Coordination

Marco Aiello

RuG

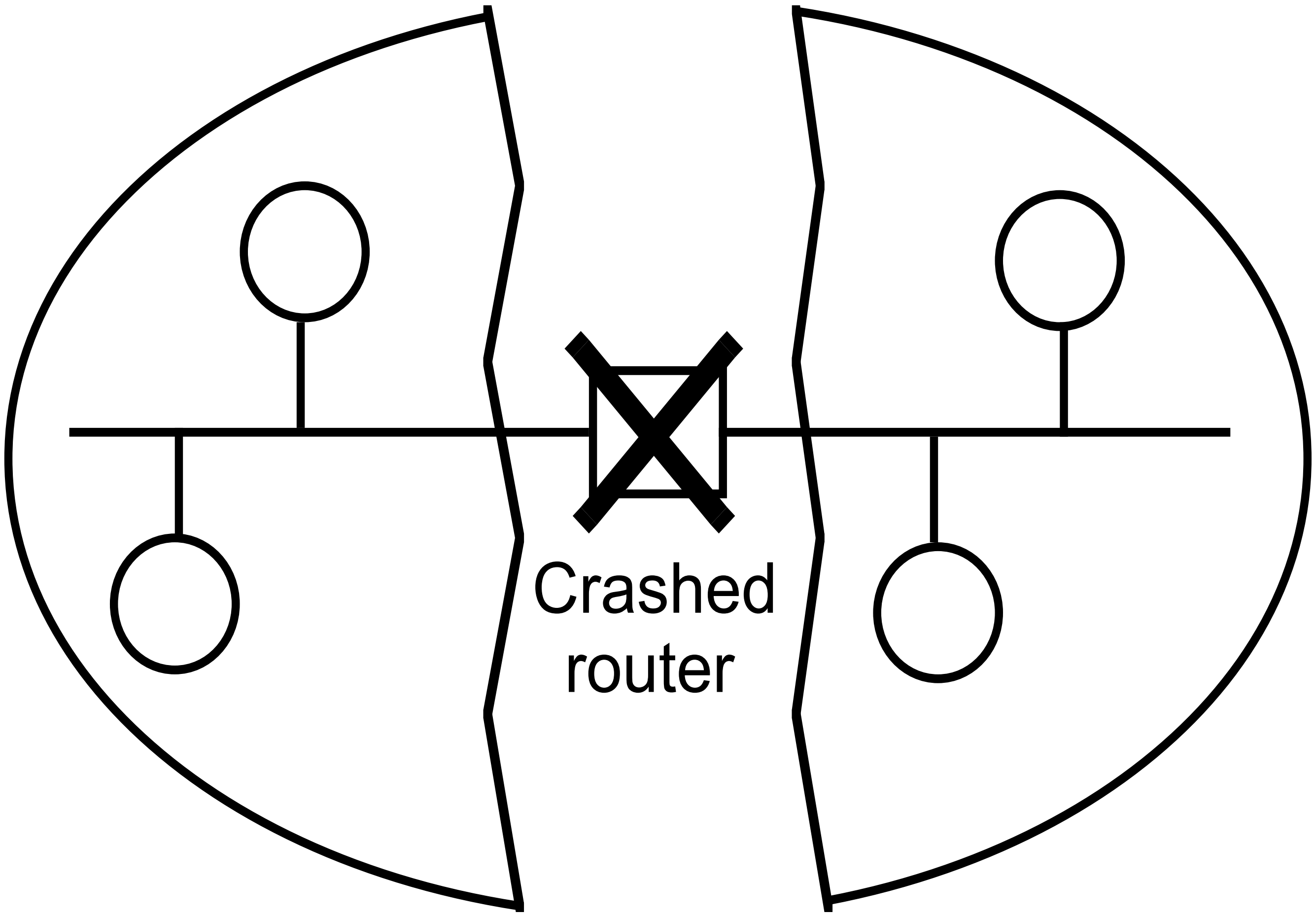
based on www.cdk4.net and

www.cs.uic.edu/~ajayk/DCS-Book

Failure assumptions and detection

- We assume reliable channels:
  - No network partitions
  - Asymmetric communication (p send to q, but not viceversa)
  - Intransitive communication (p send to q, q send to r, does not mean p sends to r)
- We have failure detectors
  - Reliable vs. unreliable
  - Suspected vs. unsuspected failures
  - Use of timeouts for suspected failures

A network partition



Distributed mutual exclusion

- Critical section problem as in OS
- Distributed mutual exclusion (message based)
- Does one have file locking in NFS? (Unix *lockd*)

- Algorithms for mutual exclusion:
  - P_i i=1…N process that do not share variables
  - One single critical section
  - Asynchronous at-most-once message passing
  - Process do not crash

Types

- Token based approaches
- Non-Token based approaches
- Quorum-based approaches

Distributed mutual exclusion

Operations for critical section:

