TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Mutual Exclusion and Election in Distributed Systems

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Main properties of algorithms for concurrent and distributed systems

- **Safety**: nothing bad will happen
  - Undesirable states will not occur
  - Example: If two processes attempt to write to the same resource at the same time, no data corruption or race conditions happen
- **Liveness**: something good will eventually happen
  - The system will make progress and avoid situations where it stops (deadlock or starvation)
  - Example: if a process requests a resource, it will eventually get it

Mutual exclusion

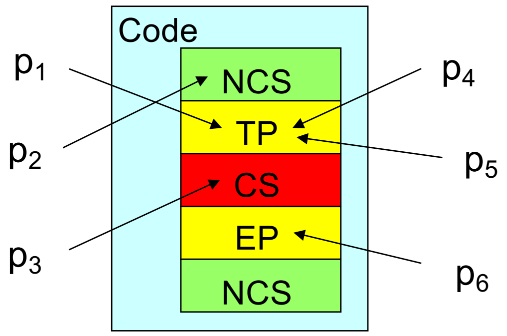
- N processes want to access a shared resource
- Goal: ensure that each process can exclusively access a shared resource without interference from others
- Components of mutual exclusion algorithm:
  - Critical section (CS): the part of the code where a process accesses the shared resource
  - Trying protocol (TP): instructions that occur before entering the CS, ensuring no conflicts
  - Exit protocol (EP): instructions that occur after exiting the CS, releasing the resource for others

Properties of mutual exclusion algorithms

- **Mutual exclusion (ME)** or **safety**
  - At most one process at a time can execute in CS
- **No deadlock (ND)**
  - If one or more processes are blocked in their TS, at least one process will eventually enter and exit the CS
- **No starvation (NS)** or absence of indefinite postponement
  - No process remains blocked forever in the TS; every request to enter the CS is eventually satisfied
- Observations:
  - NS ⇒ ND, but ND ⇏ NS
  - NS and ND are liveness properties
  - NS is also a fairness property: every process eventually gets access
- **Ordering**
  - Requests to enter the CS are served in order of arrival (according to the happened-before relation)
  - Stronger fairness than NS: ordering ⇒ NS, but not viceversa

Mutual exclusion in concurrent systems

- Mutual exclusion originated in concurrent systems
- Early mutual exclusion algorithms are based on the use of shared variables to coordinate *N* processes
- Dijkstra's algorithm (1965)
  - Designed for single-processor systems
  - Guarantees ME and ND, does not guarantee NS
- Lamport's bakery algorithm (1974)
  - Designed for shared-memory multiprocessor systems
  - Guarantees ME, ND, and NS



Lamport's bakery algorithm

- Solution inspired by a real-world situation
  - Waiting to be served at a bakery
- Assumptions on concurrent system model
  - Processes communicate by reading and writing to shared variables
  - Reading and writing to a shared variable are not atomic operations
    - A process may write while another is reading the same variable
  - Each shared variable is owned by a process:
    - Everyone can read it, but only the owner can write to it
    - No process can perform two writes simultaneously
  - The execution speeds of processes are not correlated
- Shared variables:
  - `num[1,…,N]`: array of integers, initialized to `0`
  - `choosing[1,…,N]`: array of booleans, initialized to false
- Local variable: `j`: integer in the range `[1,…,N]`

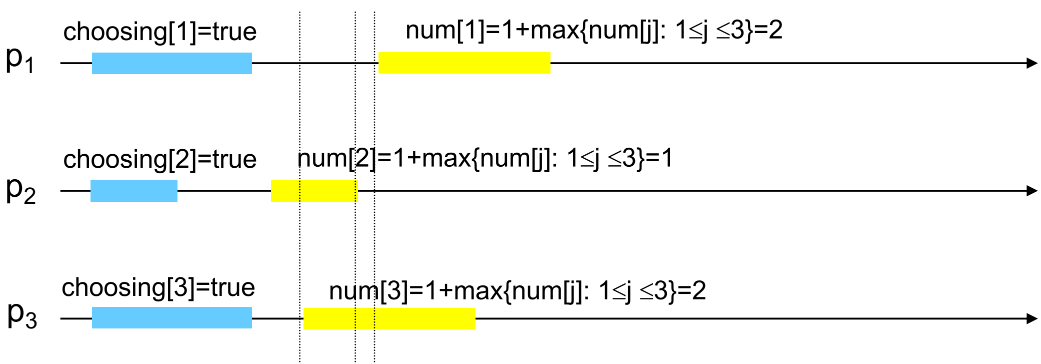
Lamport's bakery algorithm

```
// non-critical section

// take a ticket                                              doorway
choosing[i] = true;        // start choosing a ticket
num[i] = 1 + max(num[x] : 1 ≤ x ≤ N);
choosing[i] = false;       // finish choosing the ticket

// wait until its number is called, comparing with others    bakery
for j = 1 to N do
    // busy waiting while process j is choosing
    while choosing[j] do NoOp();

    // busy waiting until pᵢ has the smallest number
    // ties are broken by favoring the process with smaller id
    while num[j] ≠ 0 and {num[j], j} < {num[i], i} do NoOp();

// critical section

num[i] = 0;  // release the ticket
// end of critical section
```

Precedence relation < on ordered integer pairs:
{*a*,*b*} < {*c*,*d*} if a < c OR if *a* = *c* AND *b* < *d*

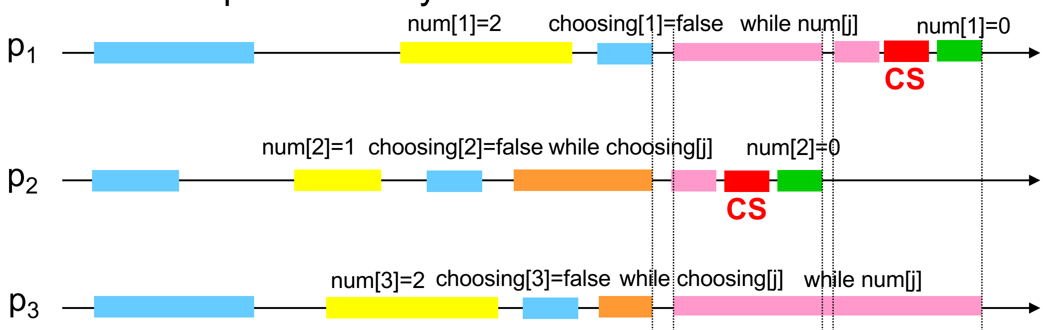
Lamport's bakery algorithm

- Doorway
  - When p_i starts the EP, it notifies the other processes by setting `choosing[i]`
  - p_i takes a ticket number equal to one plus the maximum of the ticket numbers chosen by the other processes
  - Other processes may concurrently enter the doorway (ticket selection phase)
  - Example of doorway execution



Lamport's bakery algorithm

- Bakery
  - p_i must check that it is next in line among the processes waiting to enter the CS
  - The first while loop allows all processes in the doorway to finish choosing their ticket
  - The second while loop keeps p_i waiting until:
    - Its ticket number becomes the smallest among all waiting processes
    - Any process with the same ticket number has a larger process ID
  - Observation:
    - Cases where the same ticket number is chosen are resolved using the process ID as a tie-breaker
  - Example of bakery execution



