WANT
P3
DO-NOT-WANT
P2
DO-NOT-WANT
P4
DO-NOT-WANT
P1
DO-NOT-WANT
P5
DO-NOT-WANT
P6
Critical section (CS)

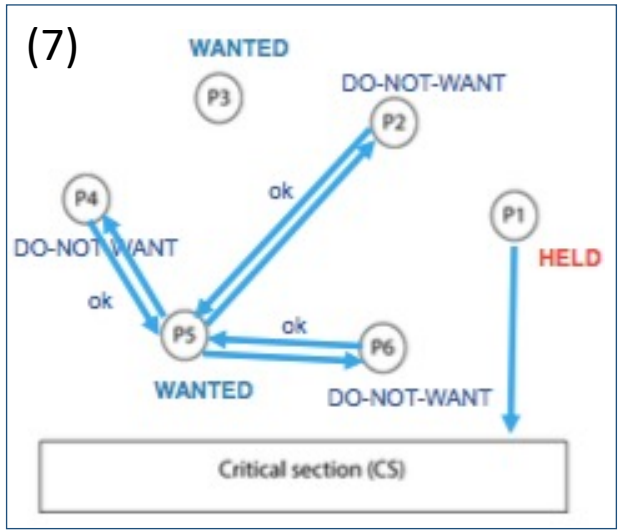

(2)
DO-NOT-WANT
P3
DO-NOT-WANT
P2
WANTED
DO-NOT-WANT
P4
P1
Message
(Timestamp, ID)
(102, 1)
DO-NOT-WANT
P5
P6
DO-NOT-WANT
Critical section (CS)

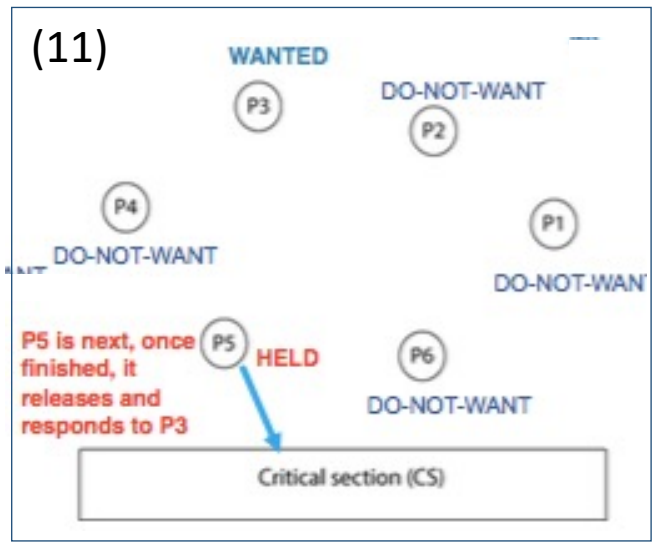

(4)
DO-NOT-WANT
P3
DO-NOT-WANT
P2
P4
DO-NOT-WANT
P1
HELD
P5
P6
DO-NOT-WANT
DO-NOT-WANT
Critical section (CS)

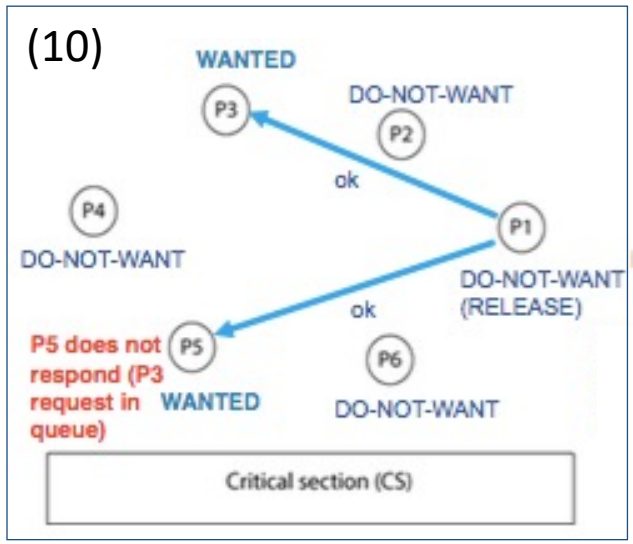

(3)
DO-NOT-WANT
P3
DO-NOT-WANT
P2
ok
ok
DO-NOT-WANT
WANTED
P4
ok
P1
ok
DO-NOT-WANT
P5
P6
ok
DO-NOT-WANT
Critical section (CS)