`enter()` `resourceAccess()` `exit()`

Requirements

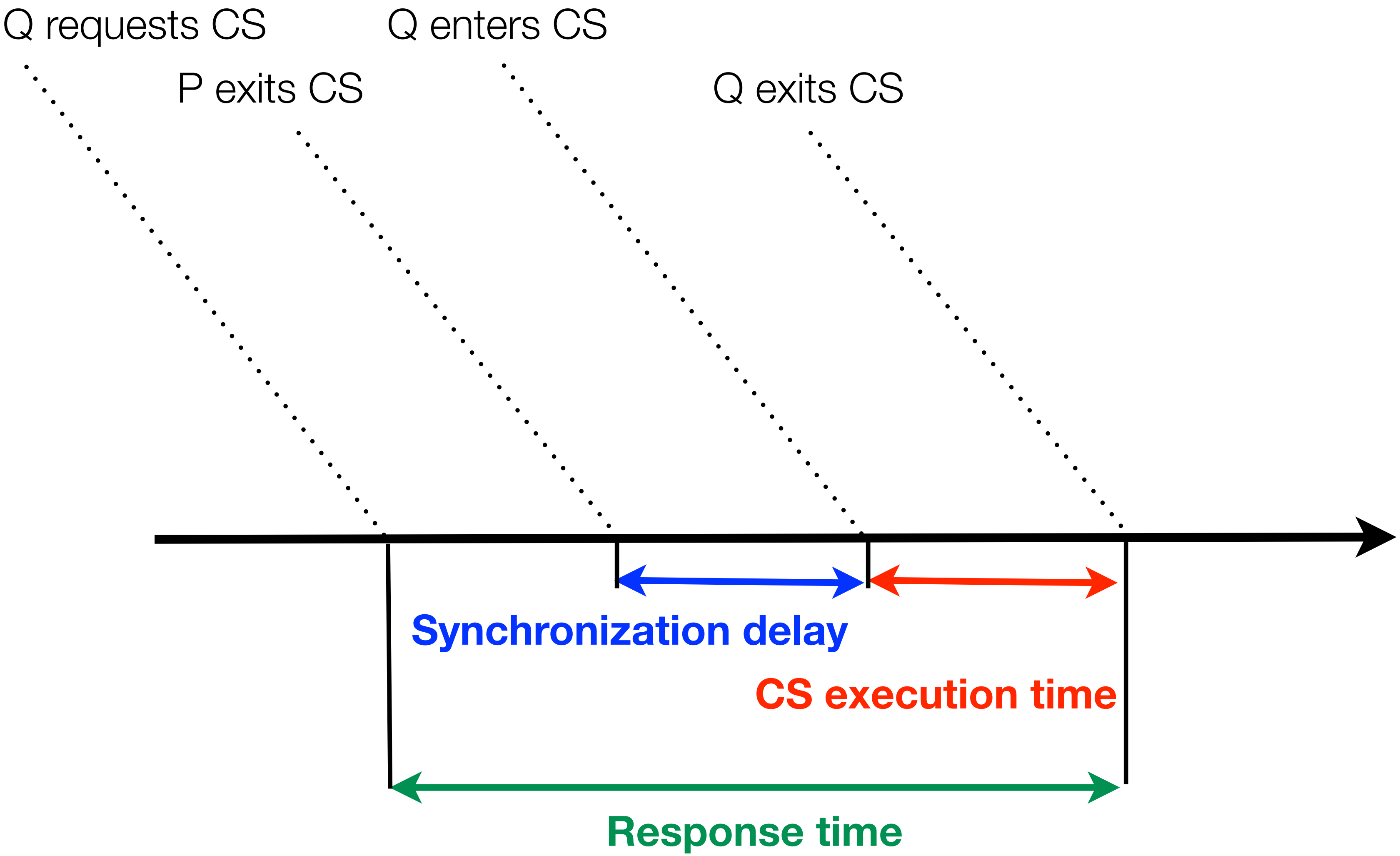
1. *Safety*: at most one process may execute their critical section
2. *Liveness/Progress*: requests to enter critical section eventually succeed (no deadlocks nor starvation)
3. *Ordering/Fairness*: requests for entering critical sections are served with an ordering policy (usually FIFO)