Lamport's bakery algorithm: properties

- ME property
  - Ensured because if p_i is in the doorway and p_j is in the bakery, then *{num[j], j} < {num[i],i}*
- NS property
  - No process waits forever, because eventually its ticket number becomes the minimum
- Ordering
  - If p_i enters the bakery before p_j enters the doorway, then p_i will enter the CS before p_j

Distributed mutual exclusion: model

- Communication
  - Processes communicate via message passing; cannot directly access each other's variables
  - A process sends a request and receives a reply with the value
  - Message transmission delays are unknown but finite
  - Reliable channels: messages are delivered correctly, FIFO-ordered, with no duplicates or spurious messages (received but never sent)
- System
  - N processes p_i (i = 1, …, N)
  - Asynchronous
  - Processes are fail-free (no crashes)
  - Each process spends a finite time in the CS

Adapting Lamport's bakery algorithm

- Adapting Lamport's bakery algorithm to DS

  Each process p_i acts as a server for its own local variables num[i] and choosing[i]

  Communication is via message passing: processes read other processes' local variables using request-reply messages
- Works correctly, but…
  - ✗ High communication cost to enter CS: 6N messages
    - Must read 3N variables (N for doorway, 2N for bakery)
    - Each variable read requires 2 messages
  - ✗ Latency depends on the slowest process-channel combination
  - ✗ Efficiency and scalability are limited
    - No cooperation among processes

Distributed ME: considered algorithms

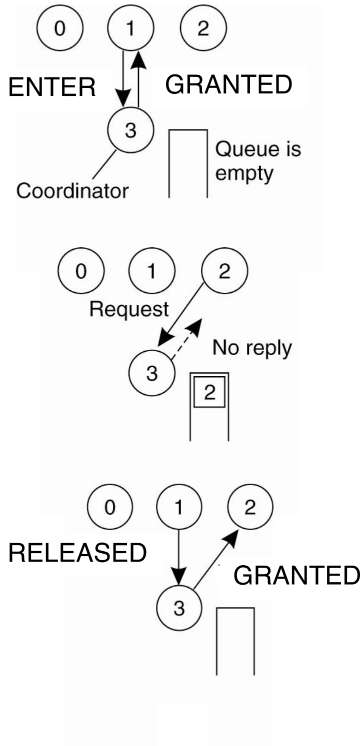
1. **Permission-based** algorithms
   – A process requesting access to the shared resource asks for permission, which can be managed:
     • Centrally (coordinator-based)
     • Distributed (Lamport's distributed algorithm, Ricart and Agrawala's algorithm)
2. **Token-based** algorithms
   – A special message, called a *token*, circulates among processes
     • The token is unique at any time
     • Only the process holding the token can access the shared resource
   – Token management can be distributed
3. **Quorum-based** (voting) algorithms
   – A process requests votes from a subset of processes before accessing the shared resource
   – Example: Maekawa's algorithm

Distributed ME: performance metrics

- Main metrics to evaluate distributed ME algorithms
- Number of messages to enter and exit the CS
  – Indicates the network bandwidth consumption
- Number of messages to enter the CS
  – Indicates the waiting time before a process can enter the CS

Permission-based: centralized algorithm

- Process p_i that requires access to the CS sends an access request (ENTER) to the central coordinator
- If the CS is free, the coordinator informs p_i that access is granted (GRANTED)
- Otherwise, the coordinator enqueues the request using a FIFO policy and informs p_i that access is denied (DENIED)
  - Alternatively, in case of synchronous DS, it does not reply (see figure)
- When p_i releases the resource, it notifies the coordinator (RELEASED)
- The coordinator removes the first pending request from the queue and sends GRANTED to its sender



Centralized algorithm

- Properties
  - Guarantees safety and liveness
    - NS and ND: only if the coordinator and processes do not fail
  - Ordering
    - FIFO ordering is guaranteed according to the order in which requests arrive at the coordinator
    - Not according to the order in which requests are sent or the order in which processes request entry to the CSO
- Pros and cons
  - ✓ The simplest to implement
  - ✓ The most efficient in terms of number of messages
    - Only 3 messages (ENTER, GRANTED, and RELEASE) are required to enter and exit the CS
  - ✗ The coordinator is a SPOF and a potential performance bottleneck
  - ✗ If a process fails while in the CS, RELEASE is lost

Permission-based: Lamport's distributed algorithm

- Each process p_i uses a scalar clock to timestamp messages (plus total ordering $\Rightarrow$) and maintains a local queue
  - The queue stores CS access requests from other processes
  - The scalar clock is incremented before sending messages and after receiving messages

Lamport's distributed algorithm

- Algorithm rules:
  - Requesting CS access: p_i sends a request message REQ(t_i, p_i) with timestamp t_i (its scalar clock) to all other processes and inserts REQ(t_i, p_i) into its local queue
  - Receiving a request: when p_j receives a request from p_i it inserts REQ(t_i, p_i) into its queue and sends an ACK message to p_i
  - Entering the CS: p_i enters the CS if and only if:
    - REQ(t_i, p_i) precedes all other requests in the queue (i.e., {t_i, p_i} is the minimum according to $\Rightarrow$)
    - p_i has received from every other process a message (ACK or REQ) with a timestamp greater than t_i (according to $\Rightarrow$)
  - Exiting the CS: p_i removes REQ(t_i, p_i) from its queue and sends a RELEASE message to all other processes
  - Receiving a RELEASE: when p_j receives a RELEASE from p_i it removes REQ(t_i, p_i) from its queue

Lamport's distributed algorithm

- Similar to the distributed algorithm for totally ordered multicast 🧐
- Guarantees safety, liveness, and ordering
  - Ordering: requests are served according to their timestamp, based on scalar clock, and process id
- Performance: requires 3(N−1) messages to enter and exit the CS:
  - N−1 REQUEST messages
  - N−1 ACK messages
  - N−1 RELEASE messages

Permission-based: Ricart and Agrawala algorithm

- Optimization of Lamport's distributed algorithm
  - Scalar clock (and ⇒) and local queue

- A process that wants to enter the CS sends a REQUEST message to all other processes containing:
  - its own process ID
  - timestamp based on scalar clock
- It then waits for a reply from all other processes
- After receiving all REPLY messages, it enters the CS
- Upon exiting the CS, it sends a REPLY message to the processes with queued requests

- A process that receives a REQUEST message may:
  - Not be in the CS and not want to enter it → sends a REPLY to the sender
  - Be in the CS → does not reply and places the request in its local queue
  - Want to enter the CS → compares its own timestamp and ID with the received timestamp and ID. The smaller pair wins: if the other process wins → sends a REPLY; if it wins → does not reply and places the request in its local queue

Ricart and Agrawala's algorithm

- Local variables for each process
  - #replies: number of REPLY messages received (initialized to 0)
  - State ∈ {Requesting, CS, NCS} (initialized to NCS)
  - Q: queue of pending requests (initially empty)
  - Last_Req: timestamp of the REQUEST message (initialized to 0)
  - Num: scalar clock (initialized to 0)
- Each process p_i executes the following algorithm

Begin

1. State = Requesting;
2. Num = Num+1; Last_Req = Num;
3. for j=1 to N-1 send REQUEST to pj;
4. Wait until #replies=N-1;
5. State = CS;
6. CS
7. ∀ r∈Q send REPLY to r
8. Q=∅; State=NCS; #replies=0;