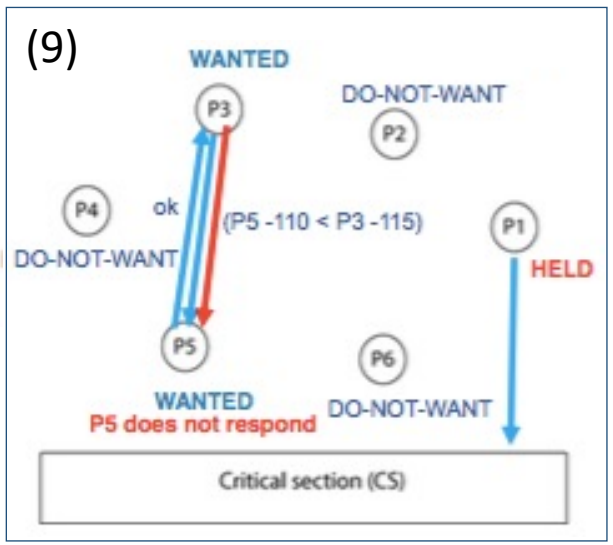

(5)
WANTED
P3
DO-NOT-WANT
P2
P4
DO-NOT-WANT
P1
HELD
P5
WANTED
P6
DO-NOT-WANT
Critical section (CS)

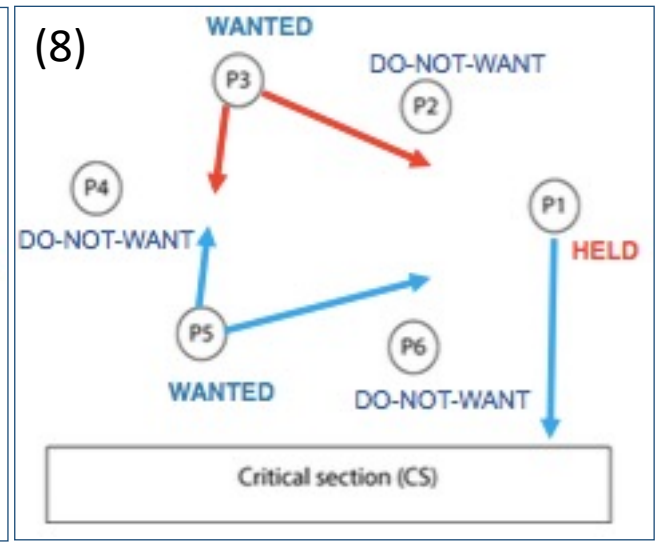

(6)
WANTED
(115, 3)
P3
ok
DO-NOT-WANT
P2
ok
P4
DO-NOT-WANT
ok
P1
HELD
P5
WANTED
(110,5)
P6
DO-NOT-WANT
Critical section (CS)


(7)
WANTED
P3
DO-NOT-WANT
P2
ok
P4
DO-NOT-WANT
P1
HELD
ok
P5
ok
P6
WANTED
DO-NOT-WANT
Critical section (CS)


(11)
WANTED
P3
DO-NOT-WANT
P2
P4
DO-NOT-WANT
P1
P5 is next, once finished, it releases and responds to P3
P5
HELD
P6
DO-NOT-WANT
Critical section (CS)


(10)
WANTED
P3
DO-NOT-WANT
P2
ok
P4
DO-NOT-WANT
P1
DO-NOT-WANT
(RELEASE)
P5 does not respond (P3 request in queue)
P5
WANTED
ok
P6
DO-NOT-WANT
Critical section (CS)


(9)
WANTED
P3
DO-NOT-WANT
P2
P4
ok
(P5 -110 < P3 -115)
DO-NOT-WANT
P1
HELD
P5
P6
WANTED
P5 does not respond
DO-NOT-WANT
Critical section (CS)

