# CES-27 Processamento Distribuído

## Mutual Exclusion

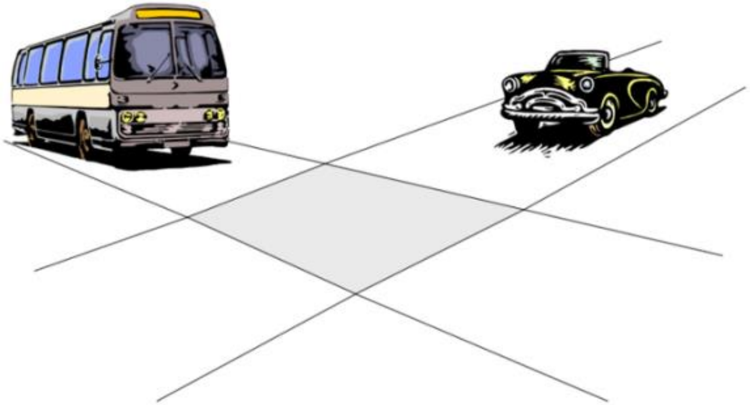
Prof Juliana Bezerra
Prof Celso Hirata
Prof Vitor Curtis

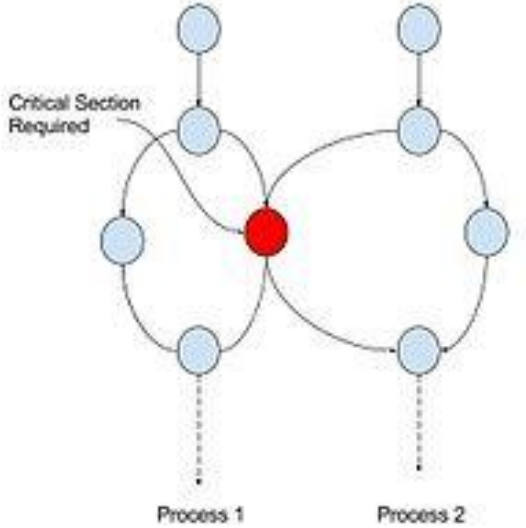
# Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa

# What is mutual exclusion?

- Scenario
  - Multiple processes may want to **access the resource concurrently**
  - At any moment in time at most one process should be privileged ⇨ to have access
- Critical section (CS)
  - A block of source code where the process needs access to the resource, so it needs to be executed atomically
- Mutual exclusion
  - It aims to serialize access to a shared resource





# What is mutual exclusion?

- Properties of mutual exclusion algorithms
  - Safety (Mutual exclusion)
    - In every configuration, at most one process is privileged
  - Liveness (Starvation-freeness)
    - If a process *P* tries to enter its critical section, and no process remains privileged forever, then *P* will eventually enter its critical section.

# Mutual exclusion in parallel system

- Remembering… two main concurrency models are **shared memory** and **message passing**

- Parallel systems is characterized by **shared memory**

- Atomicity is obtained by keeping <u>a lock on the bus</u> from the moment the value is read until the moment the new value is written
  - E.g. lock, semaphore, mutex, monitor
  - Potential problems: deadlocks

# Mutual exclusion in distributed systems

- The core of distributed computation is **message passing**

- Classes of algorithms
  - Token-based algorithms
    - They use auxiliary resources such as **tokens** to resolve the conflicts
      - The process holding the token is privileged
    - Examples:
      - *Centralized algorithm
      - *Token ring algorithm
      - Raymond's algorithm

We will study algorithms marked with *

  - Timestamp-based algorithms
    - They resolve conflict in use of resources based on **timestamps** assigned to requests of resources.
      - Requests for entering a critical section are prioritized by means of logical timestamps
    - Examples:
      - Lamport's algorithm
      - *Ricart-Agrawala

  - Quorum-based algorithms
    - To become privileged, a process needs the permission from a quorum of processes
    - Examples:
      - *Maekawa
      - Agrawal-El Abbadi algorithm

# Mutual exclusion in distributed systems

- System model (our assumptions)
  - Each pair of processes is connected by reliable channels
  - Messages are eventually delivered to recipient, and in FIFO order
  - Processes do not fail
    - Fault-tolerant variants exist in literature

- General directives about performance
  - Efficient algorithms use **fewer messages** and make processes **wait for short** durations to access CS
  - Metrics:
    - Bandwidth
      - The total number of messages sent in each *enter* and *exit* operation
    - Client delay
      - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
    - Synchronization delay
      - The time interval between one process exists CS and next process enters, when only one process is waiting

# Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa

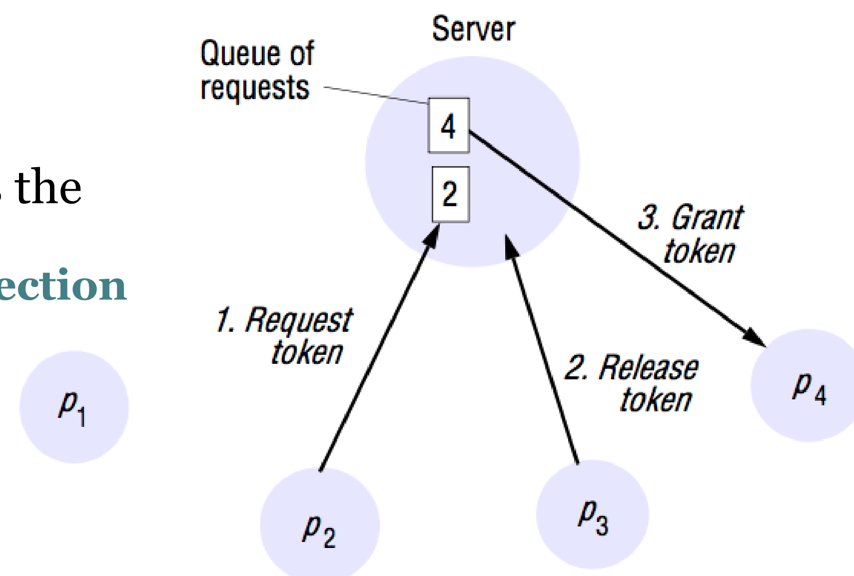
# Centralized algorithm

- Mimic single processor system
- The process, that has the token, has the resource!