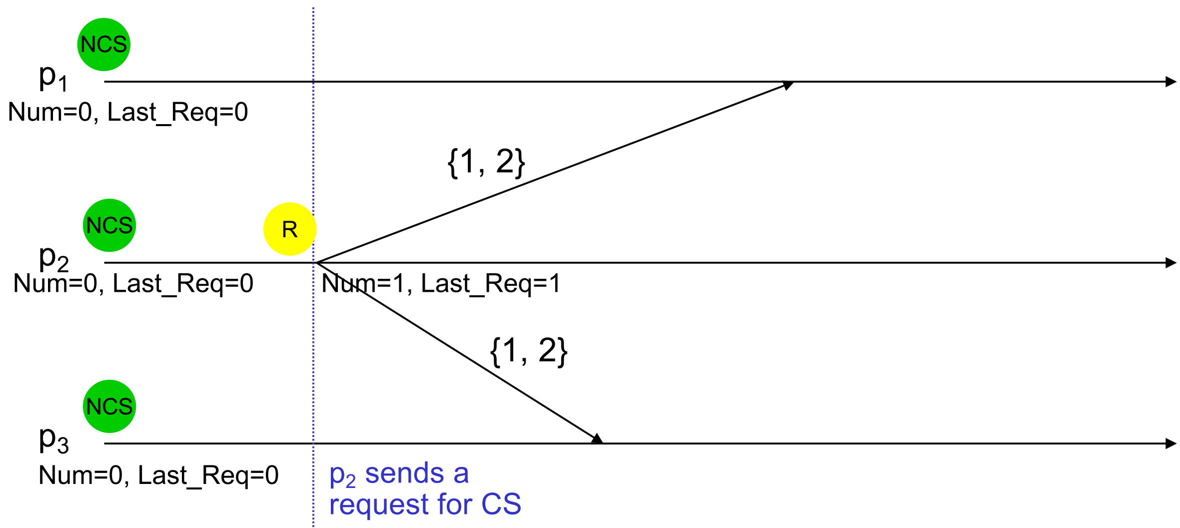
Upon receipt of REQUEST(t) from pj

1. if State=CS or (State=Requesting and {Last_Req, i} < {t, j})
2. then insert {t, j} in Q
3. else send REPLY to pj
4. Num = max(t, Num)

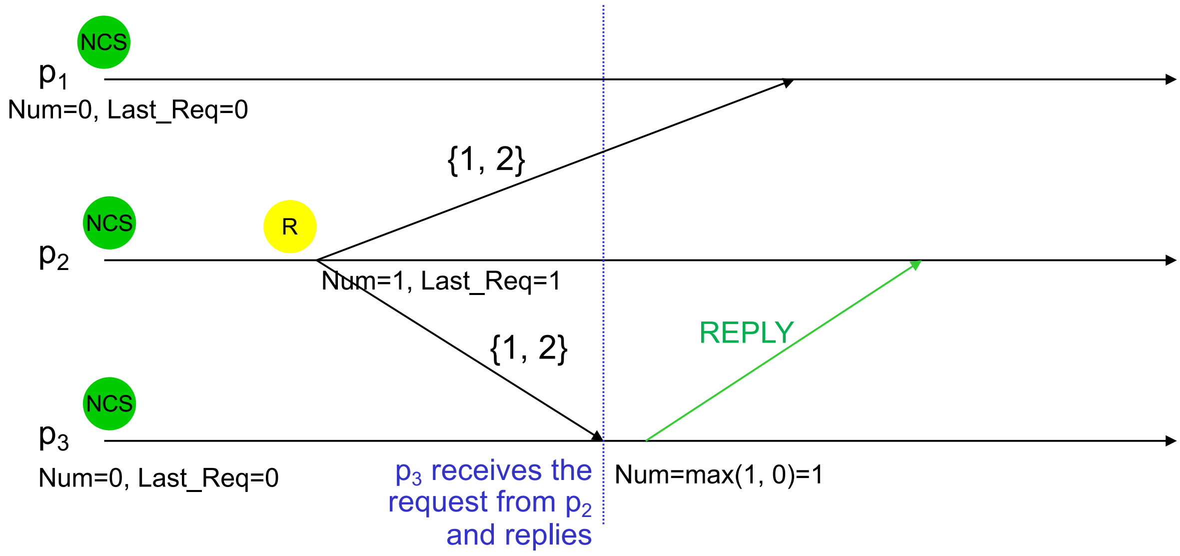
Upon receipt of REPLY from pj

1. #replies = #replies+1

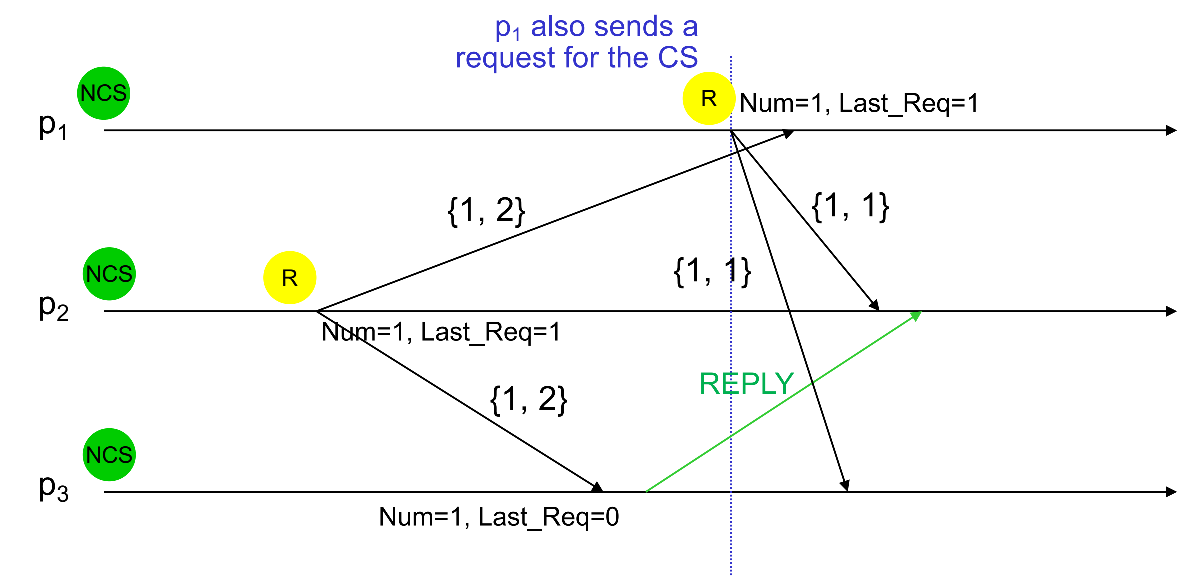
Ricart and Agrawala's algorithm: example