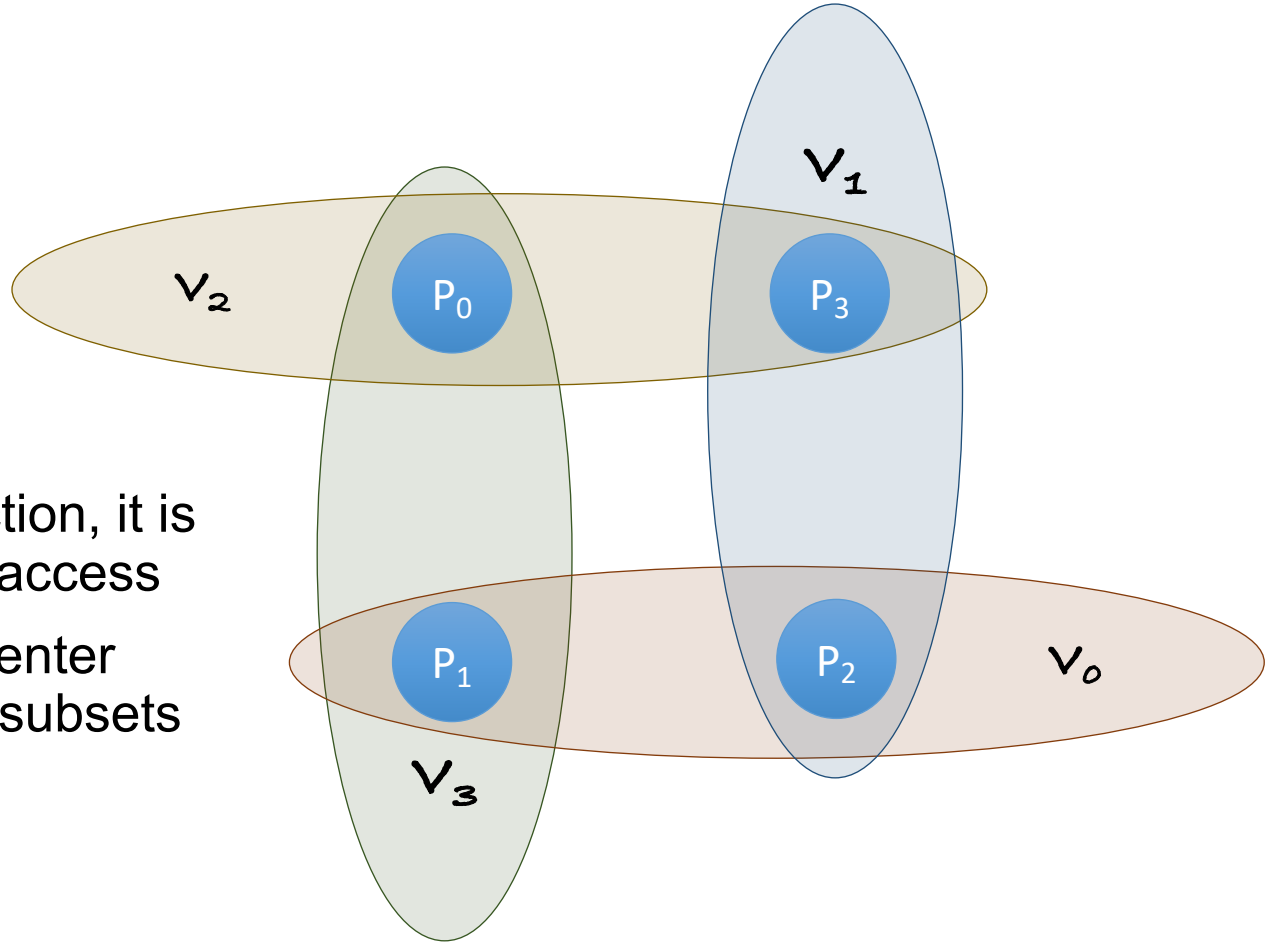

(8)
WANTED
P3
DO-NOT-WANT
P2
P4
DO-NOT-WANT
P1
HELD
P5
WANTED
P6
DO-NOT-WANT
Critical section (CS)

Ricart & Agrawala (1981)

- Pros
  - Mutual exclusion is guaranteed without deadlock or starvation
- Cons
  - N points of failure (crashed process interpreted as denial of access)
  - Requires more communication
  - Low efficiency, as all processes are involved in all decisions (n bottlenecks).

Observation

- In order for a process to enter a critical section, it is not necessary for all of its peers to grant it access
- Processes need only obtain permission to enter from *subsets* of their peers, as long as the subsets used by any two processes overlap.



