Mutual Exclusion & Election

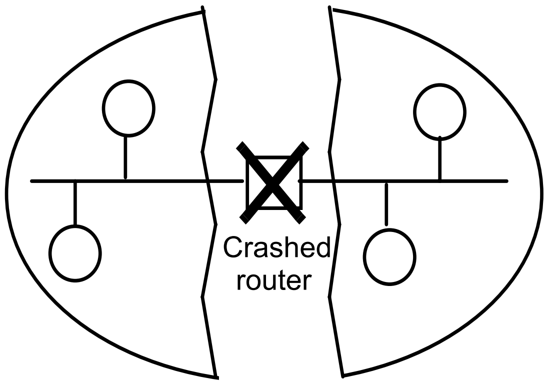
Brian Nielsen

bnielsen@cs.aau.dk

Failure Assumptions

- Reliable channels
  - Guaranteed delivery ***eventually*** in asynchronous systems
  - Guaranteed delivery within bound D)
  - $\Rightarrow$ network partitions/paths eventually repaired
- Independent processes $P_1 \ldots P_n$

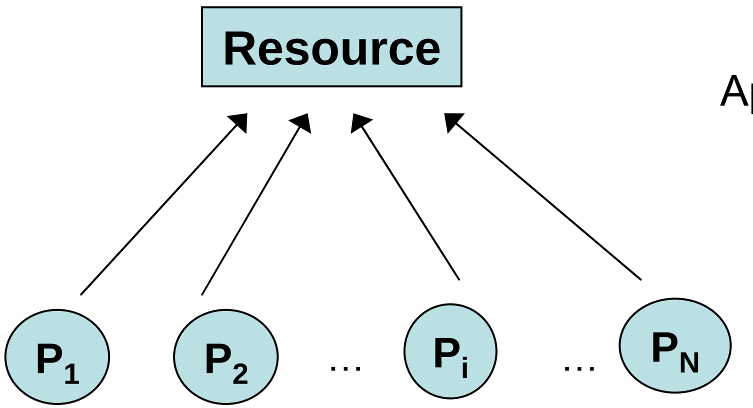
Network Partitioning



- Crash failures
  - ***Cannot*** be detected ***reliably*** in an asyncronous system by timeout
  - Heartbeats or probing in synchronous systems

Distributed mutual exclusion

- A number of processes want to access some shared resource
- Prevent interference, maintain consistency; critical section.



Application-level protocol:

```
enter()          // block till free
resourceAccess() // critical section
exit()           // free resource
```

General **requirements** for mutual exclusion

ME1: safety: at most one process may execute in the critical section at a time

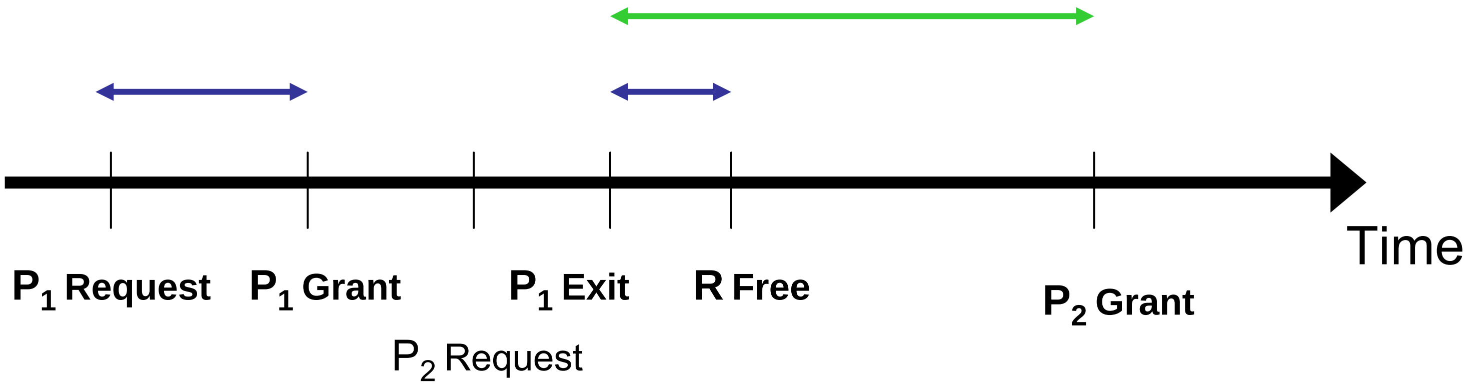
ME2: liveness: requests eventually succeed *(no deadlock, no starvation)*

ME3: ordering: if request A *happens-before* request B then grant A before grant B