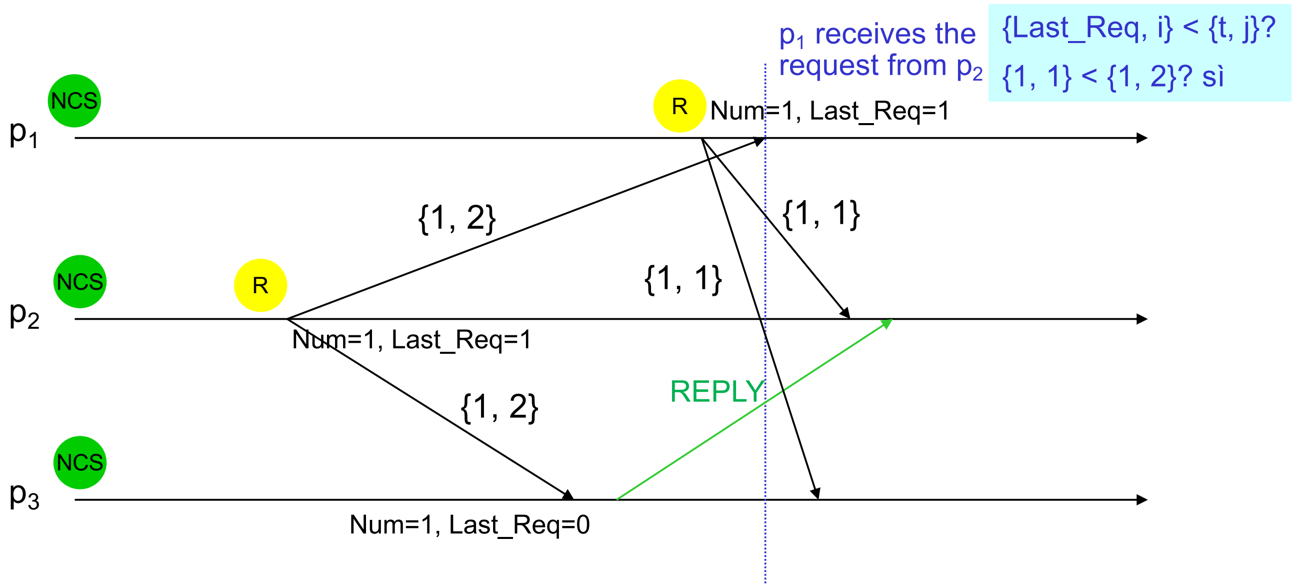
Ricart and Agrawala's algorithm: example



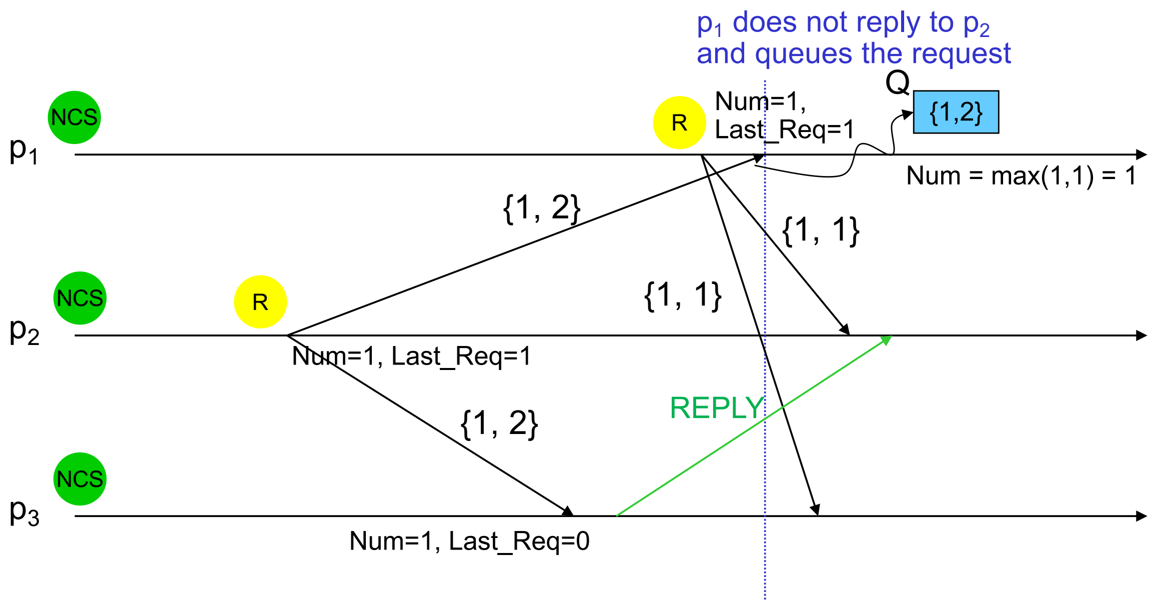
Ricart and Agrawala's algorithm: example



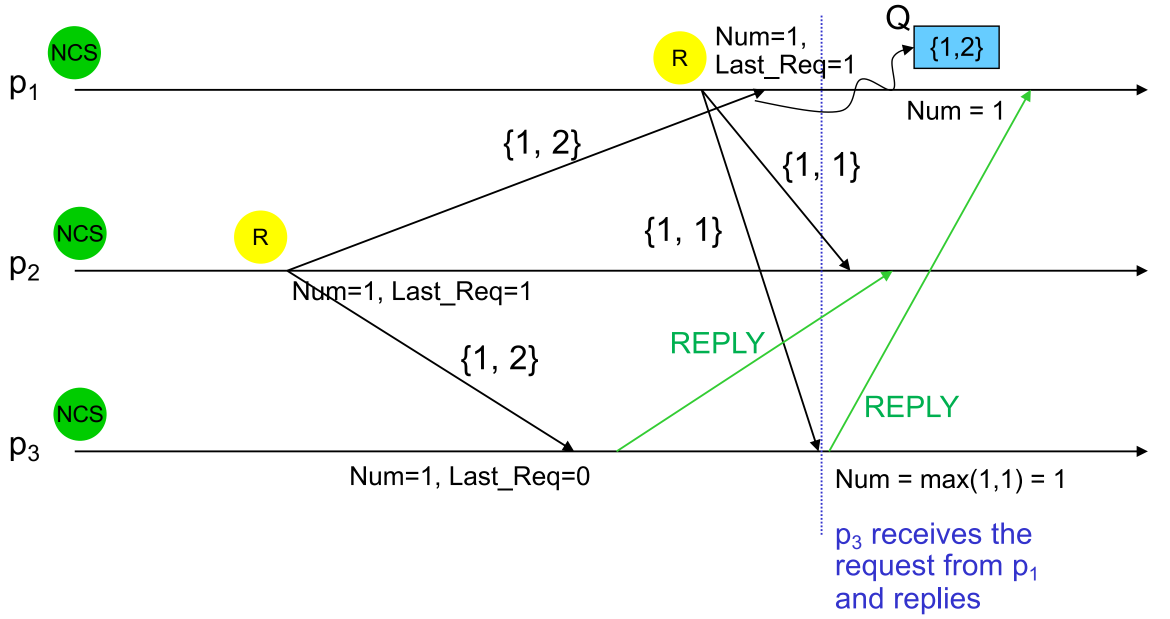
Ricart and Agrawala's algorithm: example