Maekawa's algorithm (1985)

- A *voting set* v_i with each process p_i (i = 1, 2,..., N), where $v_i \subseteq \{p_1, p_2, ..., p_N\}$
- The sets v_i are chosen so that, for all i, j = 1, 2,..., N:
  - $p_i \in v_i$
  - $v_i \cap v_j \neq \emptyset$, there is at least one common member of any two voting sets
  - $|v_i| = K$ - to be fair, each process has a voting set of the same size
  - Each process p_i is contained in M of the voting sets v_i.
- Optimal solution
  - $K \sim \sqrt{N}$, $M = K$

Maekawa's algorithm

On initialization
    state := RELEASED;
    voted := FALSE;

For p_i *to enter the critical section*
    state := WANTED;
    Multicast *request* to all processes in V_i;
    Wait until (number of replies received = K);
    state := HELD;

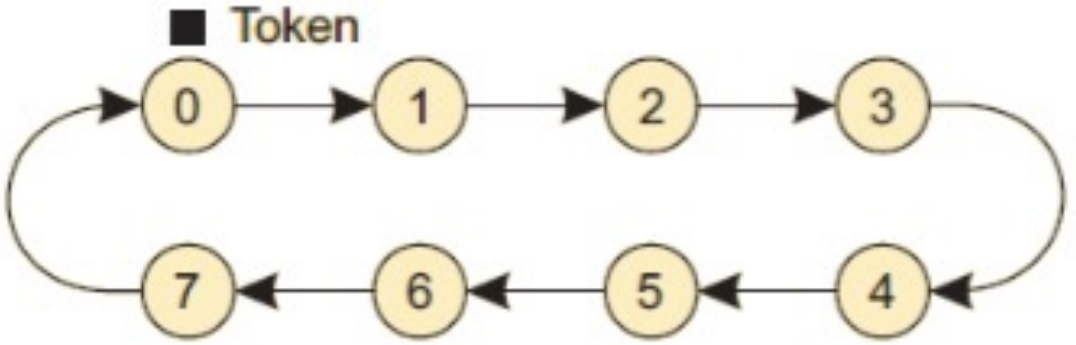
On receipt of a request from p_i *at* p_j
    if (*state* = HELD *or voted* = TRUE)
    then
        queue *request* from p_i without replying;
    else
        send *reply* to p_i;
        voted := TRUE;
    end if

For p_i *to exit the critical section*
    state := RELEASED;
    Multicast *release* to all processes in V_i;

On receipt of a release from p_i *at* p_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

Mutual exclusion: Token ring algorithm

- Organize processes in a logical ring, a token is introduced, and the idea is to let the token be passed between processes. The one that holds the token is allowed to enter the critical region (if it wants to)
- An overlay network constructed as a logical ring with a circulating token.