- One process works as **coordinator** (**master/leader/server**) that controls the granting of the token
  - Coordinator is chosen using one of our **election algorithms**!

- Other processes can:
  - Request resource
  - Wait for response
  - Receive grant
  - Access resource
  - Release resource

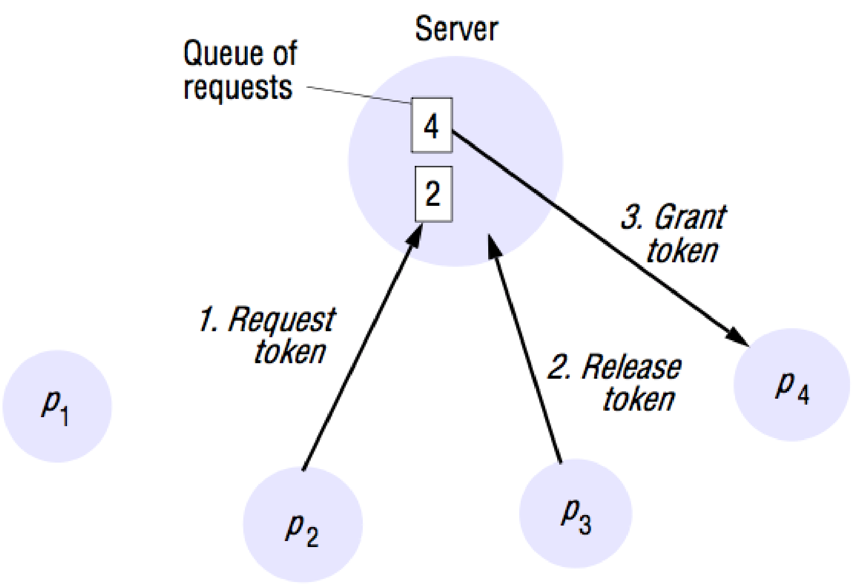
- If a process claims the resource, and the resource is hold by other process, the coordinator:
  - Does not reply until release
  - Maintains the request in its queue (FIFO order)




Note: 4 is in the top of the queue

# Centralized algorithm

- Coordinator actions
  - On receiving a request from process P*i*
    - **if** (master has token)
      - Send token to P*i*
    - **else**
      - Add P*i* to queue
  - On receiving a token from process P*i*
    - **if** (queue is not empty)
      - Dequeue head of queue (say P*j*), send that process the token
    - **else**
      - Retain token
- Benefits
  - Easy to implement, understand, verify
  - Safety
    - At most one process in CS (one token)
  - Liveness
    - All requests processed in order
    - No process waits forever
    - Every request for CS granted eventually




Note: 4 is in the top of the queue

# Centralized algorithm

- Remembering analysis metrics…
  - Bandwidth
    - The total number of messages sent in each *enter* CS and *exit* CS operation
  - Client delay
    - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
  - Synchronization delay
    - The time interval between one process exists CS and next process enters, when only one process is waiting

| Metric | Centralized | Token ring | Ricart-Agrawala | Maekawa |
|---|---|---|---|---|
| Bandwidth | *Enter*: 2 messages<br>*Exit*: 1 message<br>Then **O(1)** | | | |
| Client delay | 2 messages latencies (request + grant)<br>Then **O(1)** | | | |
| Synch. delay | 2 messages latencies (release + grant)<br>Then **O(1)** | | | |

# Centralized algorithm

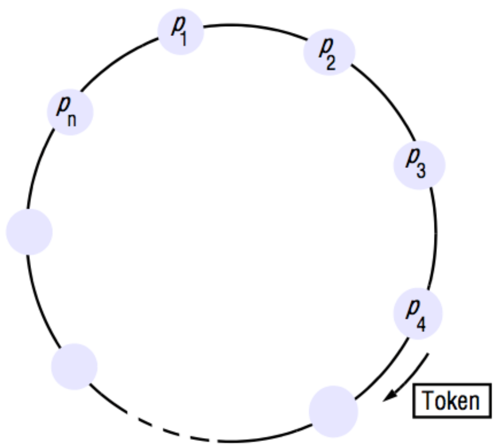
- Problems
  - The coordinator is a single point of failure
    - If it crashes, the entire system may go down
  - If processes normally block after making a request, they cannot distinguish a **dead coordinator** from "**access denied**" since in both cases coordinator does not reply
  - In a large system, a single coordinator has to take care of all process
    - The coordinator can be a **bottleneck**
  - Multiple resources can lead to a deadlock!

# Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa
- Cases
  - Chubby
  - ZooKeeper

# Token ring algorithm

- Consider a number of processes that wish to enter in a critical section




- Assumptions
  - Unidirectional logical ring of processes
  - No duplication or message corruption
  - Possible loss of message
- Safety
  - Only one token
  - Exclusion is guaranteed by allowing one process to enter the critical section if and only if it has the token
- Liveness
  - In order to avoid starvation, the token circulates around the sites

# Token ring algorithm

**ALGORITHM**: (processes **0 to n-1**)

**Process** $P_i$:

…

**receive** token **from** P(n+i-1) mod n

<critical section>

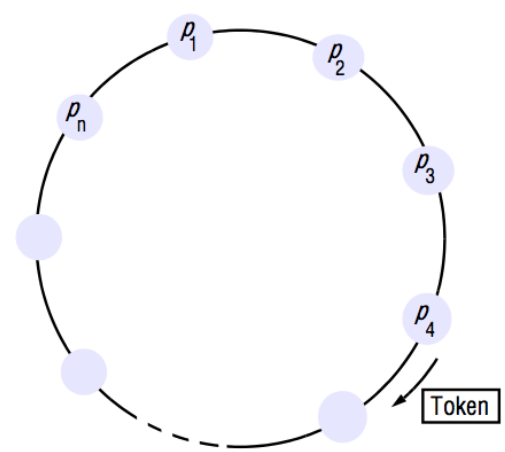
**send** token **to** P(i+1) mod n;