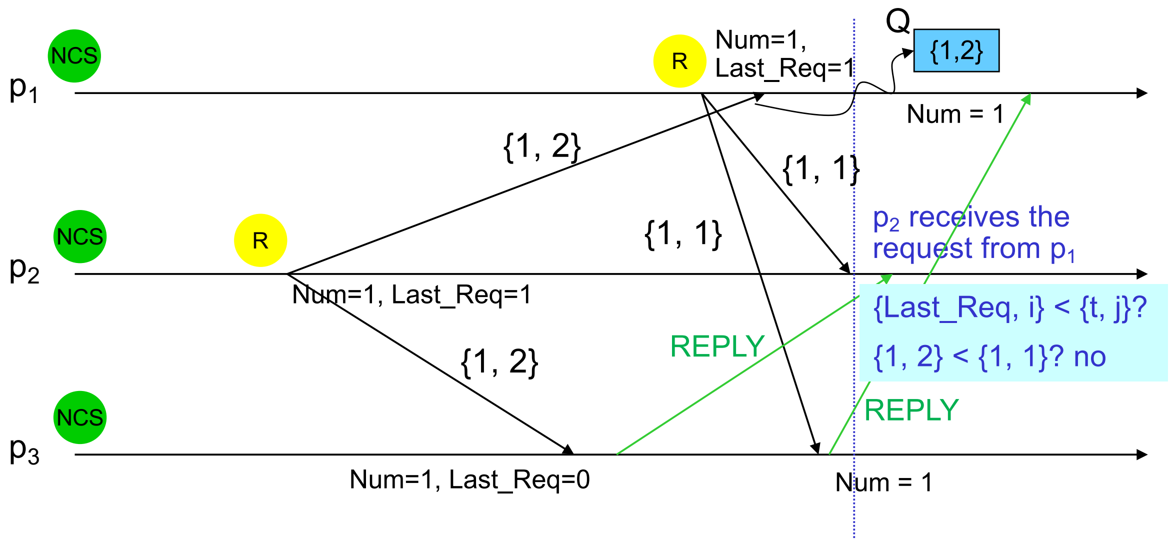
Ricart and Agrawala's algorithm: example



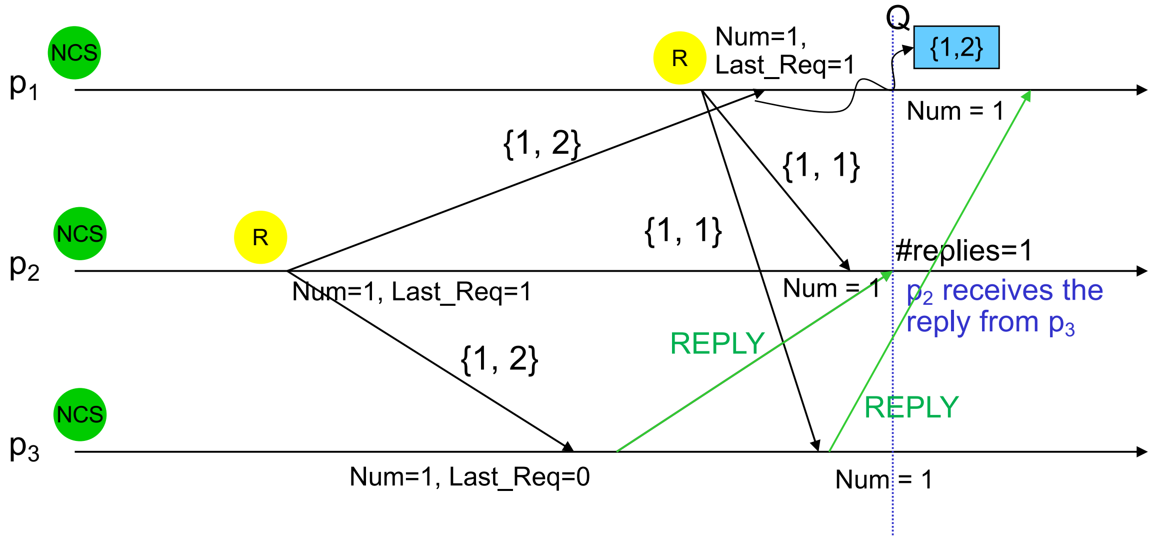
Ricart and Agrawala's algorithm: example



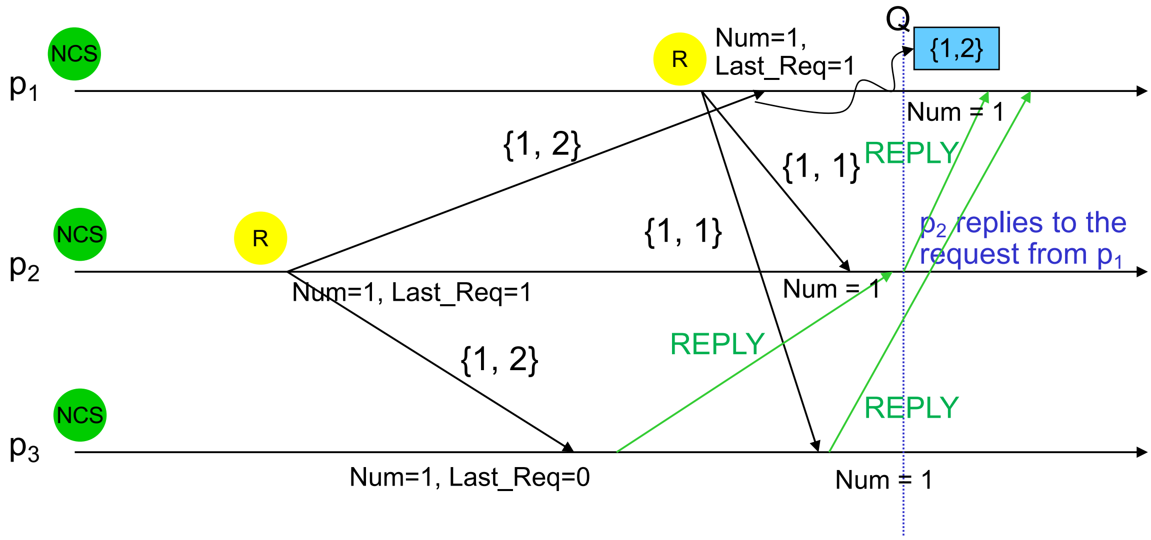
Ricart and Agrawala's algorithm: example