Performance of Algorithms

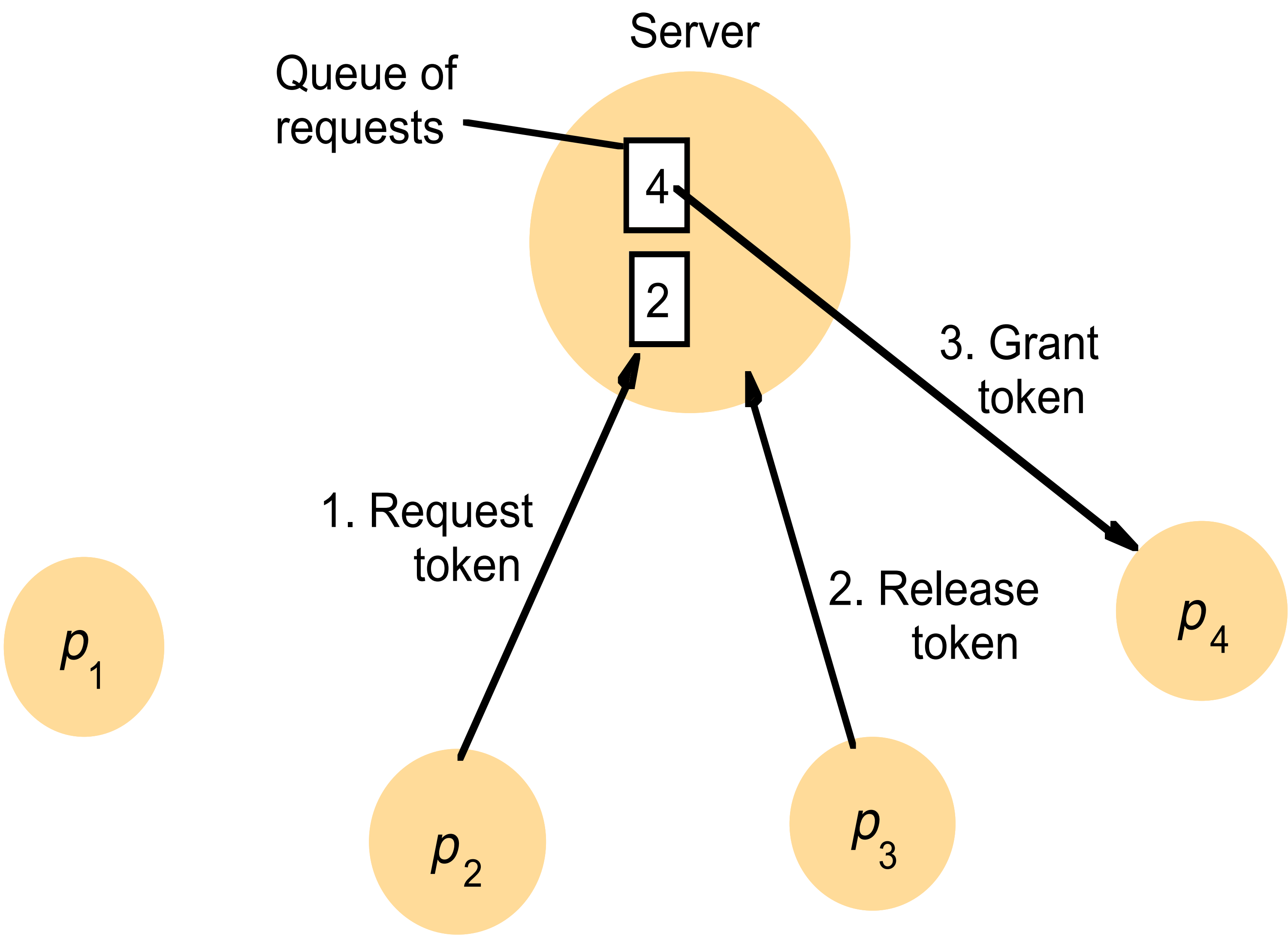
- **Message Complexity:** in the number of exchanged messages by N processes
- **Synchronization Delay (SD):** time for next P to enter a CS once the current holder leaves it. T is the average message delay.
- **Response Time:** time between CS request and end of CS execution
- **System Throughput:** rate at which CS requests are executed $= \frac{1}{SD + E}$
  where E is the average CS execution time

Low vs high load performance **AND** Best and worst case performances

Performance illustration

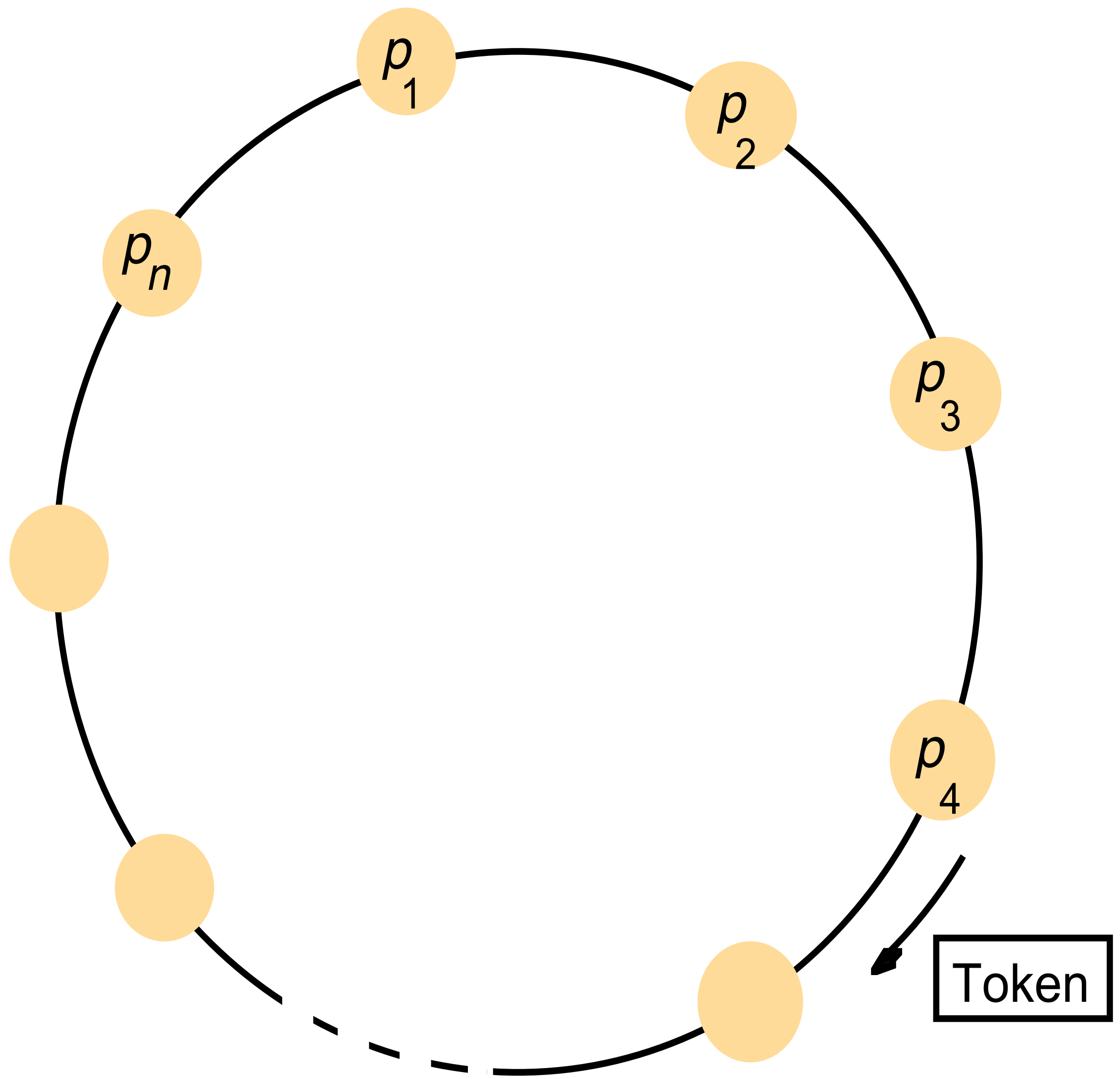


Central server algorithm (token based)