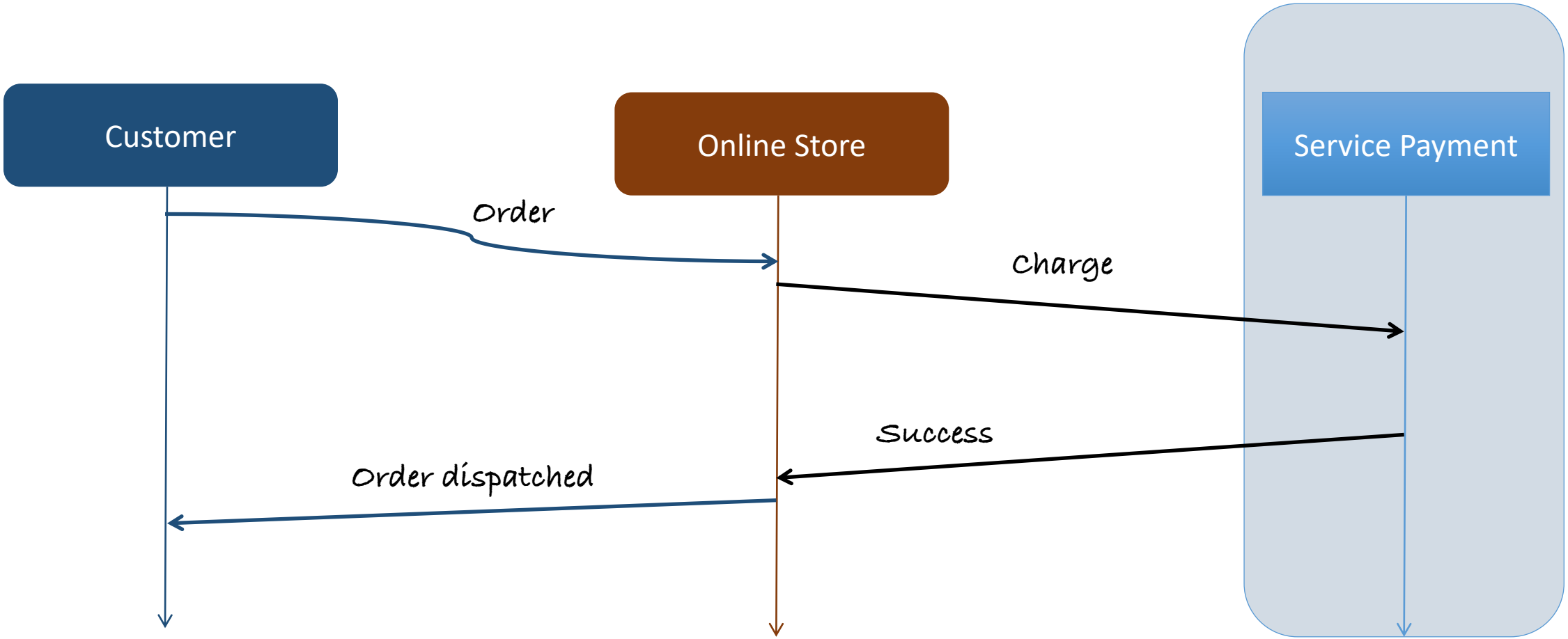


Token ring algorithm

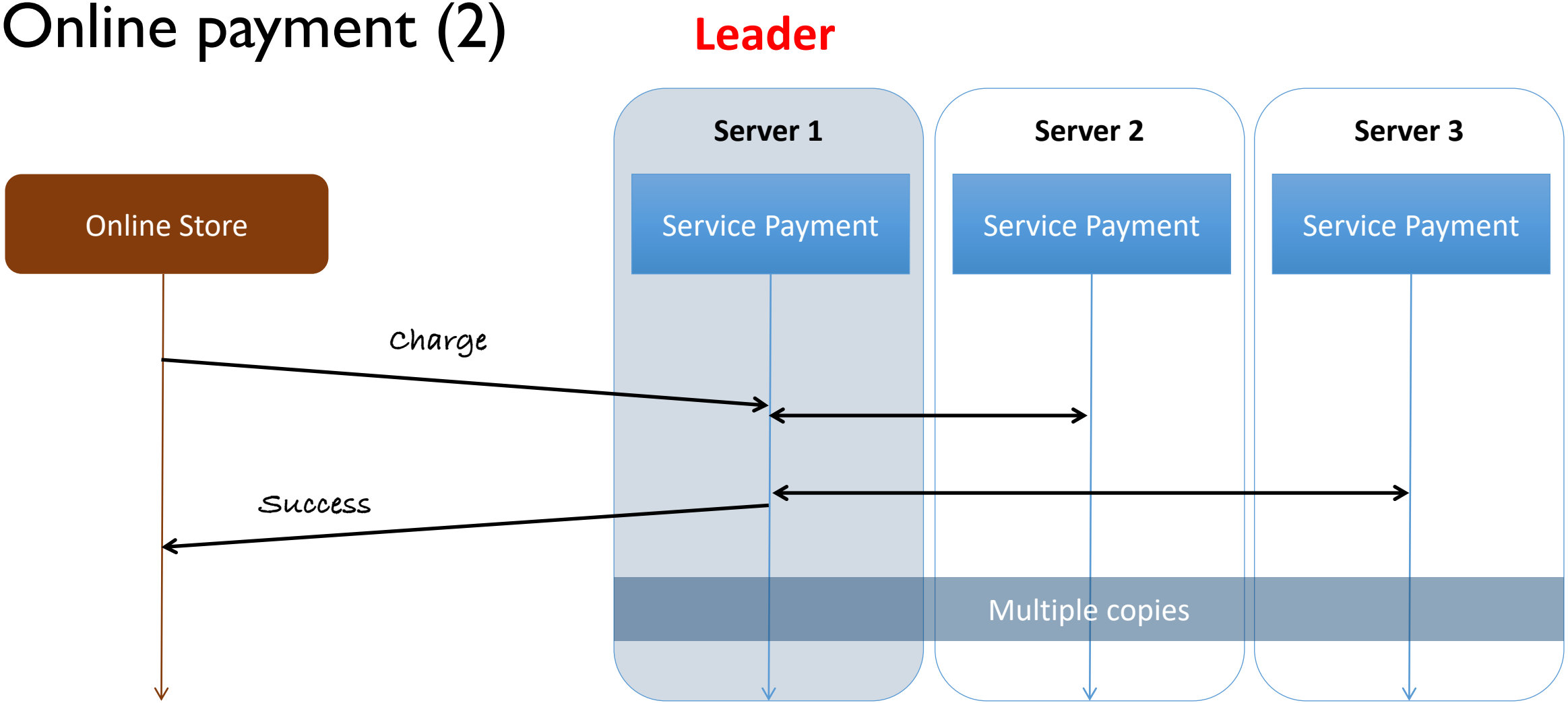
- Pros
  - No starvation
  - Relatively easy to recover
- Cons
  - Token can disappear (be loss)
  - Long delay between successive appearances of the token.