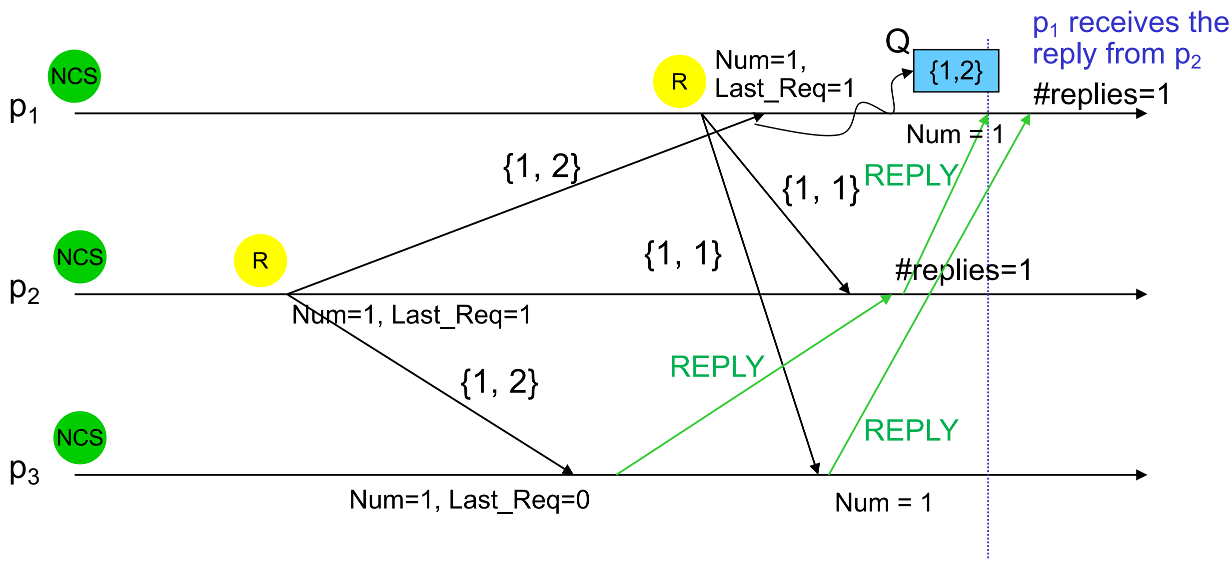
Ricart and Agrawala's algorithm: example



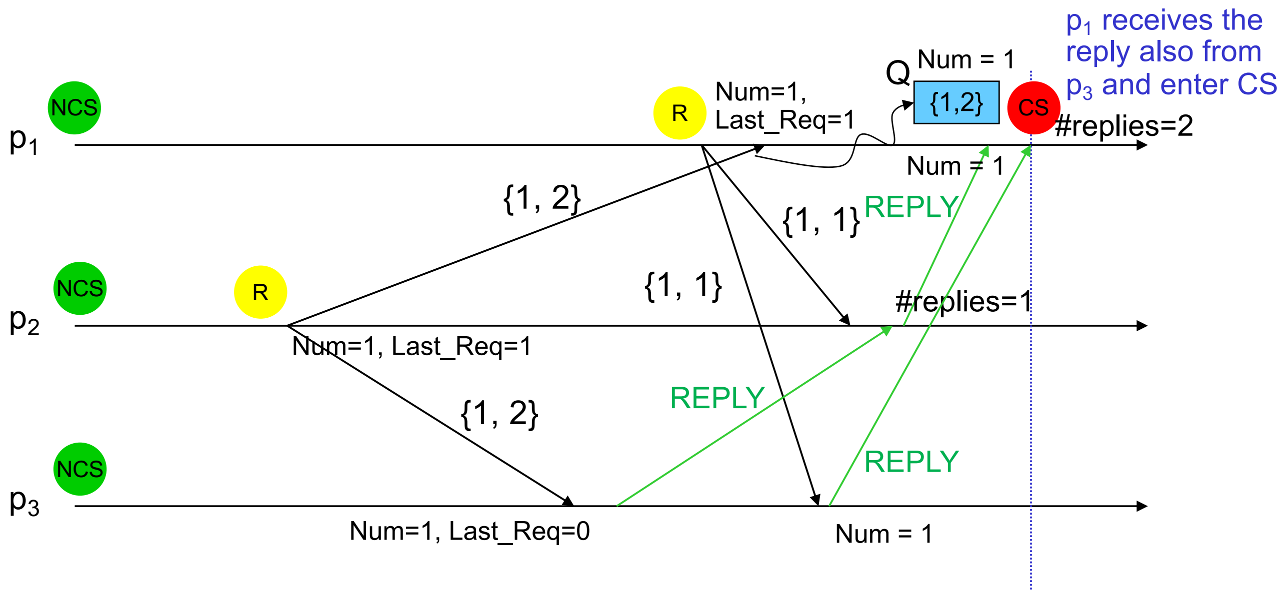
Ricart and Agrawala's algorithm: example



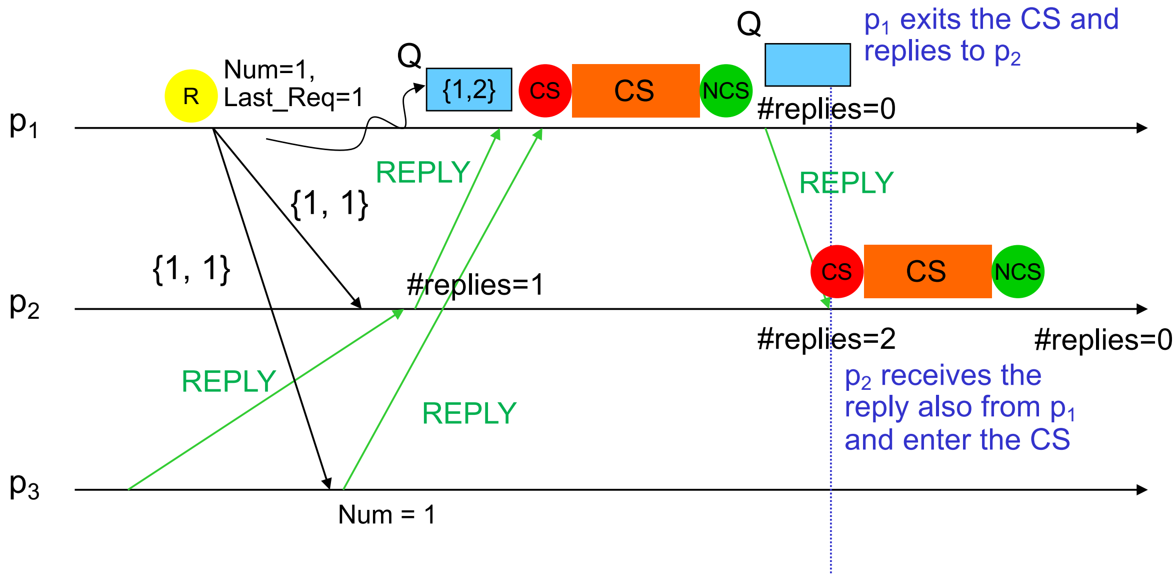
Ricart and Agrawala's algorithm: example



Ricart and Agrawala's algorithm: example



Ricart and Agrawala's algorithm: example



Ricart and Agrawala's algorithm: example

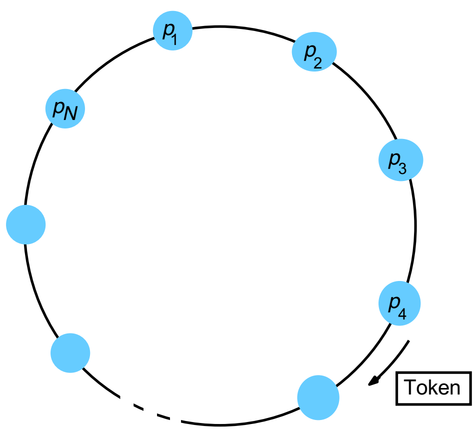
- Pros
  - ✓ Fully distributed, like Lamport's algorithm
    - No central coordinator
  - ✓ Fewer messages than Lamport's algorithm
    - No RELEASE message; ACKs are deferred until exit from CS
    - Only 2(N−1) messages per CS execution: N−1 REQUEST messages, N−1 REPLY messages
  - ✓ Literature reports further optimizations (not covered) reducing messages to N
- Cons (like Lamport's algorithm)
  - ✗ If any process fails, no one can enter the CS → requires a *failure detection* mechanism
  - ✗ Every process can become a bottleneck
    - Every process participates in every decision
  - ✗ Must know the membership of the multicast group

