Distributed Mutual Exclusion & Leader Election

Yao Liu

Distributed mutual exclusion

- A number of processes in a distributed system want exclusive access to some shared resource

- Critical section
  - At most one process executing the critical section at any time

Approaches to mutual exclusion

- On a single OS:
  - If all processes run on a single OS on a machine (or VM), then
  - Semaphore, mutex, condition variables, monitors, etc.
- Distributed systems
  - Cannot use shared variables like semaphores
  - Processes communicate by message passing
    - Centralized solution
    - Ricart-Agrawala's algorithm
    - Ring-based approach

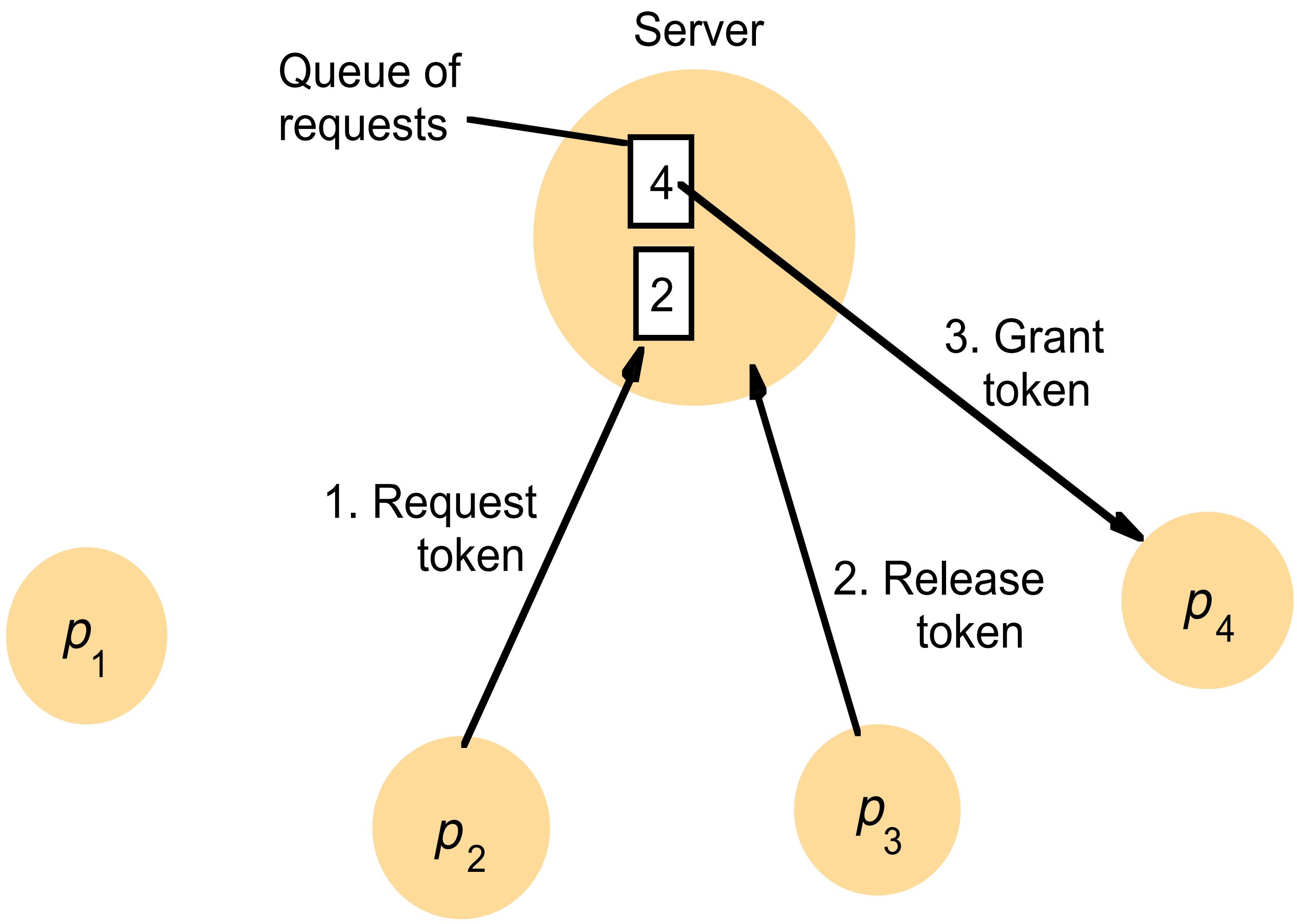
Centralized solution

- A single centralized server, e.g., master
- A unique token managed by the master
- Must hold the token to enter the critical section
- For any process to enter the critical section:
  - Send a request to the master
  - Wait for token from the master
- To exit the critical section:
  - Send back the token to the master