All **n processes** executing the **same algorithm (textual symmetry)**

Receive the **token** from the **previous process** (**clockwise**)

Not necessarily the **process** enters the **critical section**. If doesn't, just **forwards** the **token** (**clockwise**)

Sends the token to the **next** process in the ring (**clockwise**)



# Token ring algorithm

- Remembering analysis metrics…
  - Bandwidth
    - The total number of messages sent in each *enter* CS and *exit* CS operation
  - Client delay
    - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
  - Synchronization delay
    - The time interval between one process exists CS and next process enters, when only one process is waiting

| Metric | Centralized | Token ring | Ricart-Agrawala | Maekawa |
|---|---|---|---|---|
| Bandwidth | *Enter*: 2 messages<br>*Exit*: 1 message<br>Then **O(1)** | *Enter*: N messages through the ring<br>*Exit*: 1 message<br>Then **O(N)** | | |
| Client delay | 2 messages latencies (request + grant)<br>Then **O(1)** | Best case: already have the token ⇨ 0 messages<br>Worst case: just sent token to neighbor ⇨N messages<br>Then **O(N)** | | |
| Synch. delay | 2 messages latencies (release + grant)<br>Then **O(1)** | Best case: process in *enter* is successor of process in *exit* ⇨ 1 message<br>Worst case: process in *enter* is predecessor of process in *exit* ⇨ (N-1) messages<br>Then **O(N)** | | |

# Token ring algorithm

- Benefits
  - The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a CS
  - Since the token circulates among processes in a well-defined order, starvation cannot occur

- Problems
  - Once a process decides it wants to enter a CS, at worst it will have to wait for every other process to enter
    - **Performance** (worst case)
      - Maximum delay = (N-1) (max[Critical Section] + overhead)

  - What happens on **process failure**?
    - **Ring re-connections** or re-establishment of local state variables needed
    - Failure detection is, usually, external to the processes
      - The observer must be in the center of the ring

  - What happens if the **token is lost**?
    - A new one must be generated
    - A possible problem: multiple token generation
    - Token loss detection and regeneration
      - Timeout-based algorithm
      - Misra algorithm (Ping-pong algorithm)

Out of our scope

Attention: The fact that the token has not been spotted for an hour does not mean that it has been lost; some process may still be using it.

# Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa

# Ricart-Agrawala algorithm

- Reference: Glenn Ricart and Ashok K. Agrawala. "**An optimal algorithm for mutual exclusion in computer networks**." *Communications of the ACM* 24.1 (1981): 9-17.
- Classical algorithm from 1981
- No token ⇨use timestamp (Lamport logical time)
- Use the notion of causality and multicast

# Ricart-Agrawala algorithm

*Important:*

- Every *request* sent by $p_i$ has $<T, p_i>$
- A *reply* sent by $p_i$ has $<T_i, p_i>$

$T :=$ time when $p_i$ requested CS
$T_i :=$ current clock of $p_i$
$p_i :=$ process id