Token-based algorithms

- An auxiliary resource called a **token** is used
  – Other distributed algorithms also use tokens (e.g., leader election)
- The algorithm must define:
  – How token requests are made
  – How the token is maintained and granted
- In a token-based algorithm, at any time there is exactly one token holder
  – This guarantees safety (mutual exclusion)
- Many token-based ME algorithms exist in the literature; we analyze:
  – Decentralized (or *perpetuum mobile*): token management is decentralized and the token moves through the system

Token-based: decentralized algorithm

- Processes are logically organized in a (unidirectional) ring
  – No relation between the ring topology and the physical interconnection of nodes
- The token travels from one process to the next
  – Passes from p_i to $p_{(i+1) \bmod N}$
- The process holding the token can enter the CS
- If a process receives the token but does not want to enter the CS, it passes the token to the next process



Token-based: decentralized algorithm

- ✓ Safety: guaranteed
- ✓ NS: guaranteed if the ring is unidirectional
- ✓ ND: guaranteed if the token is not lost
- Ordering?
- ✗ Network bandwidth is consumed transmitting the token even when no process wants to enter the CS
- ✗ Token loss requires token regeneration
- ✗ Temporary failures may lead to multiple tokens
- Crash of individual processes:
  - Ring must be reconfigured if a process fails
  - If the token holder fails, the token must be regenerated and the next token owner elected

Quorum-based algorithms

- Idea: to enter the CS, a process only needs to collect votes from a subset of processes (*quorum*), not from all processes
- Voting within the subset:
  - The processes vote to determine which process is authorized to enter the CS
  - A process can vote for only one process per turn
- Voting set V_i: subset of $\{p_1, \ldots, p_N\}$, associated with each process p_i

- A process p_i to enter the CS
  - Sends a *request* to all other members of V_i
  - Waits for a *reply* from all members of V_i
  - Upon receiving all the replies from V_i members, it enters CS
  - Upon exiting the CS, it sends a *release* to all members of V_i

- A process p_j in V_i that receives a *request*
  - If the process is in CS or has already replied after receiving the last release, it does not reply and queues the request
  - Otherwise, it replies immediately with a reply

- A process that receives a release:
  - Extracts one request from the queue and sends a reply

Maekawa's algorithm

- Each process p_i executes the following algorithm:

Initialization

state = RELEASED;
voted = FALSE;

CS entry section for p_i

state = WANTED;
multicast *request* to all processes in V_i (including itself);
wait until (number of replies received = K); // $K = |V_i|$
state = HELD;

Upon receiving a request from p_j ($i \neq j$)

```
if (state = HELD or voted = TRUE) then
        queue request from p_j without replying;
else
        send reply to p_j;    // vote in favour of p_j
        voted = TRUE;
end if
```


Maekawa's algorithm

Exit protocol from CS for p_i

```
state = RELEASED;
multicast release to all processes in V_i;
if (queue of requests is non-empty) then
        remove head of queue – from p_k say;
        send reply to p_k;    // vote in favour of p_k
        voted = TRUE;
else
        voted = FALSE;
end if
```

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

```
if (queue of requests is non-empty) then
        remove head of queue – from p_k say;
        send reply to p_k;    // vote in favour of p_k
        voted = TRUE;
else
        voted = FALSE;
end if
```

Maekawa's algorithm: voting set

- How is the voting set V_i defined for p_i?

1. $V_i \cap V_j \neq \emptyset \quad \forall\, i, j$
   - Every pair of voting has non-null intersection: why?
2. $|V_i| = K \quad \forall\, i$
   - All processes have voting sets with the same cardinality K (same *effort* for each process)
3. Each process p_i belongs exactly to K voting sets
   - Equal *responsibility* for every process
4. $p_i \in V_i$
   - To reduce the numer of transmitted messages

- The optimal solution that minimizes K is $K = \lceil \sqrt{N} \rceil$

N=3

V_1={1,2}
V_3={1,3}
V_2={2,3}

N=7

V_1={1,2,3}
V_4={1,4,5}
V_6={1,6,7}
V_2={2,4,6}
V_5={2,5,7}
V_7={3,4,7}
V_3={3,5,6}

Maekawa's algorithm: properties and performance

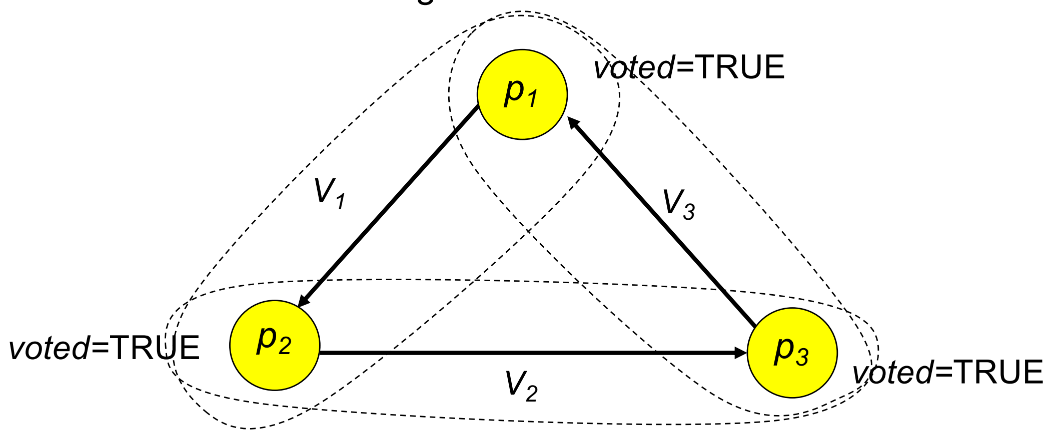
- Safety is guaranteed
  - Voting sets are constructed to ensure they have a non-empty intersection
  - If a quorum grants access to the CS for a process, no other quorum can grant the same permission
- Liveness is not guaranteed
  - Deadlock can occur
  - The algorithm can be made deadlock-free with additional messages
- Performance
  - To enter and exit the CS, $3\sqrt{N}$ messages are required ($2\sqrt{N}$ to enter and $\sqrt{N}$ to exit)
    - More efficient than Ricart and Agrawala for large-scale systems, since $3\sqrt{N} < 2(N-1)$ for $N > 4$

Maekawa algorithm: deadlock example

$P = \{p_1, p_2, p_3\}$
$V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$, $V_3 = \{p_3, p_1\}$

Deadlock situation

- p_1, p_2, and p_3 simultaneously request entry to the CS
- p_1, p_2, and p_3 each set voted=TRUE and wait for a response from the other processes
- There is a circular waiting that causes the deadlock



Comparison of distributed ME algorithms

Algorithm	#msg to enter and exit the CS	#msg to enter the CS	Issues
Permission-based centralized	3	2	Coordinator crash
Ricart Agrawala	$2(N-1)$	$2(N-1)$	Crash of any process
Token-based decentralized	Da 1 a ∞ (se anello bidirezionale)	Da 0 a N-1	Token loss Crash of any process
Maekawa	$3\sqrt{N}$	$2\sqrt{N}$	Possible deadlock