- Safety and progress are guaranteed, what about ordering?
- Bandwidth: 2 messages for request+1 for release
- The server may become a bottleneck

Ring algorithm (token based)



Ring Algorithm

- What is the message complexity, sync delay, response time, throughput of the algorithm?

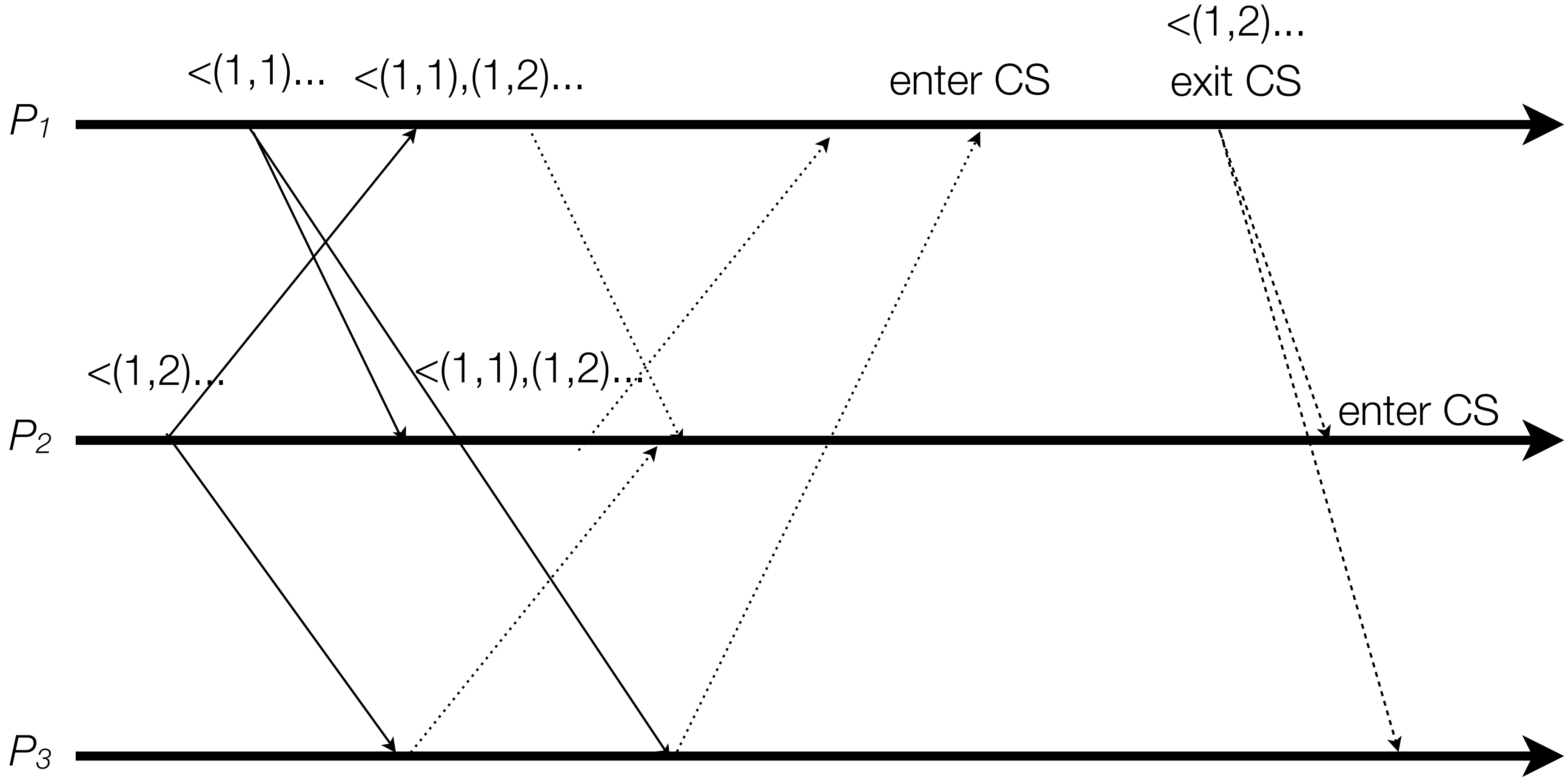
- Are safety, progress and ordering guaranteed?