Master actions

- On receiving a request from process Pi
  - if (master has token)
    - Send token to Pi
  - else
    - Add Pi to queue
- On receiving a token from process Pi
  - if (queue is not empty)
    - Extract the first process from the queue, Pj
    - Send Pj the token
  - else
    - Hold the token

Server managing a mutual exclusion token for a set of processes



Ricart and Agrawala’s algorithm

- No token
- No centralized master

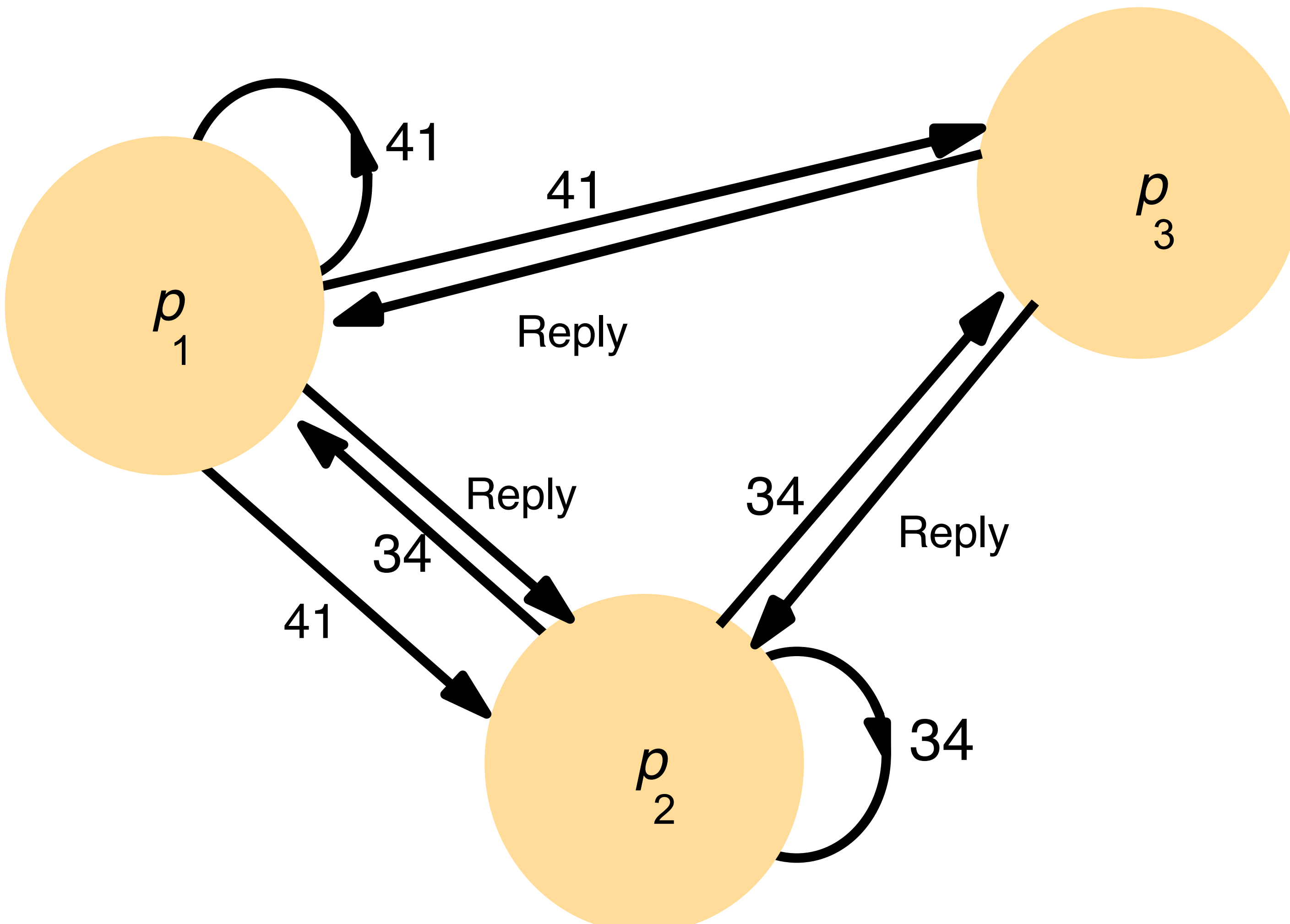
- For a process to enter the critical section
  - Send requests to all processes
  - Wait until **all** other processes have responded positively to request