- In order to be prepared to next requests of CS, a process should update its clock after receiving any message. You can use Lamport idea: 1+ maximum (my clock, received clock)

```
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 <Ti, pi> at pj (i ≤ j)
    if  (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
        queue request from pi without replying;
    else
        reply immediately to pi;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

T = Time when I requested CS

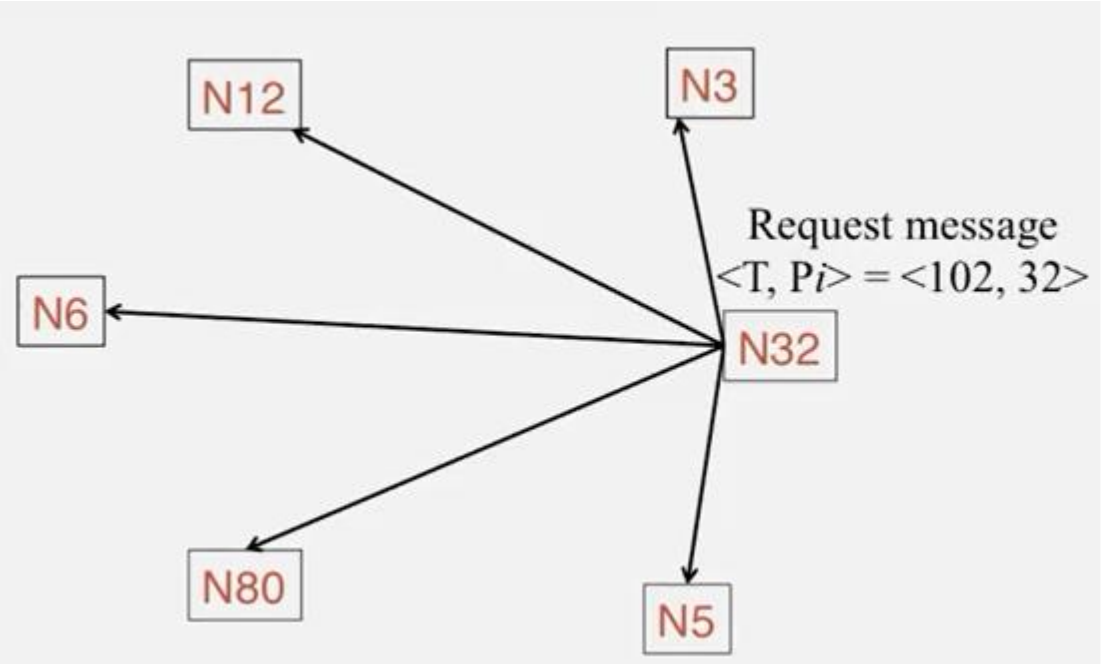
Waiting in a non-blocking mode, so I can process other messages

I am in CS

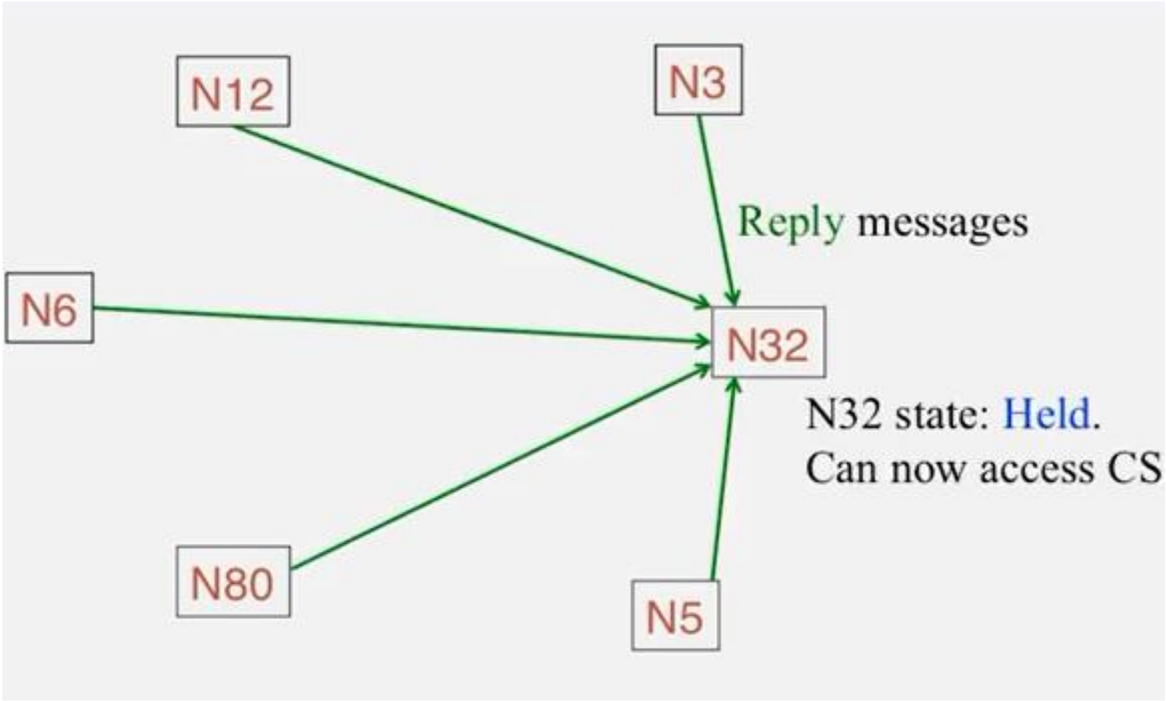
I want CS and the preference is mine!

If T==Ti, the preference is for the lower process id (in this case i, since i<=j)

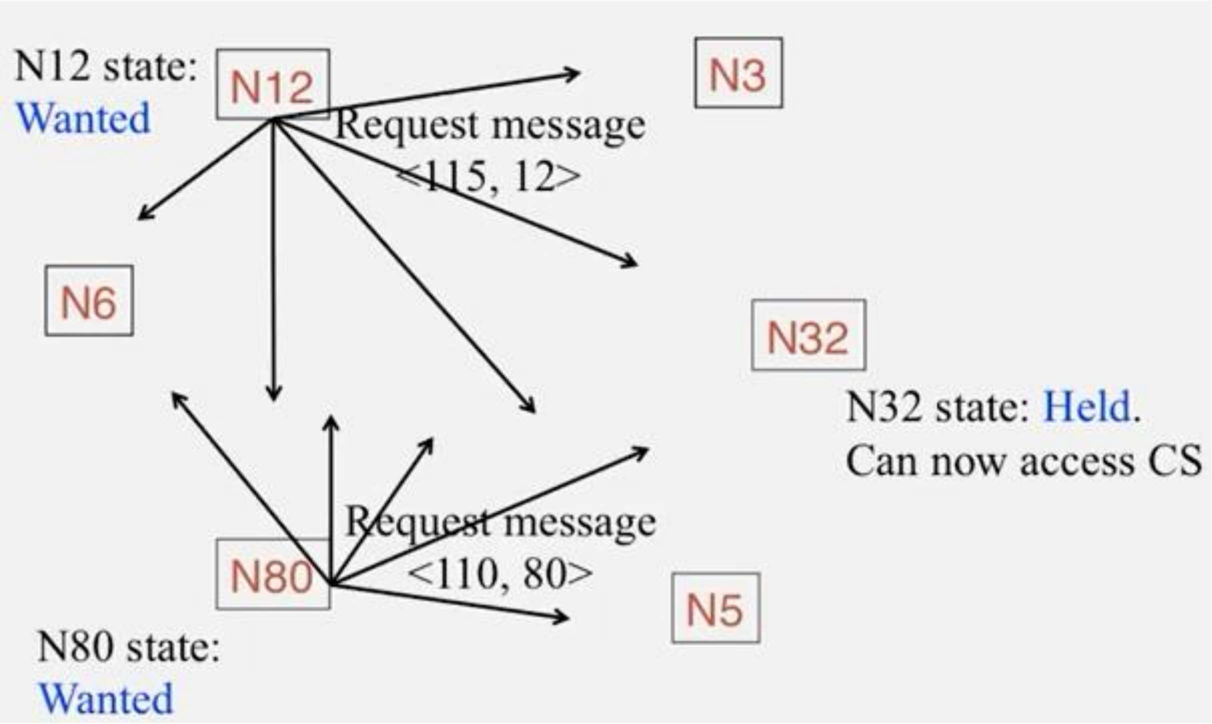
# Ricart-Agrawala algorithm



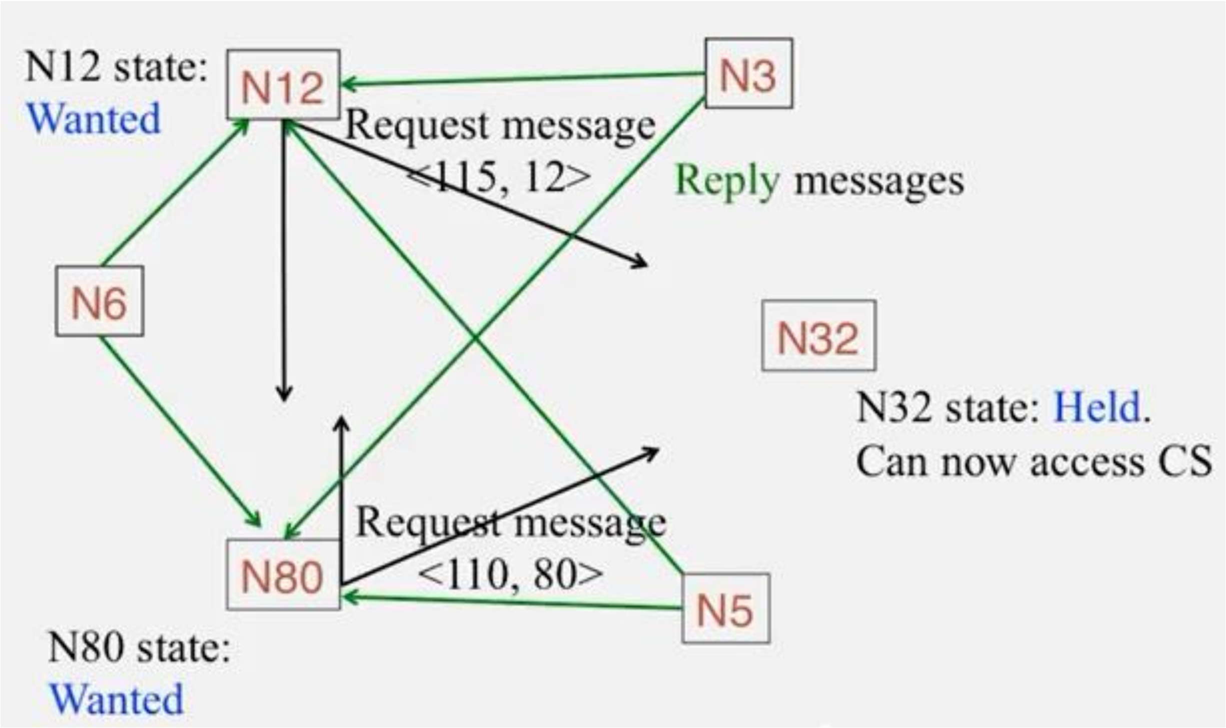
# Ricart-Agrawala algorithm



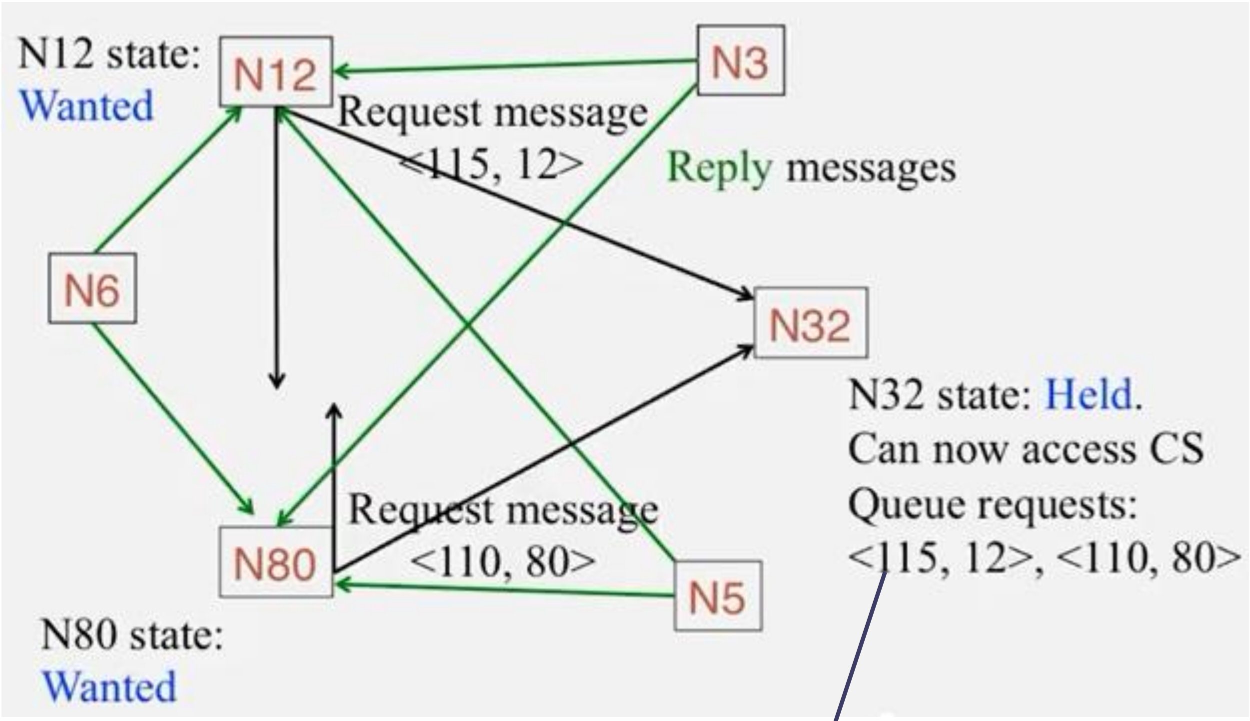
# Ricart-Agrawala algorithm



# Ricart-Agrawala algorithm

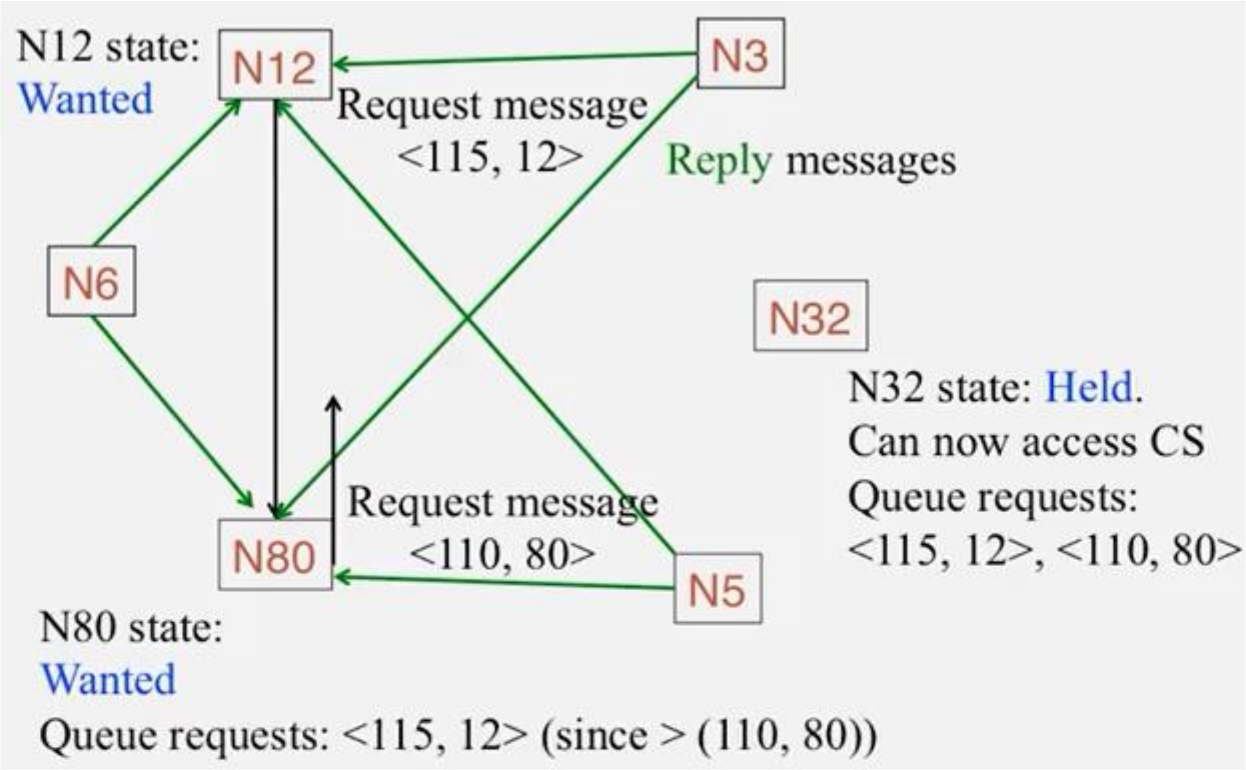


# Ricart-Agrawala algorithm

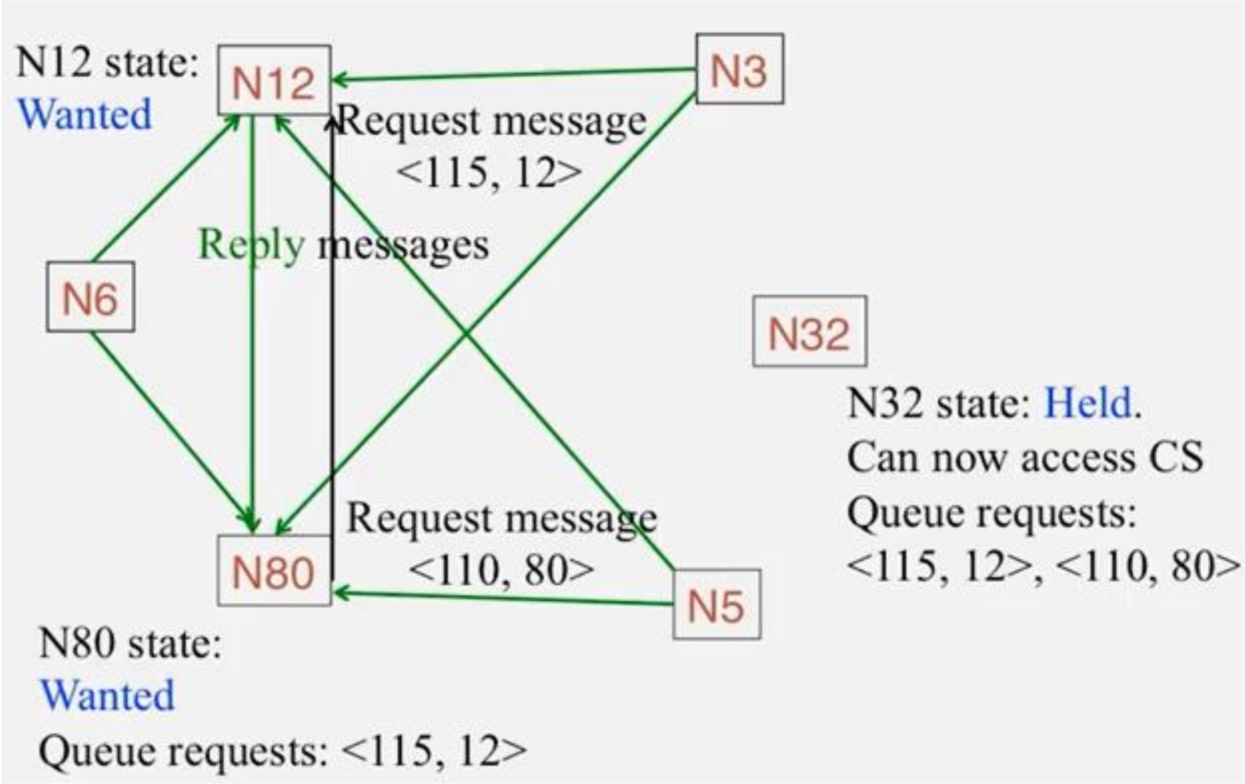



The order in queue here does not matter, since N32 will send *reply* to entire queue when exits CS

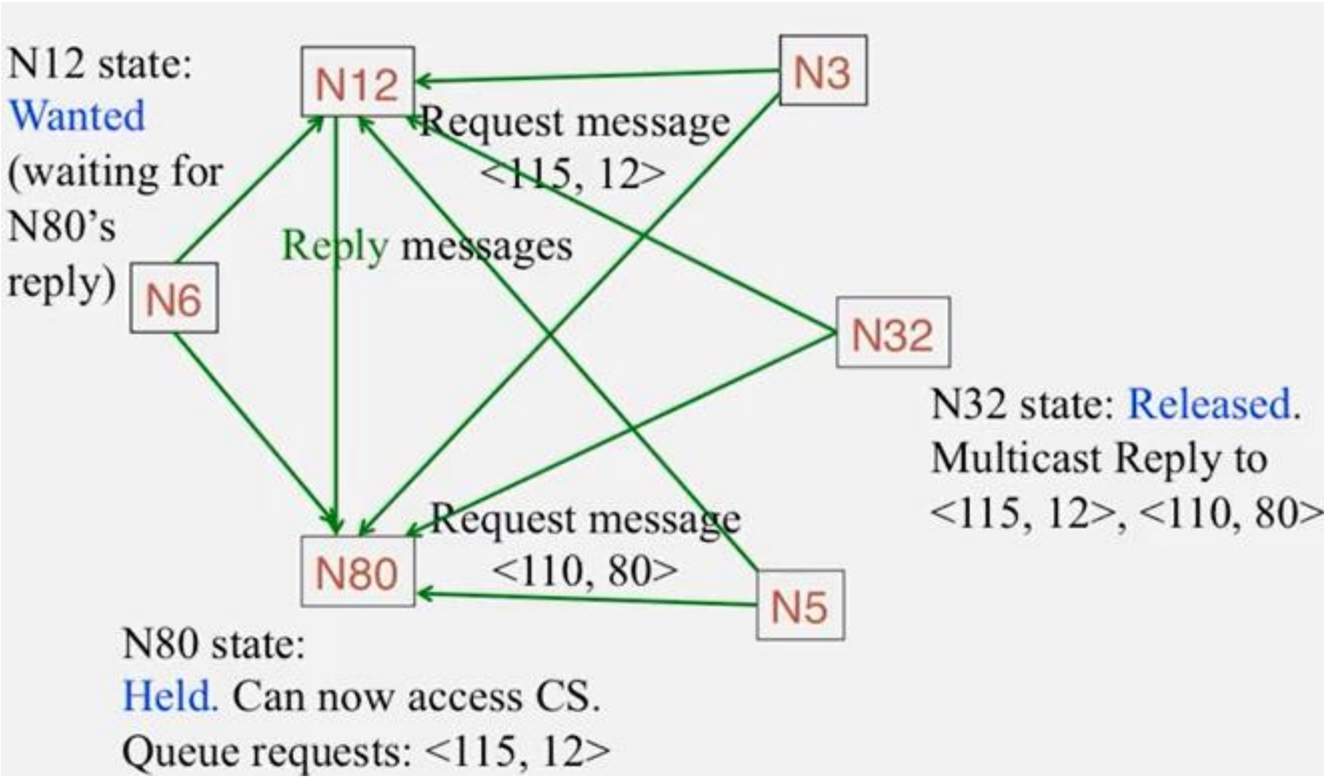