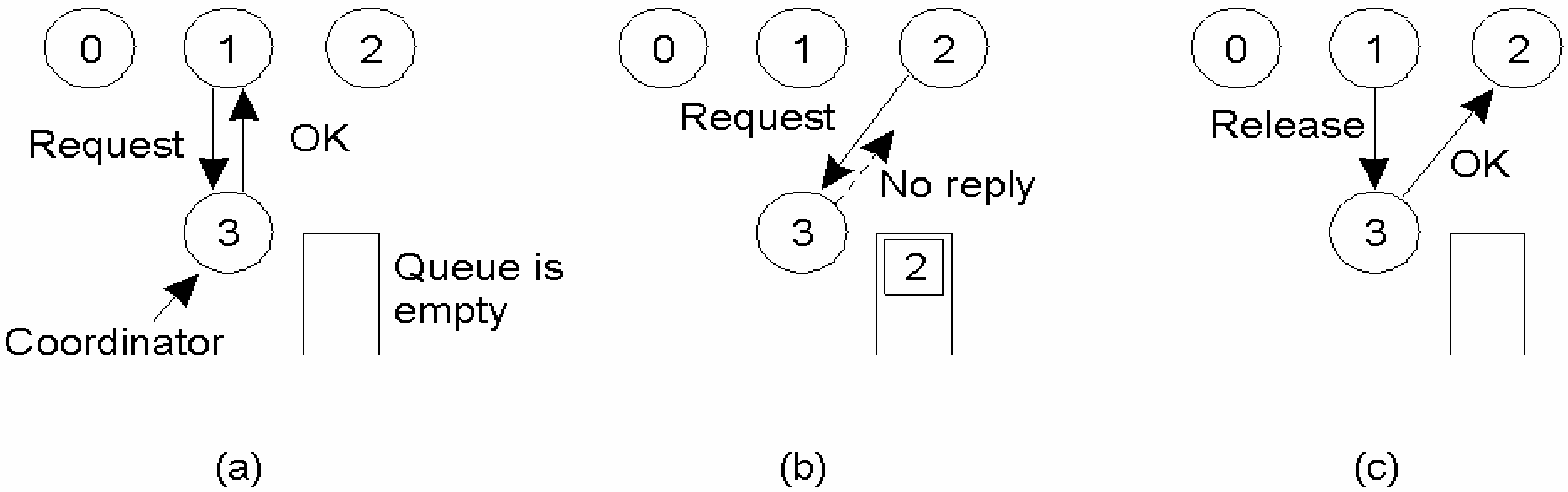
Problems: fault tolerance, performance

Performance Measures

- Bandwidth: number of messages required for entry and exit
- Client delay (entry and exit)
- Throughput (Synchronization delay)



Mutual Exclusion: A 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, which then replies to 2

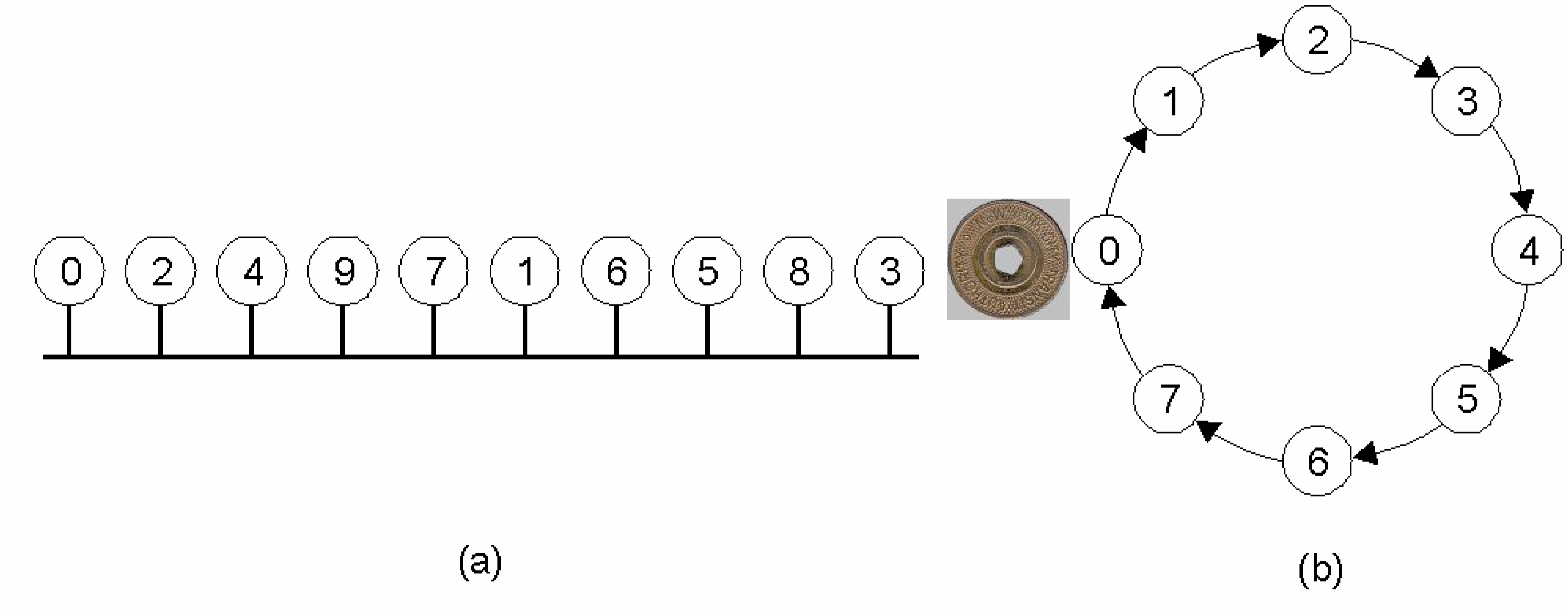
Mutual Exclusion: A Centralized Algorithm

- Shortcomings
  - The coordinator is a single point of failure, so if it crashes, the entire system may go down.
    - Wait; why not just elect another coordinator?
    - You can. The only concern is figuring out who has access to the critical section.
    - How do you tell the difference between a dead coordinator and "permission denied"?
  - In a large system a single coordinator may become a performance bottleneck.
- Advantages:
  - Simple
  - Reasonable efficient

Mutual Exclusion: A Centralized Algorithm

- Bandwidth
  - 3 messages to enter and leave a critical region: A request, a grant to enter and a release to exit
- Client Delay:
  - Entry: 2 messages: request + grant
  - Exit: 0 (asynchronous sending of release)
- Synchronization Delay: release + grant

A Token Ring Algorithm



a) An unordered group of processes on a network.
b) A logical ring constructed in software.
c) Token holder may enter CS