Leader election

Online payment (1)



Online payment (2)



Other applications

- Master/Slave
  - Hadoop (Map-reduce jobs)
- Primary/Secondary
  - DNS
- Primary/Replica
  - Drupal, Django, Amazon Relation Database Service
- Controller/Agent
  - Jenkins.

Leader election

- Sometimes a single process in the system needs to have special powers, e.g., grant access of a shared resource and assign work to others
- Selection of a leader given a candidate of processes/nodes
- When selecting a leader, two properties need to be guaranteed:
  - Safety: There is at most one leader at any given time
  - Liveness: The election eventually is completed
- The leader election process can be extended to provide consensus.

Election algorithms

- Principle: an algorithm requires that some process acts as a coordinator
  - The question is how to select this special process dynamically.
- In many systems the coordinator is chosen by hand (e.g. file servers)
  - This leads to centralized solutions → single point of failure.
- Questions:
  - If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?
  - Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

Basic assumptions

- All processes have unique id's
- All processes know id's of all processes in the system (but not if they are up or down)
- Election means identifying the most suitable process based on different factors, e.g., highest id.

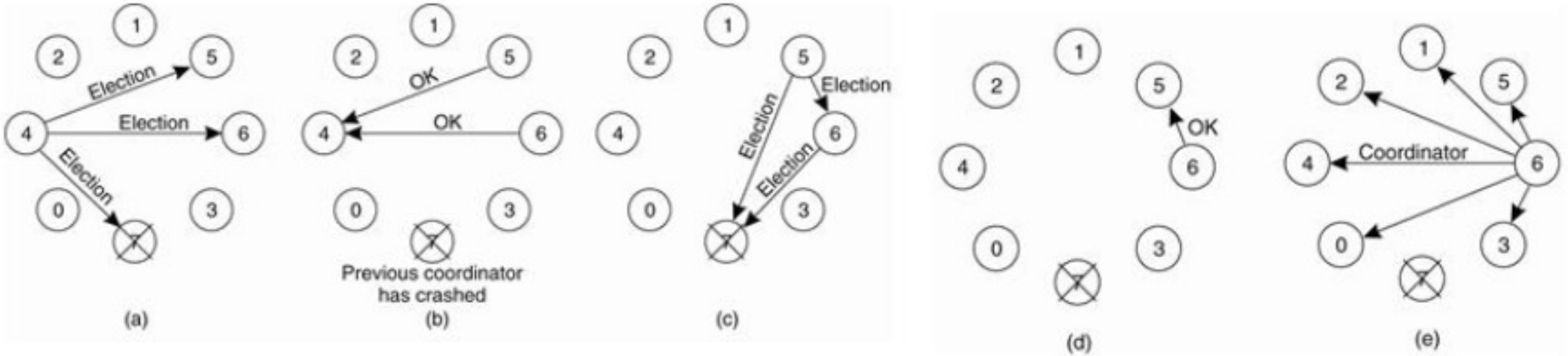
Election by bullying (1)

Principle (Garcia-Molina 1982)

Consider N processes $\{P_0 \ldots P_{N-1}\}$ and let $id(P_k) = k$. When a process P_k notices that the coordinator is no longer responding to requests, it initiates an election:

1. P_k sends an ELECTION message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$
2. If no one responds, P_k wins the election and becomes coordinator
3. If one of the higher-ups answers, it takes over and P_k 's job is done.