# Ricart-Agrawala algorithm



# Ricart-Agrawala algorithm



# Ricart-Agrawala algorithm



# Ricart-Agrawala algorithm

- Safety
  - Two processes $p_i$ and $p_j$ cannot both have access to CS
    - If they did, then both have sent *reply* to each other
    - Thus $(T_i, p_i) < (T_j, p_j)$ and $(T_j, p_j) < (T_i, p_i)$, which are together impossible
- Liveness
  - Worst case: all other processes request CS, so wait for all (N-1) replies
  - But you will have CS eventually!
- Ordering
  - Requests with lower Lamport timestamps are granted earlier

# Ricart-Agrawala algorithm

- Remembering analysis metrics…
  - Bandwidth
    - The total number of messages sent in each *enter* CS and *exit* CS operation
  - Client delay
    - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
  - Synchronization delay
    - The time interval between one process exists CS and next process enters, when only one process is waiting

| Metric | Centralized | Token ring | Ricart-Agrawala | Maekawa |
|---|---|---|---|---|
| Bandwidth | *Enter*: 2 messages<br>*Exit*: 1 message<br>Then **O(1)** | *Enter*: N messages through the ring<br>*Exit*: 1 message<br>Then **O(N)** | *Enter*: 2(N-1) messages<br>*Exit*: (N-1) messages<br>Then **O(N)** | I can do it better! |