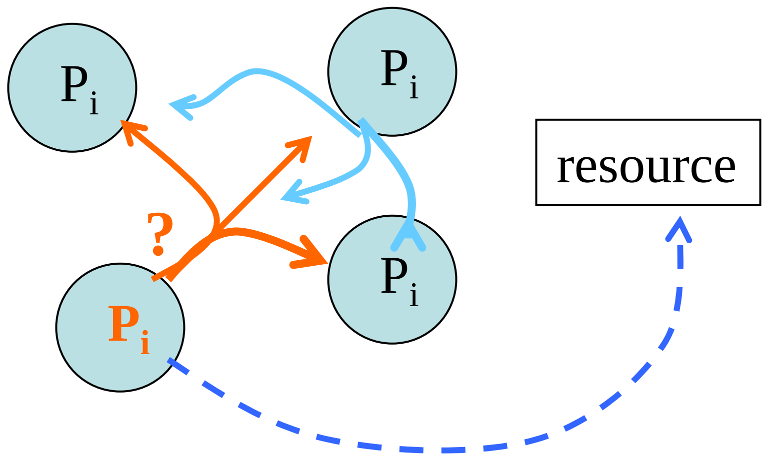
Token Ring

- Client Delay
  - Entry: Wait: 0…N hops (N/2 in average)
  - Exit: send 1 msg (asynchronously)
- Synchronization Delay
  - 0…N hops (N/2 in average)
- Bandwidth
  - Always uses bandwidth to circulate token, used or not.

Ricart and Agrawala's Algorithm [`81]

- Fully Distributed
- Optimized version of Lamports '78 algorithm
- Send "request" to N–1 other processes.
- Execute CS when "reply OK" permission is received from all other processes.
- P_i maintains Lamport Clock
  - I.e., adjust counter C_i on every internal event, and send and receive
- Break ties with Lamport time-stamp.

Ricart and Agrawala



- The general idea:
  – ask everybody
  – wait for permission from everybody

The problem:

– several simultaneous requests (e.g., P_i and P_j)

– all members have to agree (*everybody*: "first P_i then P_j")

Ricart – Agrawal's Algorithm

On initialization

state := RELEASED;

To enter the section

state := WANTED;

T := request's timestamp;

Multicast *request* to all processes;

request processing deferred here

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 OK immediately to p_i;

end if;

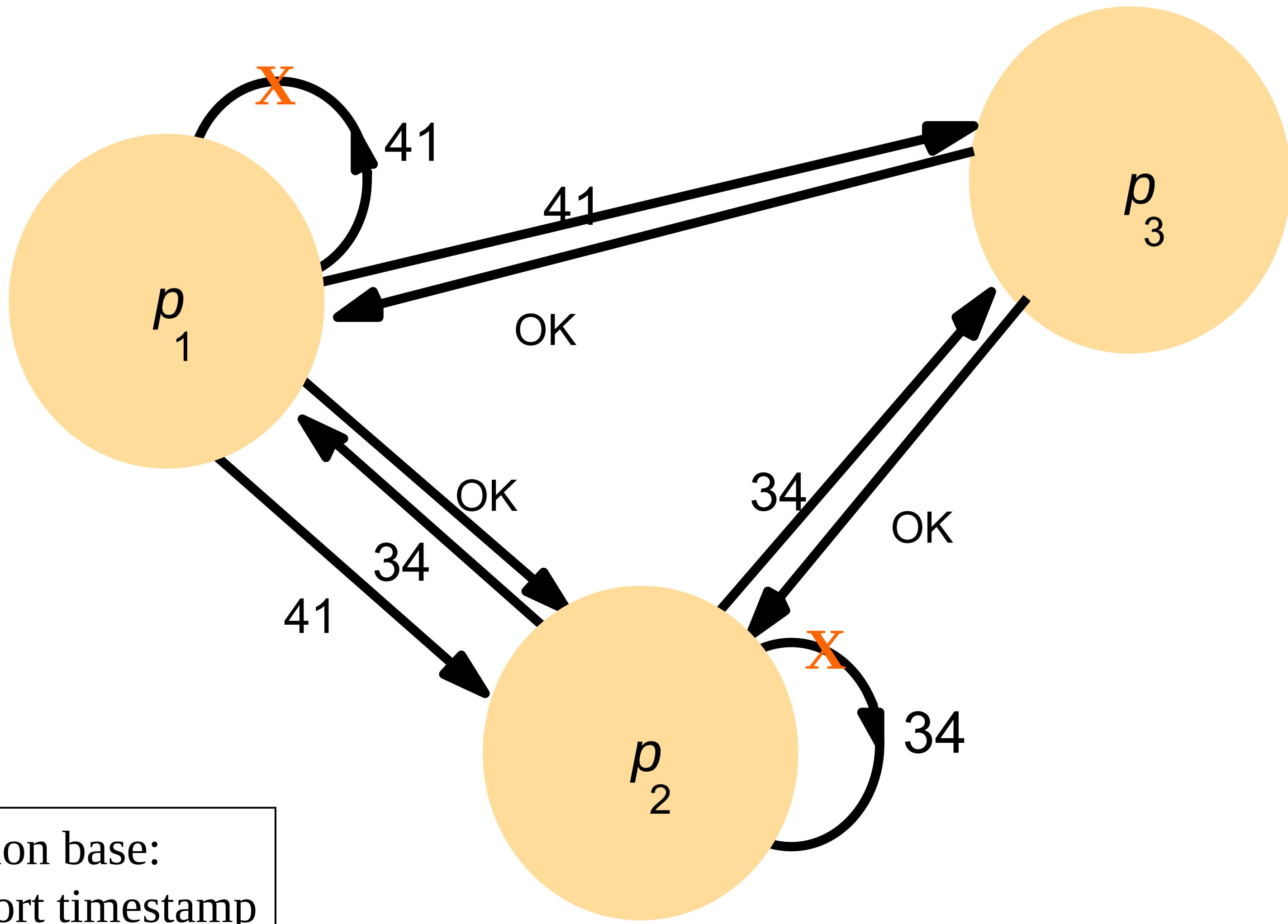
To exit the critical section

state := RELEASED;

reply OK to all queued requests;

Ricart – Agrawala EX.