Assumes FIFO channels

Lamport’s Algorithm (non-token based)

- P_i Request:
    - Broadcast *request(ts_i, i)* and put request in *$request\_queue_i$*
- P_j Receives it:
    - place request on *$request\_queue_j$* and reply with a timestamp
- P_i enters CS when:
    - It has received a message with larger timestamp than *(ts_i, i)* from all others
    - P_i request is at the top of *$request\_queue_i$*
- P_i exists CS:
    - remove from *$request\_queue_i$* and broadcast *release*
    - each process removes the request from the queue upon *release* receipt

An example



Lamport Algorithm

- Correctness
- Liveliness
- Fairness

(proof by contradiction looking at the conditions that must hold for P_i to enter the CS and the properties of logical clocks)

3(N-1) per CS invocation, Sync delay is T

Ricart and Agrawala's algorithm

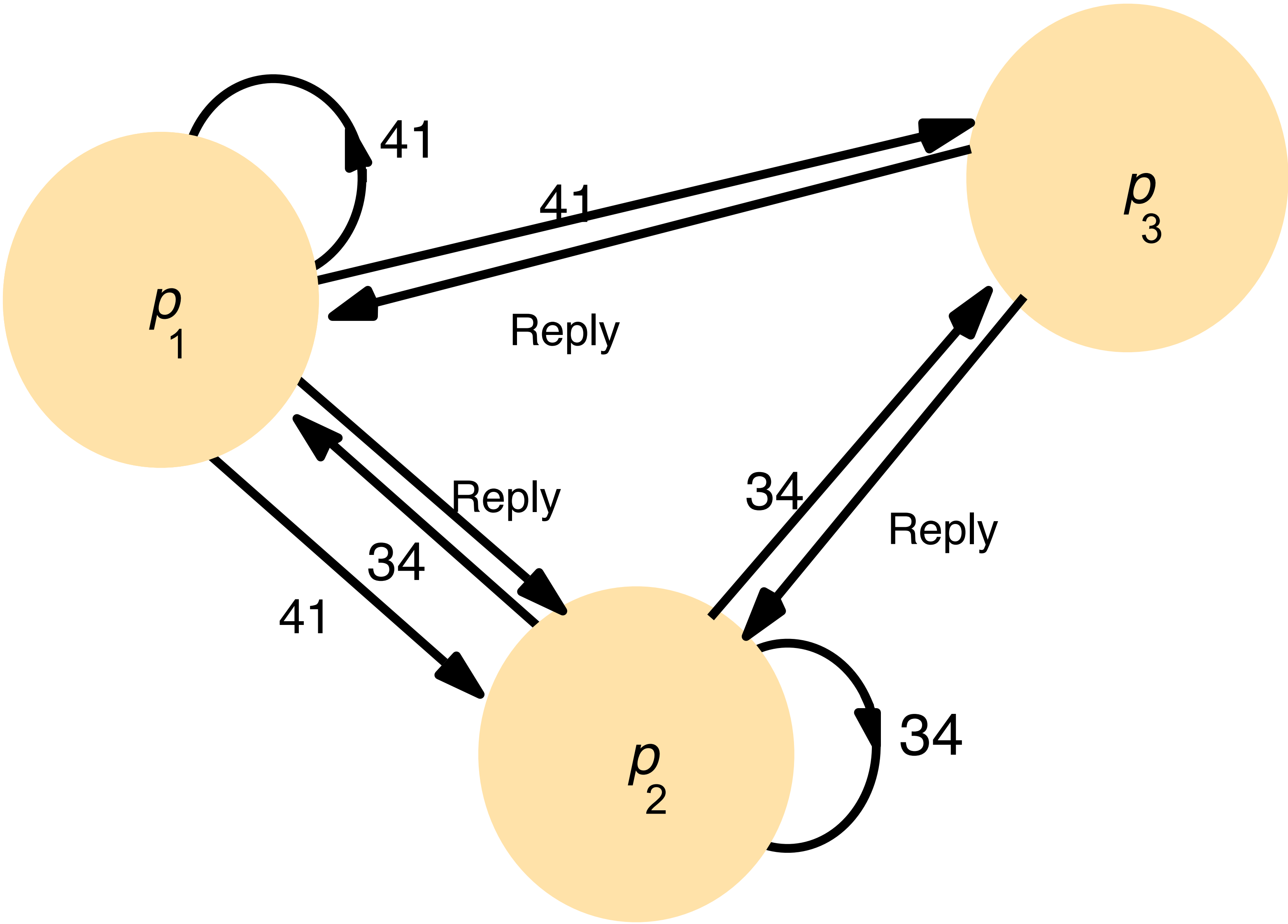
- N peer process synchronize using multicast
- Replies to all message
- Does not require FIFO channels
- Using Lamport's logical clock

Ricart and Agrawala's algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* 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* $(T, p_j) < (T_i, p_i)$))
    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



Ricart and Agrawala's algorithm

- Safety, progress and fairness are met
  - (proof by contradiction)

- High bandwidth consumption:
  - 2(N-1) message passed
- Client delay
  - A round trip time of a message
- Sync delay is T

Maekawa’s algorithm (Quorum based)

- It is not necessary to have replies of all peers, one can use a voting mechanism
- A voting set V_i associated with each process

  $V_i \subseteq \{p_1, p_2, \ldots, p_N\}$

  For all i and j

  - $p_i \in V_i$
  - $V_i \cap V_j \neq \emptyset$ there is at least a common member
  - $|V_i| = K$ all voting sets are of the same size
  - Each process is contained in M of the voting sets

Maekawa's algorithm – part 1

On initialization

state := RELEASED;

voted

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

Maekawa’s algorithm

- Optimal solution is with K close to $\sqrt{N}$ and M=K
- How to decide who to put in the voting sets?
- Safety property satisfied, but not progress and ordering
- Using a timestamped queue, the problem can be solved
- Bandwidth = $3\sqrt{N}$ (2 per entry and 1 for exit)
- Client delay = roundtrip time of a message