| Client delay | 2 messages latencies (request + grant)<br>Then **O(1)** | Best case: already have the token ⇨ 0 messages<br>Worst case: just sent token to neighbor ⇨N messages<br>Then **O(N)** | Multicast (N-1) requests is O(1) + Receive (N-1) replies in parallel is O(1)<br>Then **O(1)** | |
| Synch. delay | 2 messages latencies (release + grant)<br>Then **O(1)** | Best case: process in *enter* is successor of process in *exit* ⇨ 1 message<br>Worst case: process in *enter* is predecessor of process in *exit* ⇨ (N-1) messages<br>Then **O(N)** | 1 reply from the process in CS<br>Then **O(1)** | |

# Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa

# Maekawa algorithm

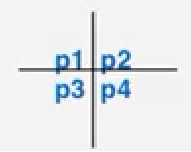
- Approach
  - To get access, **not all processes** have to agree
  - Suffices to split set of processes up into **subsets** (“***voting sets***”) that **overlap**
    - Concept of **quorums**!
  - Suffices that there is consensus within every subset

- Key differences from Ricart-Agrawala
  - Each process requests permission from only its voting set members
    - Not from all as in Ricart-Agrawala
  - Each process (in a voting set) gives permission to at most one process at a time
    - Not to all as in Ricart-Agrawala