P1 and P2 requests access concurrently at time 41 and 34



Decision base:
Lamport timestamp

X: i=j

Performance

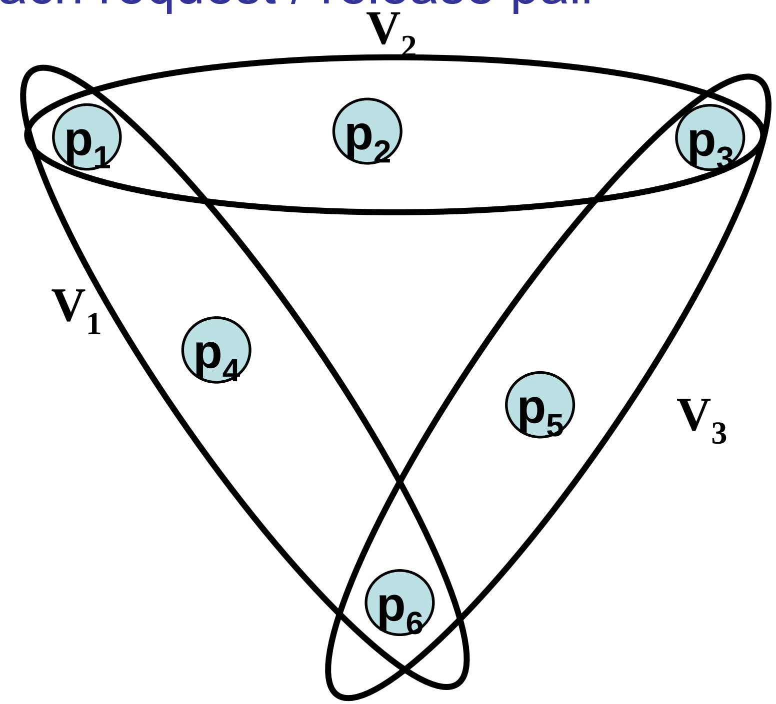
- Gaining entry: 2(n-1) messages per request without HW-multicast
  - N-1 to multicast request
  - N-1 replies
- Client Entry Delay: 1 round-trip time (multicasting is counted as 1 step)
- Client Exit Delay: 1 message
- Synchronization delay is one message
- N-points of failure

Maekawa's Algorithm [1981]

- **Idea:** Get permission from only a subset of processes.
  - **quorum:**
    - ”The minimal number of officers and members of a committee or organization, usually a majority, who must be present for valid transaction of business.”

Voting

- To enter its CS, a process gets permission from all members of its group
  - A process may grant permission to only one process at a time (between each request / release pair



- Complexity depends on group size
  - Want to minimize group size

Voting-Sets

Voting-set V_i for P_i

1. $\forall i,j$: $V_i \cap V \neq \varnothing$
   - Safety: at least one common member of any two voting-sets
2. V_i contains process p_i
   - Saves a message
3. $|V_1| = |V_2| = \ldots = |V_N| = K$
   - *Fairness: every process has a voting set of the same size*
4. Each process is in M of the voting sets V_i's
   - Each processor has the same responsibility

- Minimal K satisfying 1..4 is $c\sqrt{N}$.
- Heuristic algorithms exist

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;

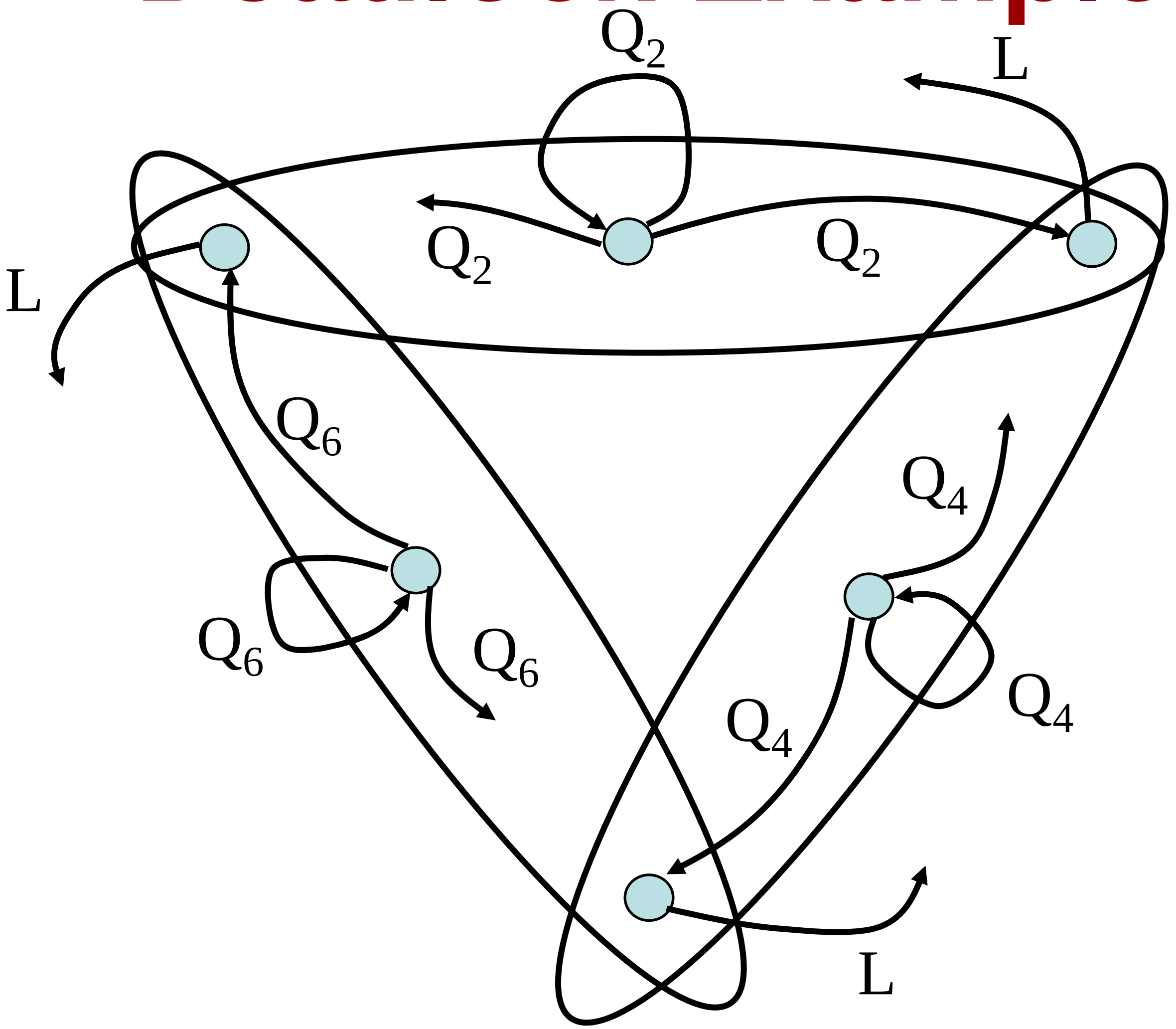
voted := TRUE;

