TDDD25
Distributed Systems

Distributed Mutual Exclusion and Election

Christoph Kessler

IDA
Linköping University
Sweden

LINKÖPING UNIVERSITY

Agenda

DISTRIBUTED MUTUAL EXCLUSION AND ELECTION

1. Mutual Exclusion in Distributed Systems
2. Non-Token-Based Algorithms
3. Token-Based Algorithms
4. Distributed Election
5. The Bully and the Ring-Based Algorithms

Mutual Exclusion

- **Mutual exclusion** ensures that concurrent processes make a **serialized access to shared resources or data.**
  - Solves the well-known *critical section problem*!

- In a distributed system, **no shared variables (semaphores)** can be used in order to implement mutual exclusion!
  - Mutual exclusion has to be

    based exclusively on **message passing**,

    in the context of unpredictable message delays and

    no complete knowledge of the state of the system.

Mutual Exclusion

NOT INTERESTING HERE

- Sometimes the resource is managed by a **server** which implements its **own lock** locally together with the mechanisms needed to synchronize access to the resource

  → mutual exclusion and the related synchronization are transparent for the process accessing the resource.

    - For example, database systems with ***transaction processing***

- Often there is no synchronization built in that implicitly protects the resource (files, display windows, peripheral devices, etc.).

  → A mechanism has to be implemented at the level of the processes requesting for access.

- **Basic requirements for a mutual exclusion mechanism:**
  - ***safety***: only one process may execute a critical section (CS) at a time;
  - ***liveness***: a process requesting entry to the CS is eventually granted it (so long as any process executing the CS eventually leaves it).
    - Liveness implies freedom of deadlock and starvation.

Mutual Exclusion

There are two basic approaches to *distributed* mutual exclusion:

1. **Non-token-based**:
   - Each process freely and equally competes for the right to use the shared resource;
   - Requests are arbitrated
     - either by a central control site
     - or by distributed agreement.

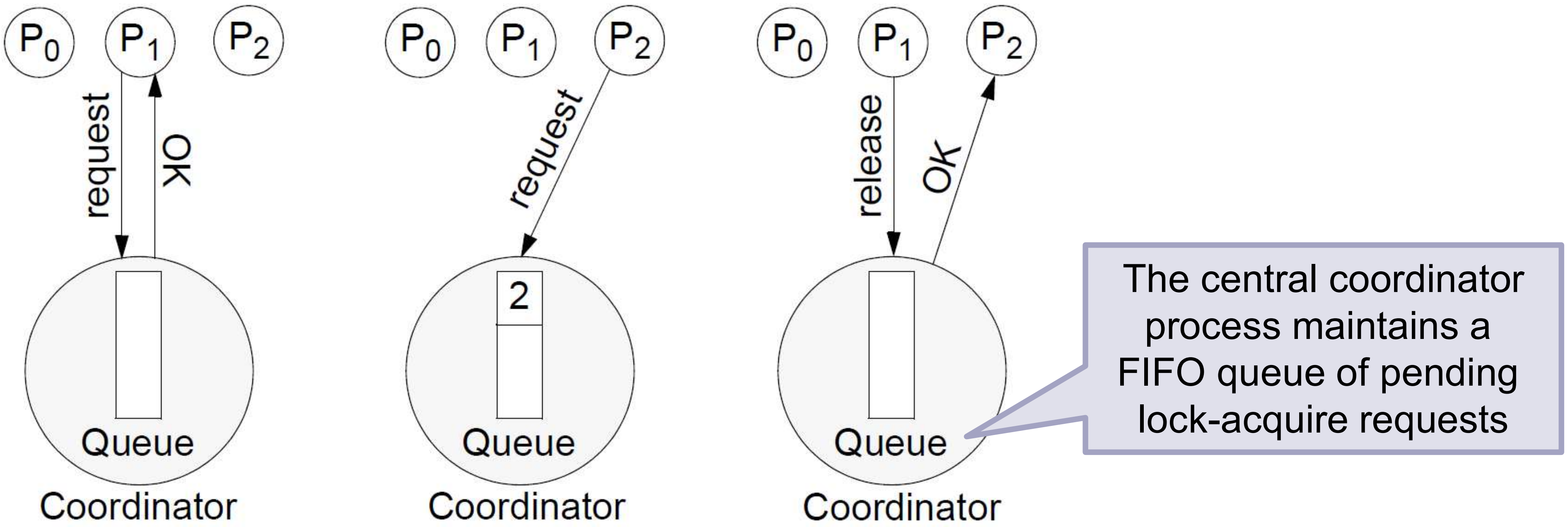
2. **Token-based:**
   - A **logical token** representing the access right to the shared resource is passed in a regulated fashion among the processes;
   - whoever holds the token is allowed to enter the critical section.