Ricart and Agrawala’s algorithm

On initialization
    state := RELEASED;
To ***enter*** *the section*
    state := WANTED;
    Send *requests* to all processes;
    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* $(T_j, p_j) < (T_i, p_i)$))
    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;

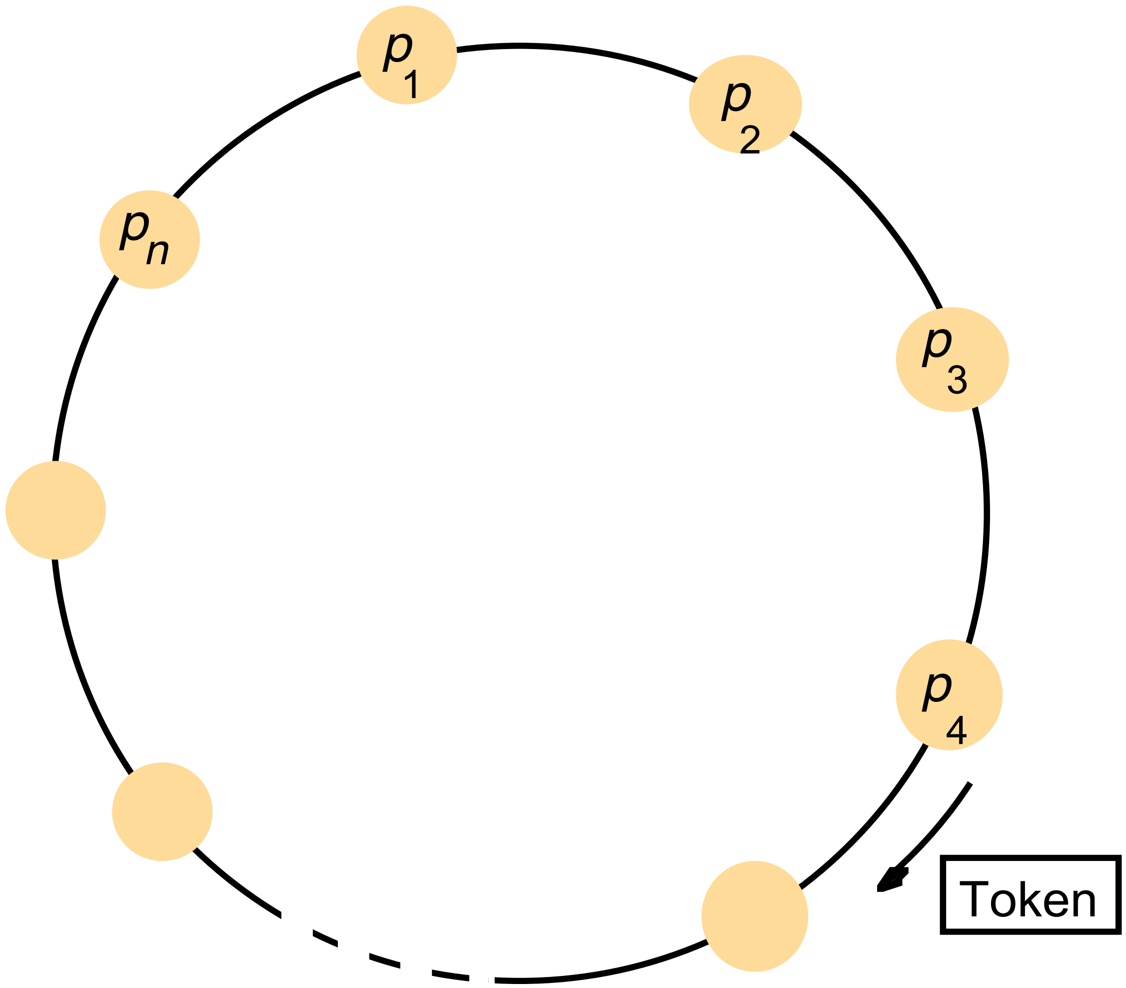
Example



Ring-based solution

- N processes organized into a virtualized ring
- Each process on the ring can communicate with its successor
- A unique token to grant access
- To enter: wait til a process gets the token
- To exit: pass the token to the successor
- If receive the token but do not need to enter, simply pass it to the successor