else

voted := FALSE;

end if

Deadlock Example



Concurrent request ⇒ common processes votes to left most quorum ⇒
Circular wait possible ⇒ deadlock possible

Comparison

Algorithm	Messages per entry/exit	Synchronization Delay (seq. msgs)	Problems
Centralized	3	2	Coordinator crash
Token ring	1 … ∞	0..n-1 (Avg: n/2)	Lost token, process crash
Ricart & Agrawali	2 (n – 1)	1	Crash of **any** process
Maekava voting	$3\sqrt{N}$	2	Crash of process in voting set

A comparison of mutual exclusion algorithms.

Notice: the system may contain a remarkable amount of sharable resources!

Summary

- All distributed algorithms suffer badly in event of crashes.
- Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system.

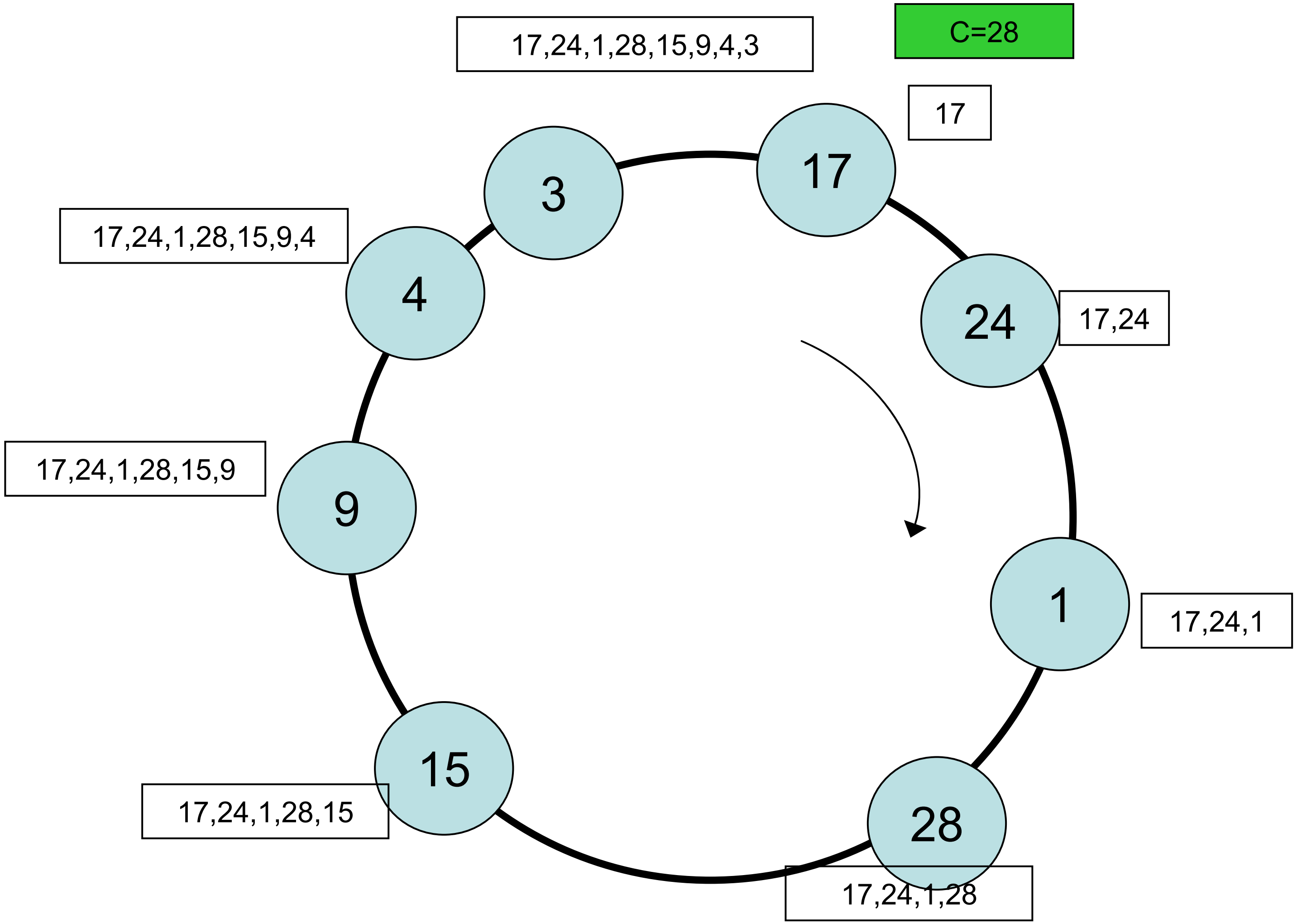
Election Algorithms

- Need:
  - computation: a group of concurrent processes
  - algorithms based on the activity of a special role (coordinator, initiator)
  - election of a coordinator: initially or after some special event (e.g., the previous coordinator has disappeared)
- Premises:
  - each member of the group