Agarwal-El Abbadi Algorithm (Quorum based)

- Using Tree quorums
- Trees are complete binary ones (any node can be a root to build the tree)
- 2^k leaves in the tree with $k \sim O(\log n)$
- Algorithm starts with a defined root
- A quorum set is a path root-leaf

Agarwal-El Abbadi Algorithm (Quorum based)

```
quorumSet GetQuorum(tree, networkHierarchy)
    var left,right,quorumSet
    if (tree is empty) then
        return {}
    elseif GrantsPermission (tree↑.Node) then
        return ((tree↑.Node) ∪ GetQuorum(tree↑.LeftChild))
        OR
        return ((tree↑.Node) ∪ GetQuorum(tree↑.RightChild))
        else
            left←GetQuorum(tree↑.left)
            right←GetQuorum(tree↑.right)
            if (left or right are empty) then
                return error
            else
                return (left ∪ right)
```

TRUE WHEN NODE AGREES TO BE IN THE QUORUM

FAILED TO FIND A PATH E.G. DUE TO NODE FAILURE

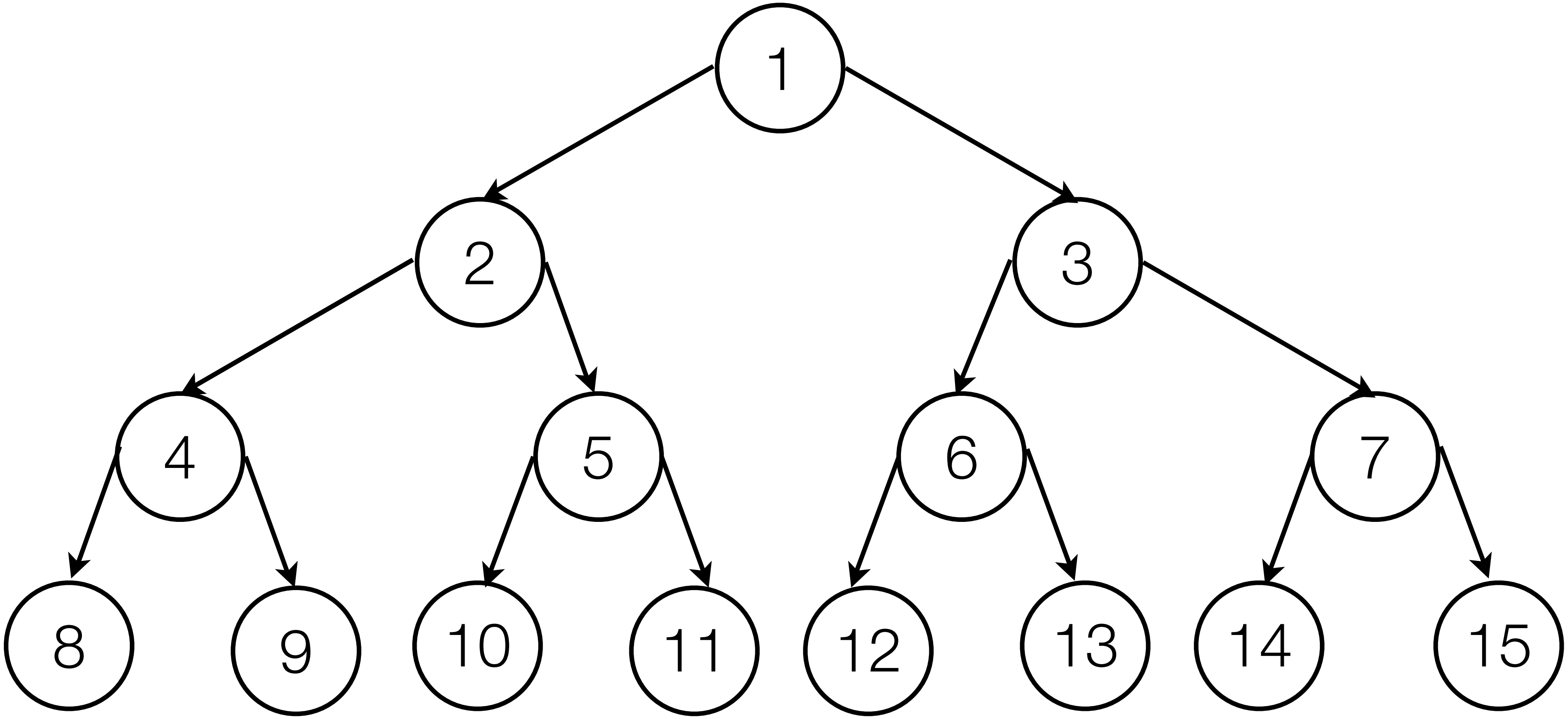
TAKE TREE↑.NODE CHILDREN IN THE QUORUM/PATH

IF LEAF IS DOWN, NO QUORUM CAN BE FORMED

Performance

- best case log n
- worst case $O((n+1)/2)$
- can tolerate up to n-log n failures

Example



Quorum sets: 1-2-4-8 1-2-4-9 1-2-5-10 1-2-5-11 etc.

If 2 fails: 1-4-8-5-10 1-4-9-5-11 1-4-9-5-10 etc.