A ring of processes transferring a mutual exclusion token



Comparison

- A comparison of three mutual exclusion algorithms:

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

Maekawa's algorithm

- Every node needs permission from other nodes in its quorum before it can enter the critical section
- **Quorums** are constructed in such a way that no two nodes can be in their critical section at the same time
- Each quorum is of size *K*
- The size of each node's quorum is $O(\sqrt{N})$, which can be shown to be optimal

Construction of quorum sets

Consider a system with 9 nodes

The quorum for any node includes the nodes in its row and column

1	2	3
4	5	6
7	8	9

Quorum for Node 1 = {1,2,3,4,7}

Quorum for Node 9 = {3,6,7,8,9}

There is a non-null intersection for the quorums of any two nodes

Maekawa's algorithm

On initialization

state := RELEASED;
voted := FALSE;

For p_i to ***enter*** *the critical section*

state := WANTED;
Send *requests* 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 – cont'd

For p_i to ***exit*** *the critical section*

state := RELEASED;

Send *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

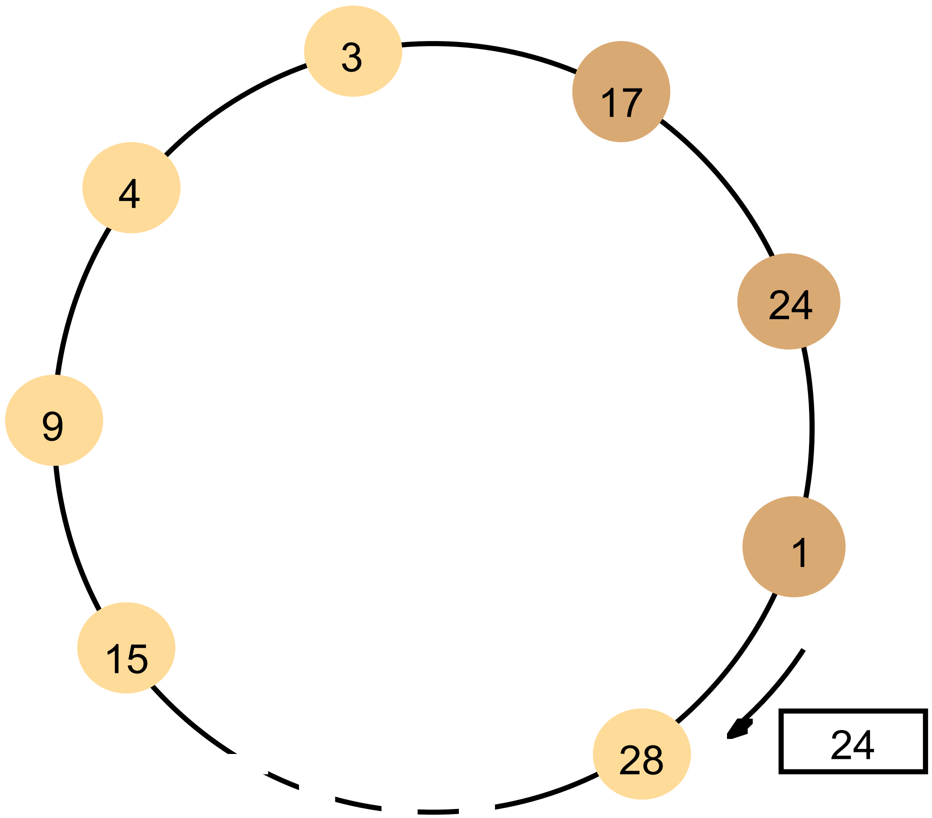
Leader election

- An election is a procedure carried out to choose a process from a group, for example to take over the role of a process that has failed
- Main requirement: elected process should be **unique** even if several processes start an election simultaneously
- Algorithms:
  - Ring-based election: processes need to know only addresses of their immediate neighbors
  - Bully algorithm: assumes all processes know the identities and addresses of all the other processes

Ring-based election

- N processes are organized in a virtual ring
- Any process p_i that discovers the old leader has failed initiates an **election** message that contains p_i 's own ID.
- Upon receiving an election message:
  - The process compare its own ID with the ID in the election message
  - If its own ID is smaller, simply forward the message
  - If its own ID is larger, and the process has not forwarded an election message earlier, it overwrites the message with its own ID and forwards it
  - If the same, must be that this message has circled around and arrived back. Elect itself as the new coordinator and send out an **elected** message.