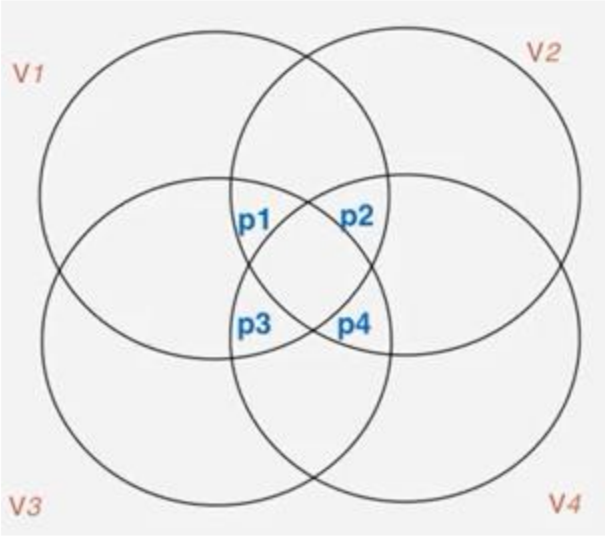
- Model
  - Processes $p_1, .., p_N$
  - Voting sets $V_1, .., V_N$ chosen such that $\forall$ i,j and for some integer M:
    - $p_i \in V_i$ (I am in my voting set)
    - $V_i \cap V_j \neq \emptyset$ (some overlap in every voting set)
    - $|V_i| = K$ (fairness: all voting sets have equal size)
    - Each process $p_k$ is contained in M voting sets

# Maekawa algorithm

- Remembering the model…
  - Processes $p_1, .., p_N$
  - Voting sets $V_1, .., V_N$ chosen such that $\forall$ i,j and for some integer M:
    - $p_i \in V_i$ (I am in my voting set)
    - $V_i \cap V_j \neq \emptyset$ (some overlap in every voting set)
    - $|V_i| = K$ (fairness: all voting sets have equal size)
    - Each process $p_k$ is contained in M voting sets




- Optimization goal
  - Minimize K while achieving mutual exclusion
    - It can be shown to be reached when $K \sim \sqrt{N}$ and M=K
  - Optimal voting sets: nontrivial to calculate
    - Approximation: derive $V_i$ so that $\mathbf{|V_i| \sim 2\sqrt{N}}$
    - Place processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix
    - Let $V_i$ be the union of the row and column containing $p_i$



# Maekawa 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
```