Non-Token-Based Mutual Exclusion

- Central Coordinator Algorithm
- Ricart-Agrawala Algorithm

Central Coordinator Algorithm

- A **central coordinator process** grants permission to enter a CS.
  - For example, the process with largest network address



- To enter a CS, a process sends a **request** message to the coordinator and then waits for a reply;
  - during this waiting period, the process can continue with other work.
- The **reply** from the coordinator gives the right to enter the CS.
- After finishing work in the CS, the process notifies the coordinator with a **release** message.

Central Coordinator Algorithm

- The scheme is simple and easy to implement.
- It requires only three messages per use of a CS (*request*, *OK*, *release*).

Problems

- The coordinator can become a performance bottleneck.
- The coordinator is a critical point of failure:
  - If the coordinator crashes, a new coordinator must be created.
  - **The coordinator can be one of the processes competing for access**;

  → an ***election algorithm*** has to be run
  in order to choose one and only one new coordinator.

Ricart-Agrawala Algorithm

- In a distributed environment, it seems more natural to implement mutual exclusion based on **distributed agreement** - not on a central coordinator.

- It is assumed that all processes keep a (Lamport's) logical clock.
  - The algorithm requires a **total ordering of requests**

    → requests are ordered according to their *global logical timestamps*; if timestamps are equal, process identifiers are compared to order them.

- A process that requires entry to a CS **multicasts** the *request* message to all other processes competing for the same resource;
  - it is allowed to enter the CS when *all* processes have replied to this message.
  - The *request* message consists of the requesting process' timestamp (logical clock) and its identifier.

- Each process keeps its **state** with respect to the CS:
  - *RELEASED*, *REQUESTED*, or *HELD*.

Ricart-Agrawala Algorithm

Rule for process initialization:

/* performed by each process P_i at initialization */

[RI1]: $state_{P_i}$:= *RELEASED*

Rule for access request to CS:

/* performed whenever process P_i requests an access to the CS */

[RA1]: $state_{P_i}$:= *REQUESTED*

T_{P_i} := the value of the local logical clock corresponding to this request.

[RA2]: P_i sends a request message to all processes;

the message is of the form (T_{P_i}, *i*), where *i* is an identifier of P_i

[RA3]: P_i waits until it has received replies from *all* other *n*-1 processes.

Rule for executing the CS:

/* performed by P_i after it received the *n*-1 replies */

[RE1]: $state_{P_i}$:= *HELD*

P_i enters the CS.

Ricart-Agrawala Algorithm (cont.)

Rule for handling incoming requests:

/* performed by P_i whenever it received a request (T_{Pj}, j) from P_j */

[RH1]: **if** $state_{Pi} = HELD$
or (($state_{Pi} = REQUESTED$) **and** ((T_{Pi}, i) < (T_{Pj}, j))) **then**
Queue the request from P_j without replying
else
Reply immediately to P_j.
end if