Mutual exclusion with tree quorum sets

1. P send ***request*** to all members of the quorum sets it belongs to
2. Each process stores requests in a *queue*
3. Each site replies to the head of its *queue* only
4. If a site gets replies from all members of its quorum sets it enters the CS
5. Existing the CS, it sends a *relinquish* message, all members remove the request from their queues
6. If a request arrives with a timestamp smaller than the head of the queue, an *inquiry* message is sent to the process at the head of the queue and the process waits for a *yield* or a *relinquish* message
7. Upon receipt of an *inquiry* message: if it has received all replies, it ignores it; otherwise it sends a *yield* message to the sender
8. Upon receipt of a *yield* message, the sender's request is put at the head of the queue and a reply is sent

Mutual exclusion with tree quorum sets

- Correct whenever the quorum sets are built with the Intersection property (such as quorum trees)

- **Example.**
  - Given the sets: 1-2-3   2-4-5   4-1-6
  - Suppose 3, 5 and 6 want to enter CS
  - They send messages to {1,2},{2,4} and {1,4}, respectively
  - At 2, request of 3 arrives before that of 5, then 2 acks 3 and rejects 5
  - At 1, request 3 arrives before that of 6, then 1 acks 3 and rejects 6
  - 3 can enter safely the CS

Leader Election

Elections

- *Election*: algorithm to choose a process to play a particular role in a distributed system
- A process *calls* an election
- Processes *participate* or not in an election
- Vote identifier: <vote,pid>
- Requirements
  - *Safety*: a particpant process p_i has $elected_i = \perp$ or $elected_i = P$
  - *Liveness*: all processes p_i partecipate and evenutally set $elected_i \neq \perp$

The bully algorithm

- Assumptions: process can crash, message delivery reliable and synchronous, complete knowledge of vote_id of all other peers
- Synchronous: message turnaround time bound by $T = T_{transmission} + T_{process}$
- Three types of messages:
  - *election* message: to call elections
  - *answer* message: to vote
  - *coordinator* message: to announce own acting as coordinator

The Bully Algorithm (Garcia Molina, 1982)

- P broadcasts an election message (inquiry) to all other processes with higher process IDs.
- If P hears from no process with a higher process ID than itself, it wins the election and broadcasts victory.
- If P hears from a process with a higher ID, P waits a certain amount of time for that process to broadcast itself as the leader. If it does not receive this message in time, it re-broadcasts the election message.
- If P gets an election message (inquiry) from another process with a lower ID it sends an "I am alive" message back and starts new elections.

The Bully Algorithm

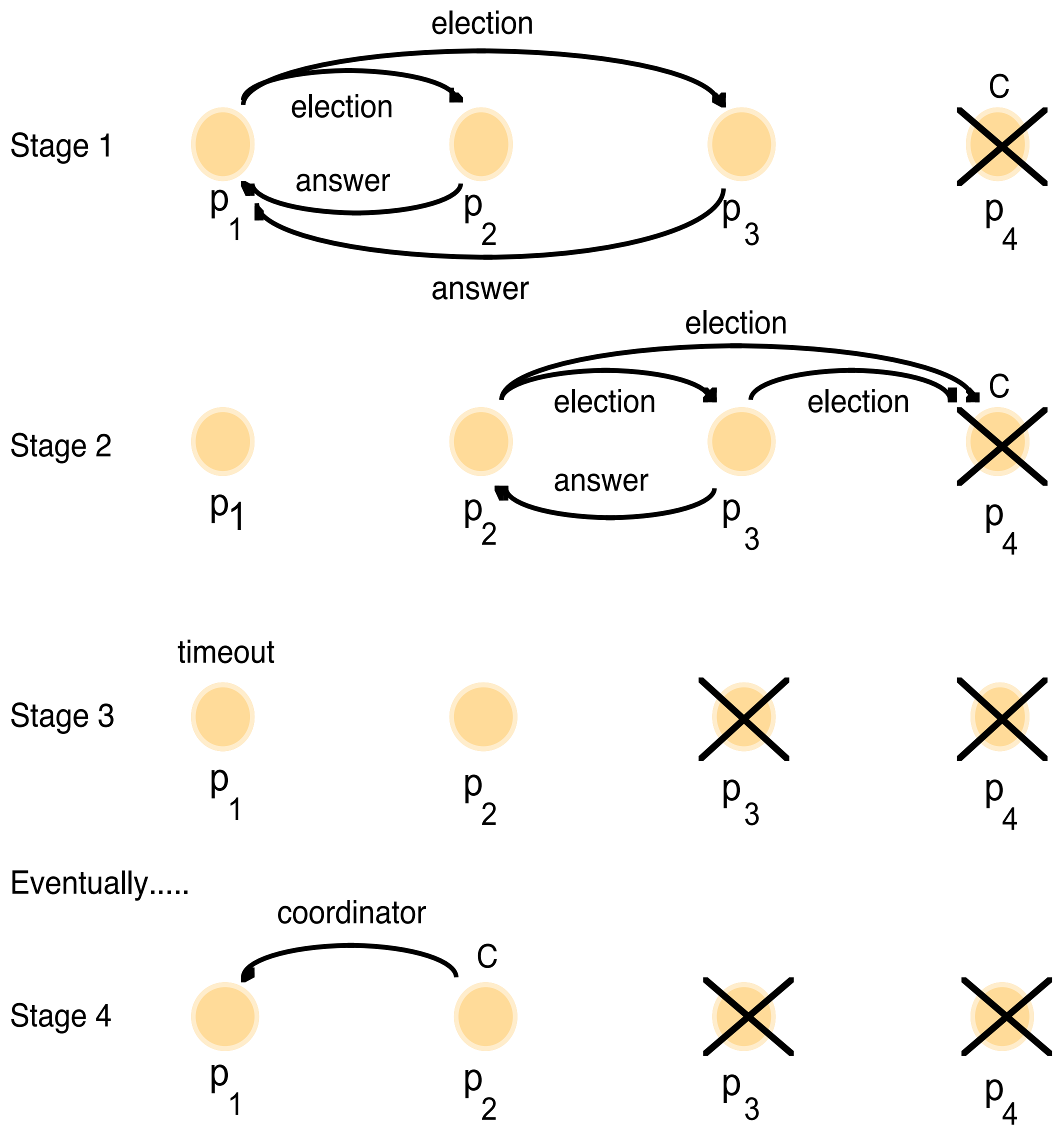
- Note that if P receives a victory message from a process with a lower ID number, it immediately initiates a new election. This is how the algorithm gets its name - a process with a higher ID number will bully a lower ID process out of the coordinator position as soon as it comes online.