    - knows the identities of all other members
    - does not know who is up and who is down
  - all electors use the same algorithm
  - election rule: the member with the highest process id

Election Requirements

- **E1:** *(safety)* A participant process p_i has either $elected_i = \perp$, or $elected_i = p$, where *p* is the *non-crashed* process having the largest process identifier
- **E2:** *(liveness)* All processes p_i participate and will at some point in time set their $elected_i$ variable to a value different from $\perp$ or crash
- $\perp$ = undefined

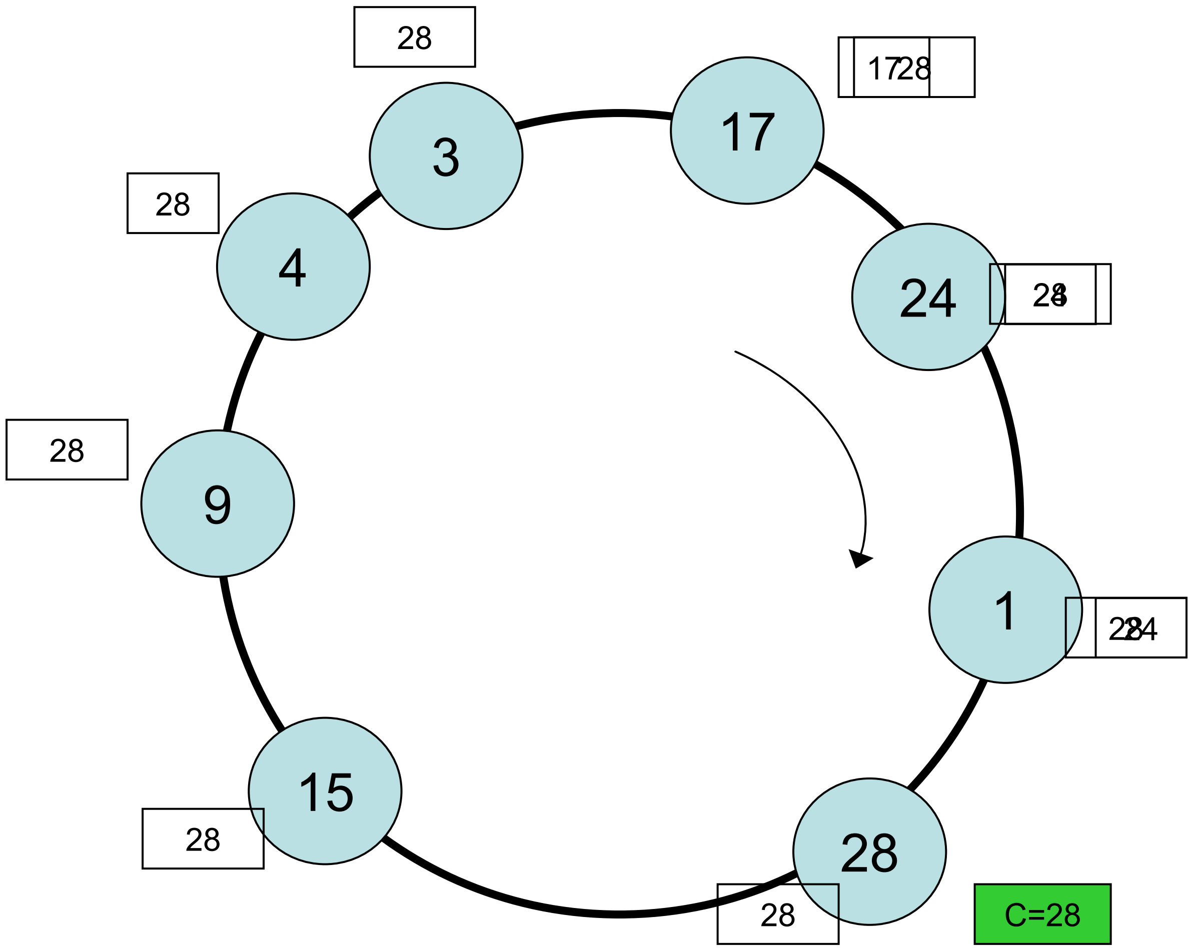
A ring-based election 1



Chang-Roberts

- Improvement Idea:
  - When a node receives a token with smaller id than itself, why should it keep forwarding it?
    - It is a waste, we know that that id will never win!
    - Lets drop tokens with smaller ids than ourselves!
  - Mark nodes that has already participated in an ongoing election to kill concurrent elections
  - A process declares itself elected when it receives its own ID back