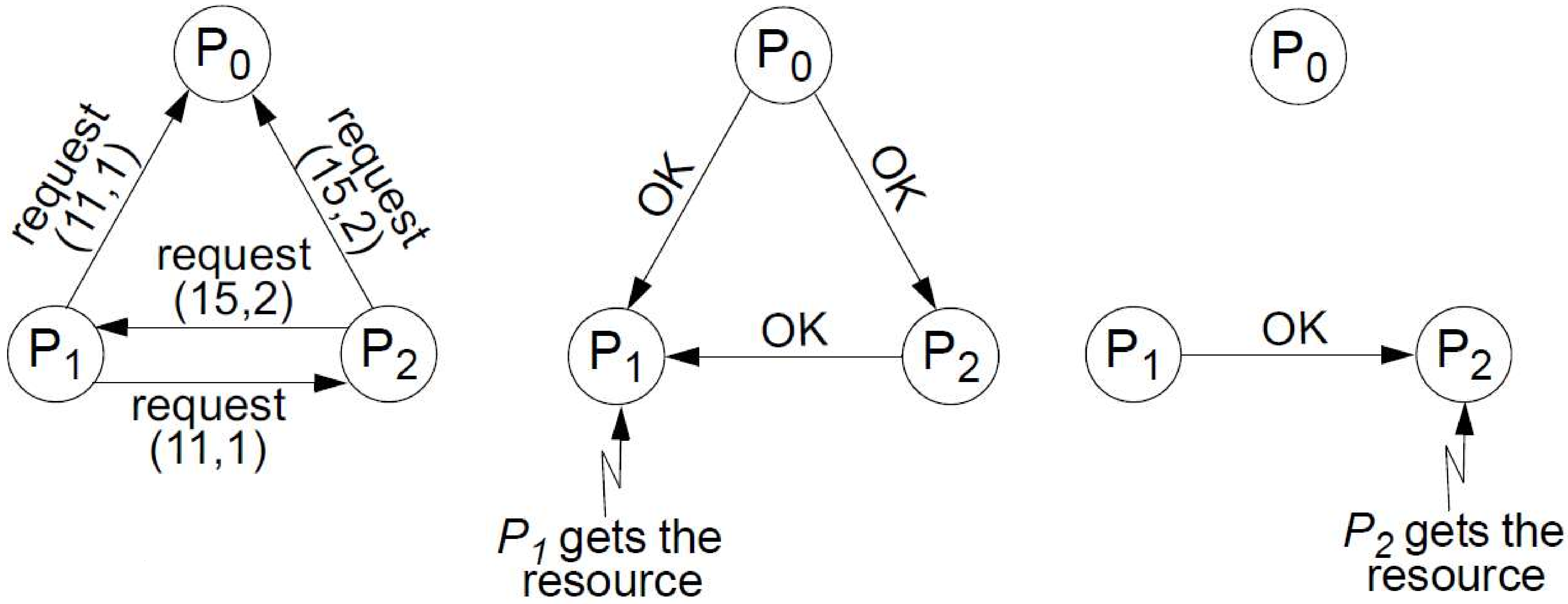
Rule for releasing a CS:

/* performed by P_i after it finished work in a CS */

[RR1]: $state_{Pi} := RELEASED$.
P_i replies to all queued requests.

- A request issued by a process P_j is **blocked** by another process P_i only if P_i is holding the resource or if it is requesting the resource with a **higher priority** (this means a **smaller timestamp**) than P_j.

Ricart-Agrawala Algorithm



Problems

- The algorithm is **expensive** in terms of message traffic;
  - it requires 2(n-1) messages for entering a CS: (n-1) requests and (n-1) replies.
- The failure of any process involved makes progress impossible if no special recovery measures are taken.

Token-Based Mutual Exclusion

- Ricart-Agrawala Second Algorithm
- Token Ring Algorithm

Ricart-Agrawala Second Algorithm

- **A process is allowed to enter the critical section when it got the token.**
- Initially, the token is assigned arbitrarily to one of the processes.
- In order to get the token, a process sends a **request** to all other processes competing for the same resource.
  - The request message consists of the requesting process' timestamp (logical clock) and its identifier.
- When a process P_i leaves a critical section, it passes the token to one of the processes that are waiting for it.
  - If no process is waiting,

    P_i retains the token (and is allowed to enter the CS if it needs);