The bully algorithm

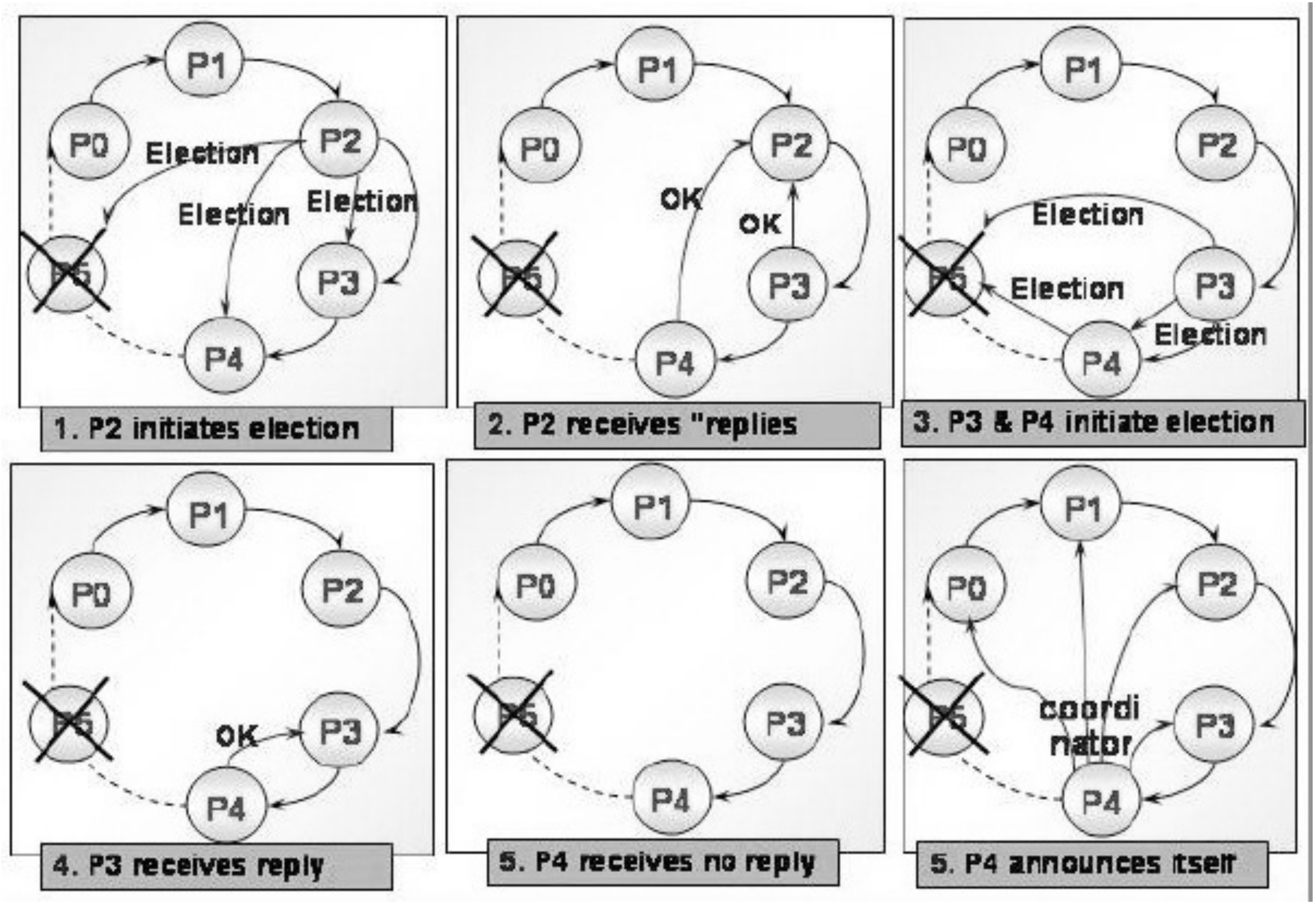
- The process with highest vote_id sends coordinator message
- A process needing a coordinator sends an election message, if no answer within time T, then it sends a coordinator message
- If a process receives a coordination message, it sets its elected variable
- If a process receives an election message, it answers and begins another election (if needed)
- If a new process starts to coordinate (highest vote_id), it sends a coordinator message and "***bullies***" the current coordinator out

The bully algorithm

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



The Bully Algorithm



The bully algorithm

- Liveness condition is met by the reliable and bounded message transmission assumption
- Safety is met only if no new process starts during an election (because there is no guarantee on order of message delivery if two processes send coordinator messages concurrently which may happen in case e new process starts)
- Bandwith
  - $O(N^2)$ worst case, i.e., the process with least vote detects the failure of the coordinator
- Turnaround
  - One message