Election by bullying (2)



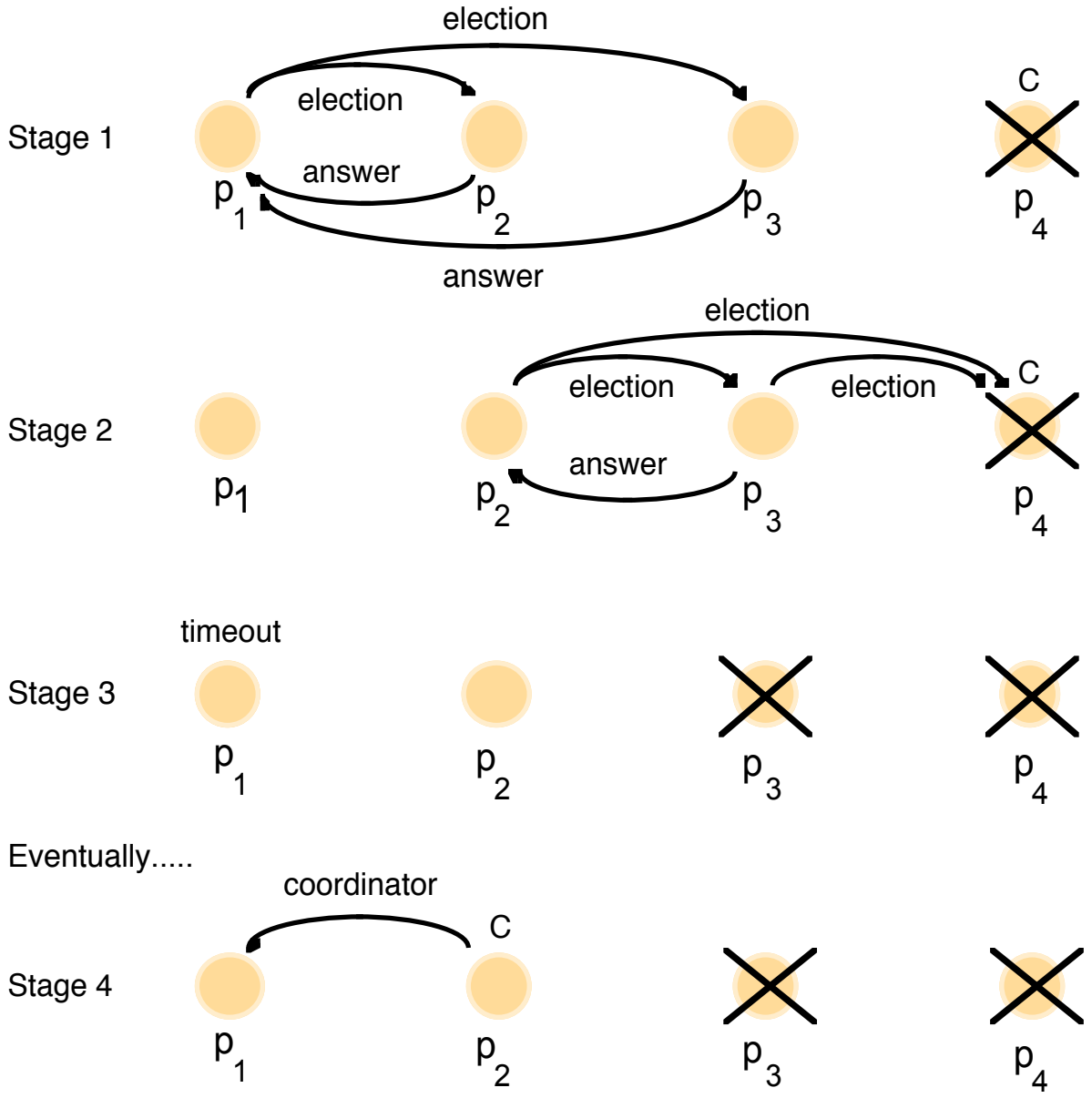
1. (a) Process 4 first notices that coordinator has crashed and sends ELECTION to processes with higher numbers 5,6, and 7
2. (b)-(d) Election proceeds, converging into process 6 winning
3. (e) By sending COORDINATOR message process 6 announces it is ready to take over
4. If process 7 is restarted, it will send COORDINATOR message to others and bully them into submission.