A ring-based election in progress

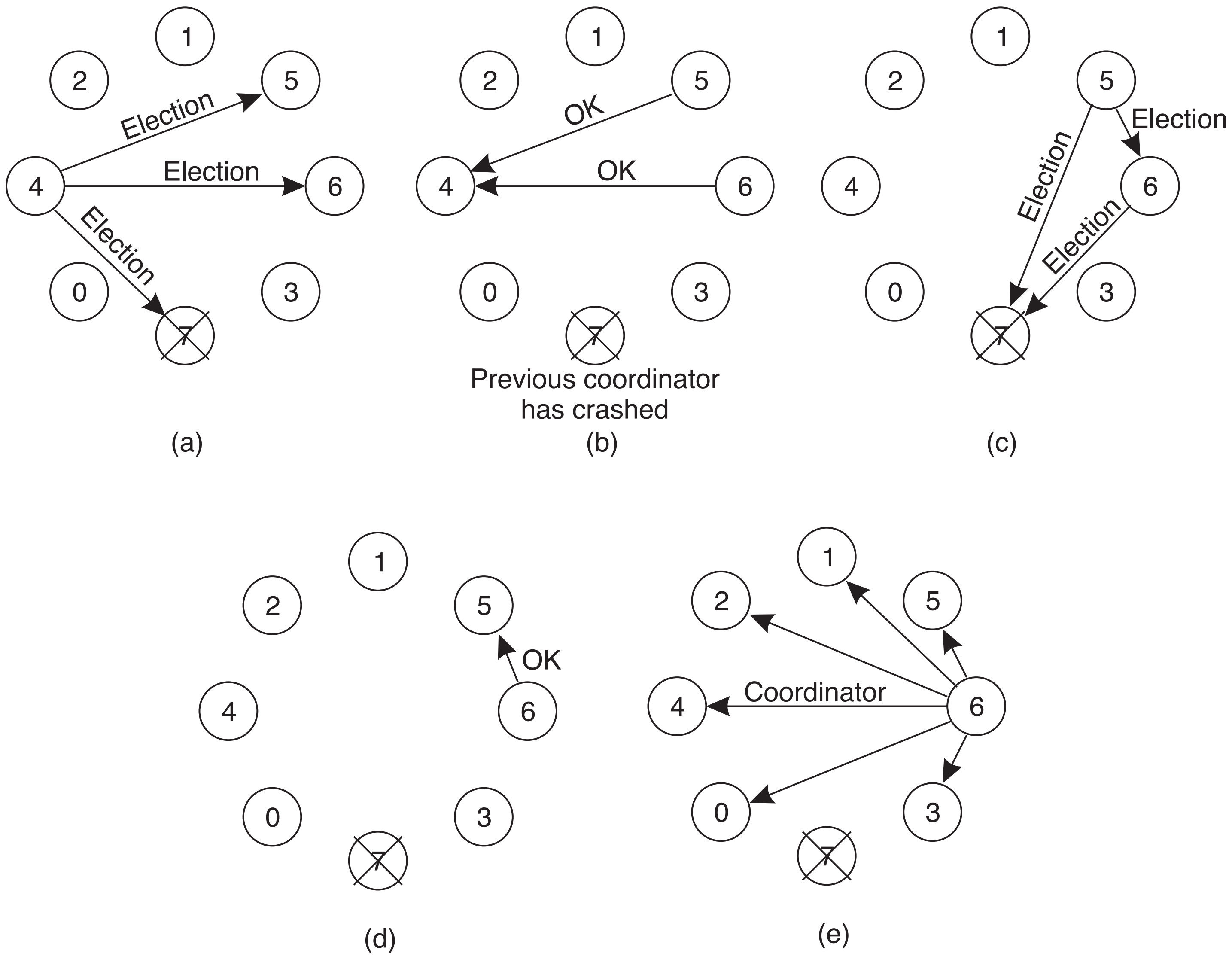


Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown darkened

Bully algorithm

- Each process has an associated priority, e.g., process ID
- The process with the highest priority should always be elected as the coordinator.
  - Any process can just start an election by sending an **election** message to all other processes with larger IDs
  - Processes with larger IDs respond with **answer** message, send out **election** messages to those with even larger IDs
  - Upon receiving **answer** message, wait
  - If a process does not receive any answer message, announce itself as the coordinator, sends out **coordinator** message to all