Distributed election algorithms

- Many distributed algorithms require a *coordinator* (or *leader*), e.g.,
  - Sequencer in totally ordered multicast
  - Coordinator in mutual exclusion
- Problem: how to elect the coordinator at runtime?
  - The existing coordinator can crash
  - Election requires reaching distributed consensus
- Two classic election algorithms
  - **Bully algorithm**
  - **Ring election algorithm** (Fredrickson & Lynch)

Distributed election: model

- System with N processes p_i, $i = 1, \ldots, N$
- Processes may crash
- Reliable communication: messages are neither lost, corrupted, nor duplicated
- Each process can hold at most one election at a time
- Each process has a unique ID and the non-faulty process with the highest ID is elected
- Processes can crash, but will eventually recover

Distributed election: properties

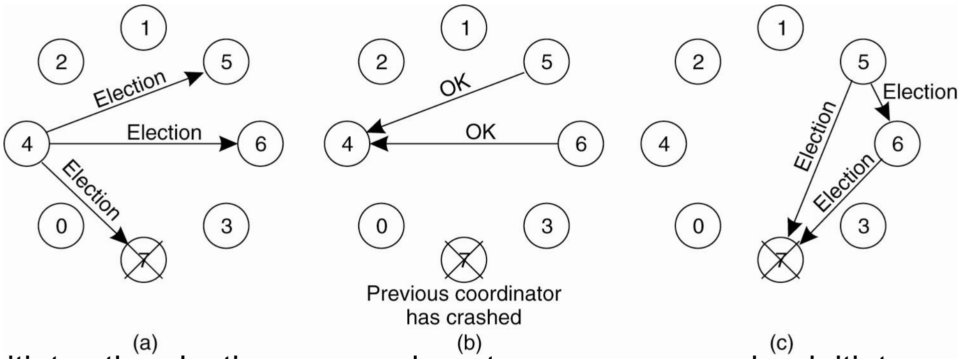
- *Safety*: only the non-faulty process with the highest ID is elected as leader
  - The election result does not depend on which process started the election
  - If multiple processes start an election at the same time, a single winner is eventually announced

- *Liveness*: at any time, some process is eventually elected as leader

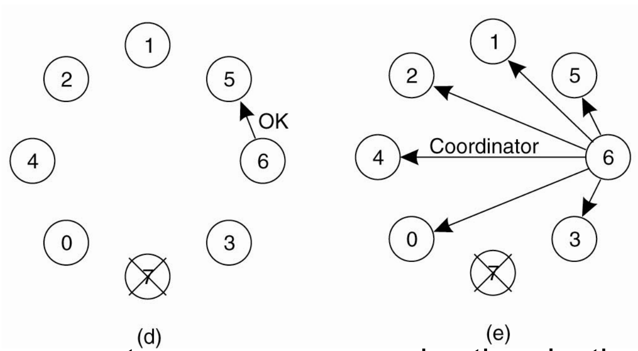
Bully algorithm (Garcia-Molina)

- "Node with highest ID bullies its way into leadership"
- Steps
  - Detection: p_i notices that the leader is not responding and initiates an election
  - Election message: p_i sends an ELECTION message to all processes with higher IDs (p_{i+1}, p_{i+2}, …, p_N)
  - If no one responds, p_i becomes the new leader and announces victory to all processes sending a COORDINATOR message
  - If p_k ($k>i$) receives an ELECTION message from p_i, it replies OK, takes over and starts a new election
  - If p_i receives an OK, it sits back
- Outcome: the non-faulty process with the highest ID is elected as leader
- Note: a new or restarted process that does not know the leader can trigger a new election

Bully algorithm: example



p_4 initiates the election | p_5 and p_6 stop p_4 | p_5 and p_6 initiate a new election



p_6 stops p_5 | p_6 wins the election and becomes the new leader

Which is a strong assumption?

Synchronous DS (or at least partially sync with no unbounded delay), so that we can use some timeout mechanism to identify faulty leader

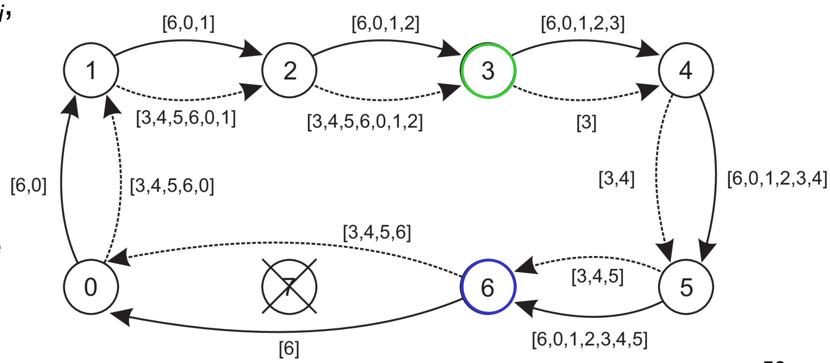
Bully algorithm: communication cost

- Communication cost = how many messages
- Best case: the process with the second highest identifier notices leader's failure
  - It can immediately select itself as leader and then send N-2 COORDINATOR messages ⇒ O(N) messages
- Worst case (assuming no process fails during election): the process with the lowest id initiates the election
  - It sends N-1 ELECTION messages to the other processes, which themselves initiate each one an election

    ((N-1) + (N-2) + … + 1) + N-1 ⇒ O(N^2) messages

Ring algorithm (Fredrickson and Lynch)

- Processes are organized in a logical ring (unidirectional)
  - Each process knows at least its successor
- p_i notices that the leader is failed and initiates election
  - p_i sends ELECTION message to $p_{(i+1) mod N}$ with its own id
  - If $p_{(i+1) mod N}$ *is* faulty, p_i skips over it and goes to the next process along the ring, until a non-faulty process is located
  - At each step, the receiver adds its own id to the list in ELECTION message and forwards the message to the next process
  - Eventually, ELECTION message gets back to p_i, which identifies the highest id in the list and circulates COORDINATOR message to inform everyone else about the new leader

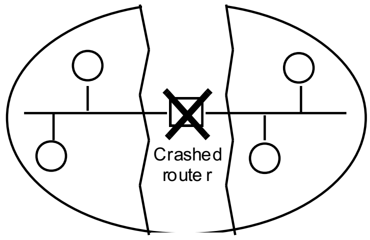


Ring algorithm: communication cost

- Requires 2N messages
  - *N* for ELECTION message, *N* for COORDINATOR message
- But messages are larger than in Bully algorithm

Election algorithms: properties

- Both algorithms assume reliable communication
- Ring election:
  - Works for synchronous and asynchronous systems
  - Works for any *N* and does not require any process to know how many processes are in the ring
- Fault tolerance with respect to process failure
  - What happens if a process crashes during election? It depends on algorithm and crashed process
  - Additional mechanisms may be needed, e.g., ring reconfiguration
- Something to consider:
  - What happens in case of network partition? Multiple new leaders, one per partition



References

- Sections 5.3 and 5.4 of van Steen & Tanenbaum book
- Sections 15.2 and 15.3 of Coulouris et al. book