Election by bullying (2)

Principle (Garcia-Molina 1982)

Consider N processes $\{P_0 \ldots P_{N-1}\}$ and let $id(P_k) = k$. When a process P_k notices that the coordinator is no longer responding to requests, it initiates an election:

1. P_k sends an ELECTION message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$
2. If no one responds, P_k wins the election and becomes coordinator
3. If one of the higher-ups answers, it takes over and P_k 's job is done.

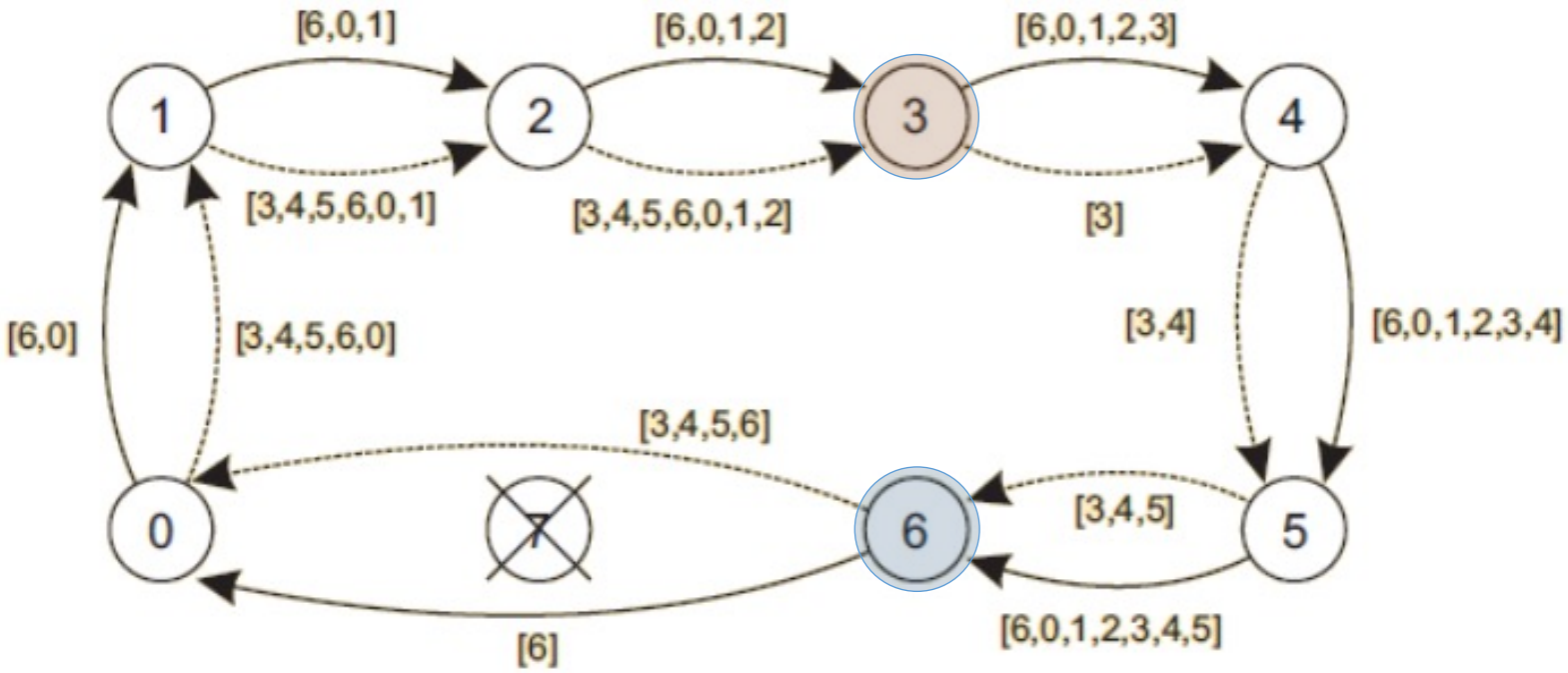


The election of coordinator p_2, after the failure of p_4 and then p_3

Election in a ring

- Process priority is obtained by organizing processes into a (logical) ring
- Process with the highest priority should be elected as coordinator
  - Any process can start an election by sending an election message 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.

Election in a ring: an example



- The initiators are P_6 and P_3.

Election in wireless networks