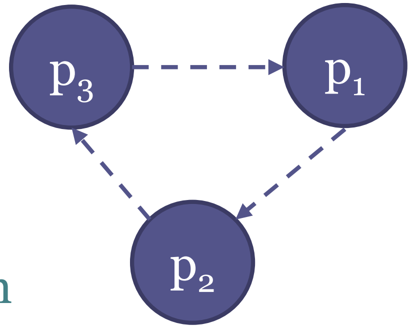
Waiting in a non-blocking mode, so I can process other messages

You can "emulate" messages from $p_i$ to $p_i$

Someone else is in CS, so I can not allow you to enter now

# Maekawa algorithm

- Safety
  - If possible for two processes to enter CS, then processes in the non-empty intersection of their voting sets would have granted access to both
  - Impossible, since all processes make at most one vote after receiving request
- Liveness
  - A process needs to wait for at most (N-1) other processes to finish CS
  - It does not guarantee liveness, since deadlocks are possible. E.g:
    - Three processes $p_1$, $p_2$ and $p_3$
    - $V_1 = V_2 = V_3 = \{p_1, p_2, p_3\}$
    - All processes requested CS
    - Possible to construct cyclic wait graph
      - $p_1$ replies to $p_2$, but queues request from $p_3$
      - $p_2$ replies to $p_3$, but queues request from $p_1$
      - $p_3$ replies to $p_1$, but queues request from $p_2$




  - There are deadlock-free version of the algorithm
    - Use of logical clocks
    - Processes queue requests in happened-before order

# Maekawa algorithm

Note: Consider optimization goal
So $|V_i| \sim \sqrt{N}$

- Remembering analysis metrics…
  - Bandwidth
    - The total number of messages sent in each *enter* CS and *exit* CS operation
  - Client delay
    - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
  - Synchronization delay
    - The time interval between one process exists CS and next process enters, when only one process is waiting

| Metric | Centralized | Token ring | Ricart-Agrawala | Maekawa |
|---|---|---|---|---|
| Bandwidth | *Enter*: 2 messages<br>*Exit*: 1 message<br>Then **O(1)** | *Enter*: N messages through the ring<br>*Exit*: 1 message<br>Then **O(N)** | *Enter*: 2(N-1) messages<br>*Exit*: (N-1) messages<br>Then **O(N)** | *Enter*: $2\sqrt{N}$ messages<br>*Exit*: $\sqrt{N}$ messages<br>Then **O($\sqrt{N}$)** |
| Client delay | 2 messages latencies (request + grant)<br>Then **O(1)** | Best case: already have the token ⇨ 0 messages<br>Worst case: just sent token to neighbor ⇨N messages<br>Then **O(N)** | Multicast (N-1) requests is O(1) + Receive (N-1) replies in parallel is O(1)<br>Then **O(1)** | Multicast $\sqrt{N}$ requests is O(1) + Receive $\sqrt{N}$ replies in parallel is O(1)<br>Then **O(1)** |
| Synch. delay | 2 messages latencies (release + grant)<br>Then **O(1)** | Best case: process in *enter* is successor of process in *exit* ⇨ 1 message<br>Worst case: process in *enter* is predecessor of process in *exit* ⇨ (N-1) messages<br>Then **O(N)** | 1 reply from the process in CS<br>Then **O(1)** | 1 release from the process that exits CS + 1 reply from a process in the voting set (this process is common in the other voting set)<br>Then **O(1)** |

# Mutual Exclusion Algorithms

- Notes on Fault Tolerance
  - None of these algorithms tolerates message loss
  - Centralized algorithm tolerates crash failure of node that has neither requested access nor is currently in the CS
  - Token ring algorithm cannot tolerate single crash failure
  - Ricart-Agrawala algorithm can be modified to tolerate crash failures by the assumption that a failed process sends all replies immediately
    - It requires reliable failure detector
  - Maekawa's algorithm can tolerate some crash failure
    - If process is in a voting set not required, rest of the system not affected