Chang-Roberts



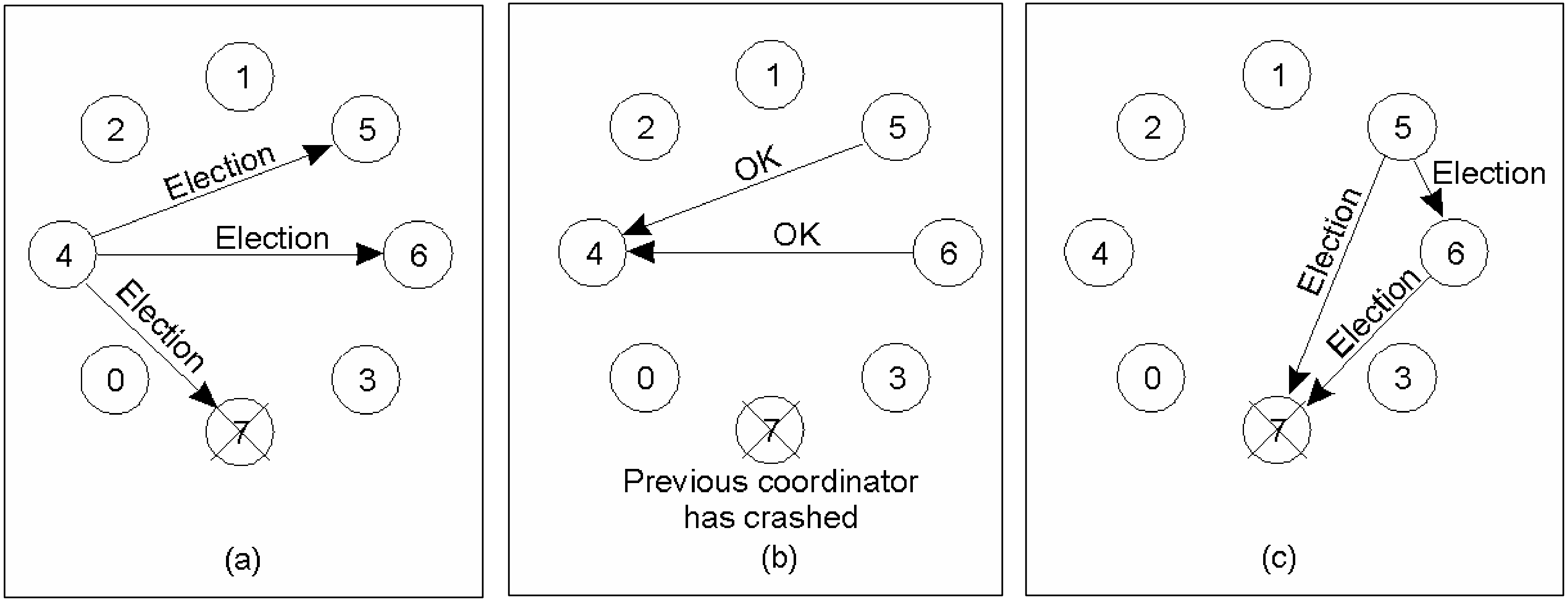
Performance

- Bandwidth: 3N-1
  - N-1 in worst case to reach process with highest ID +
  - One round of N messages before node with highest ID can announce it is a winner +
  - One round of N messages to inform other nodes about coordinator
- Turnaround: an election takes sequential 3 rounds

Bully Algorithm

- Bully
  - *A person who is habitually cruel, especially to smaller or weaker people*
- Processes may fail during election
- Uses timeout to detect failure ($\Rightarrow$assumes synchronous system)
- Each process knows processes with higher ID's
- 3 message types
  - A process sends *Election* to all processes with larger IDs to start an election
  - *Answer (OK):* to election message tells receiver that sender is alive and that receiver must shut-up
  - *Coordinator:* inform about new coordinator

The Bully Algorithm (2)