Example



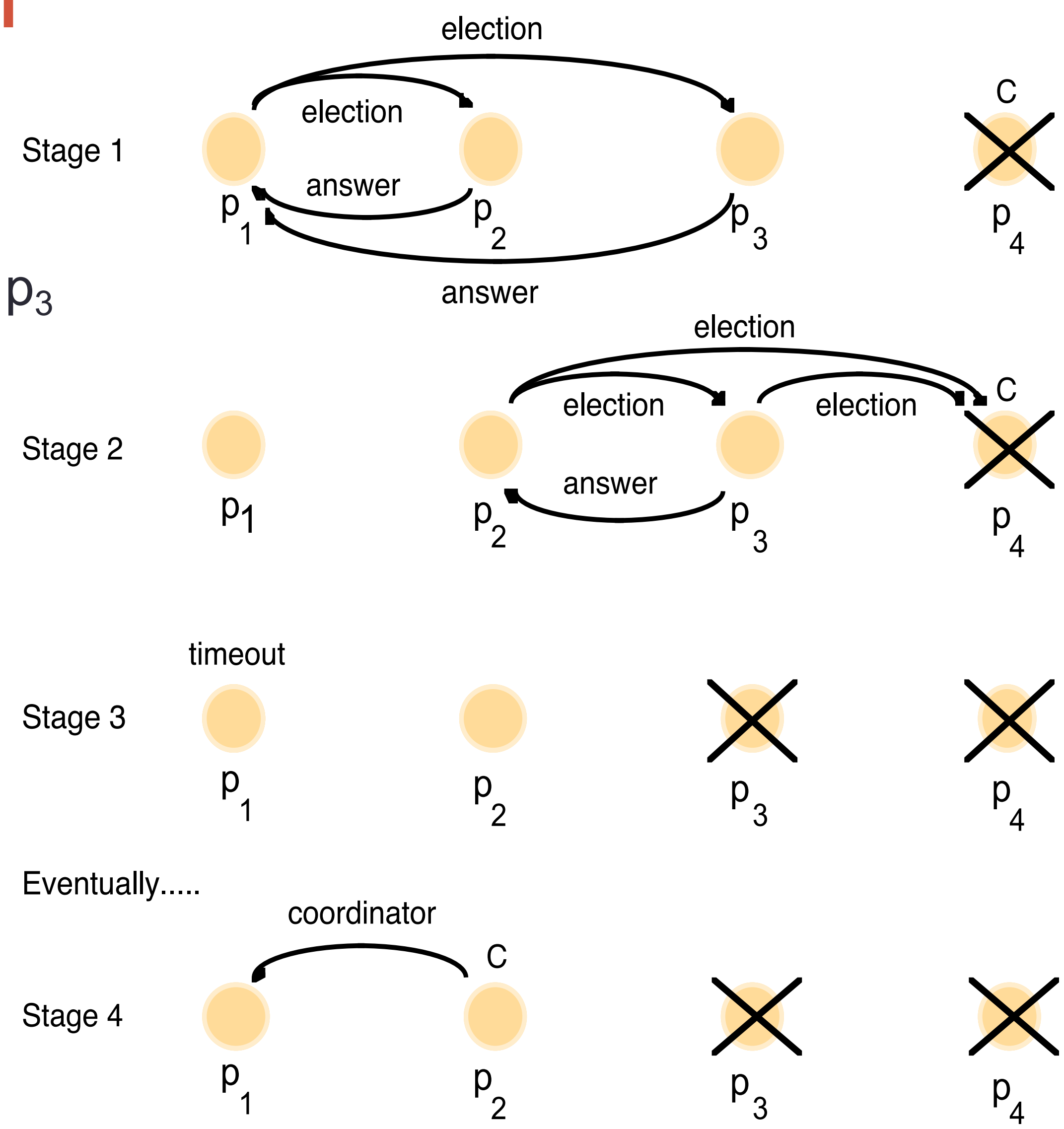
1
2
5
4
6
0
3
7
Election
Election
Election
(a)
OK
OK
Previous coordinator
has crashed
(b)
Election
Election
Election
(c)
OK
(d)
Coordinator
(e)

Bully algorithm

- Assumes the system is synchronous
  - The time to execute each step of a process has an upper and lower bound
  - Each message transmitted over a channel is received within a known bounded delay
  - Each process has a local clock whose drift rate from real time has a known bound
- Therefore, can use timeout to detect failure

Bully algorithm

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



Reading

- Sections 6.3 and 6.5 of TBook
- Sections 15.2 and 15.3 of CBook
- Paper on myCourses