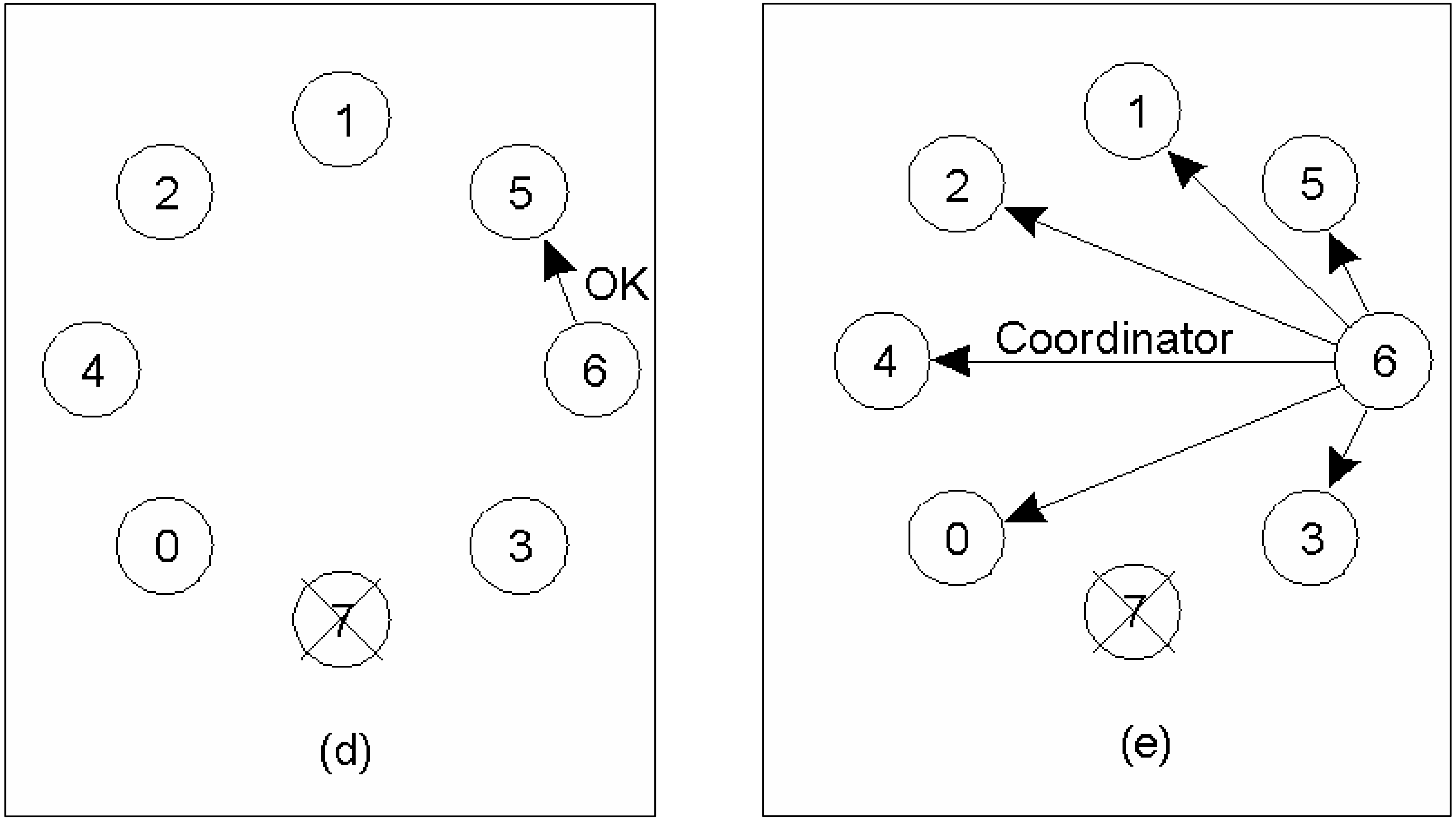
Coordinator id 7 is dead

(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

The Bully Algorithm (3)

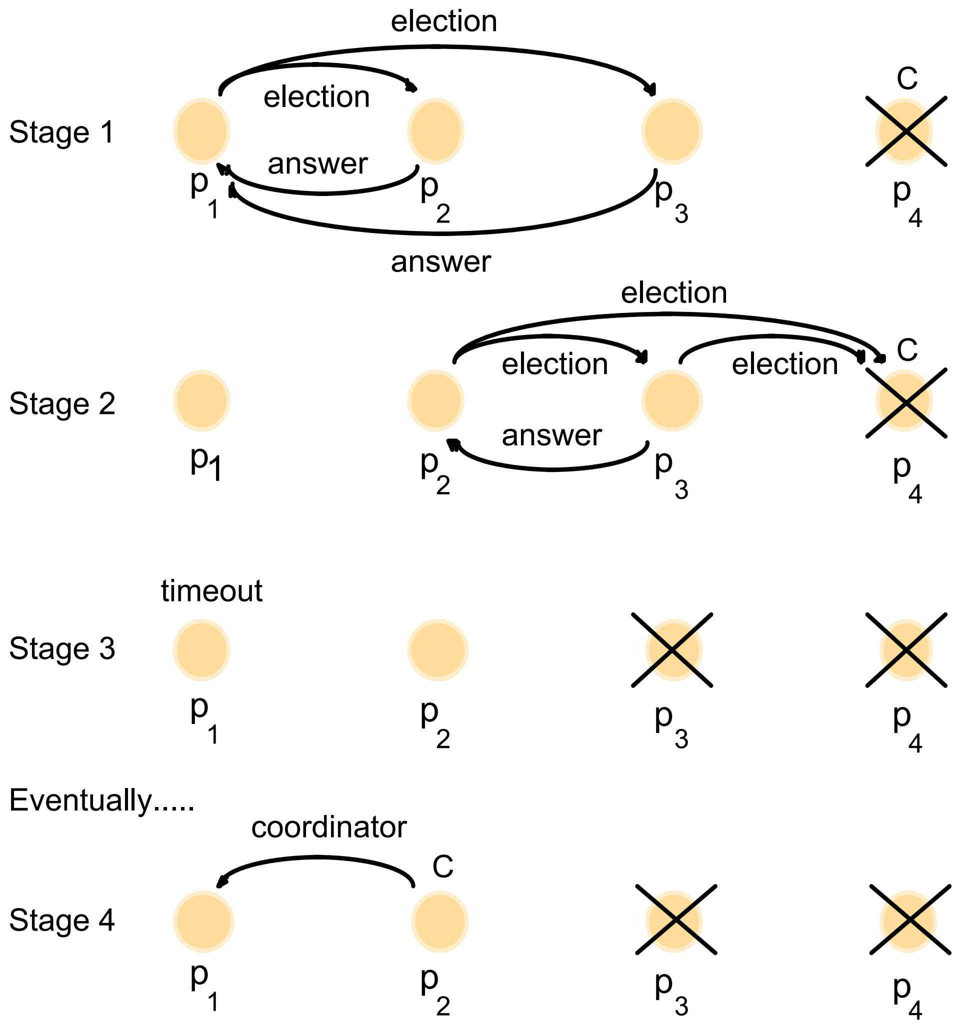


(d) Process 6 tells 5 to stop

(e) Process 6 wins and tells everyone

The bully algorithm

- P_1 detects crash of coordinator p_4
- P_1 decides to hold an election
- P_2 and p_3 tells P_1 to shut up and hold their own (concurrent) elections
- p_3 tells P_2 to shut up
- p_3 times out waiting from answer from P_4 and declares itself the coordinator
- Alas, P_3 fails
- P_1 times out waiting for coordinator and decides to hold an election
- P_2 starts an election and realizes that it is largest living process and declares itself the coordinator p_2,



END