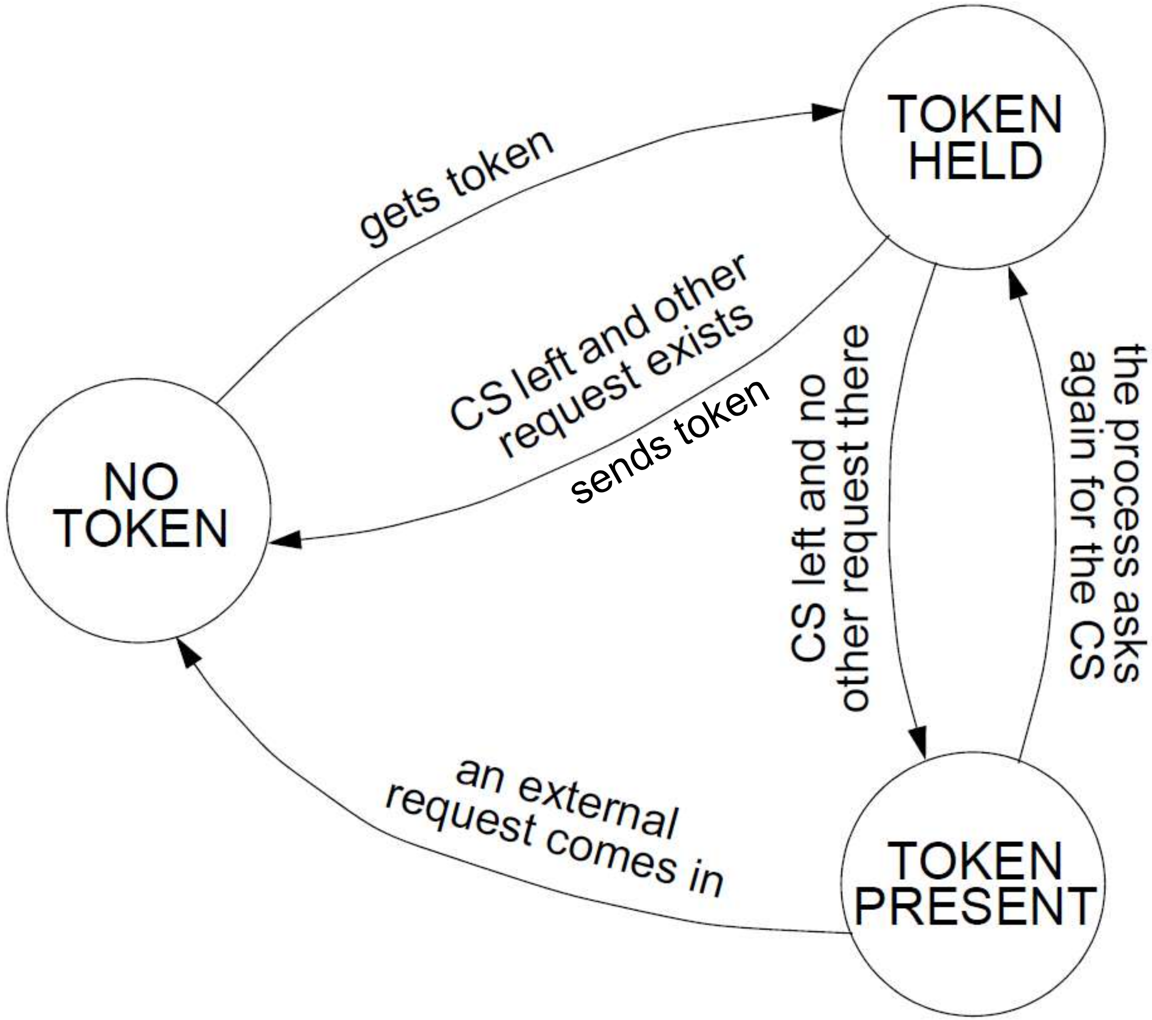
    it will pass over the token as result of an incoming request.

How does P_i find out if there is a pending request?

- Each process P_i records the **timestamp** corresponding to the last request it got from process P_j, in $\textbf{\textit{request}}_{\textbf{\textit{Pi}}}[\,j\,]$.
- In the *token* itself, ***token***[*j*] records the **timestamp** (logical clock) of P_j's last holding of the token.
  - If $request_{Pi}[\,j\,] > token[\,j\,]$ then P_j has a pending request.

Ricart-Agrawala Second Algorithm

- Each process keeps its **state** with respect to the token:
  - *NO-TOKEN*, *TOKEN-PRESENT*, *TOKEN-HELD*.



Ricart-Agrawala Second Algorithm

Rule for process initialization:

/* performed at initialization */

[RI1]: $state_{P_i}$:= *NO-TOKEN* for all processes P_i, except one process P_x for which $state_{P_x}$:= *TOKEN-PRESENT*

[RI2]: *token*[*k*] initialized to 0 for all elements *k* = 1 .. *n*.

request$_{P_i}$[*k*] initialized to 0 for all processes P_i and all elements *k* = 1 .. n.

Rule for access request and execution of the CS:

/* performed whenever process P_i requests an access to the CS.
Note that P_i can already possess the token (state *TOKEN-PRESENT*) */

[RA1]: **if** state$_{P_i}$ = *NO-TOKEN* **then**

Pi **sends** a request message to all processes;

the message is of the form (T_{P_i}, *i*),
where $T_{P_i} = C_{P_i}$ is the value of its local logical clock.

Pi **waits** until it receives the token.

end if

$state_{P_i}$:= *TOKEN-HELD*

P_i enters the CS.

Ricart-Agrawala Second Algorithm

Rule for handling incoming requests:

/* performed by P_i whenever it received a request (T_{Pj}, j) from P_j */

[RH1]: $request_{Pi}[j]$:= **max** ($request_{Pi}[j]$, T_{Pj})

[RH2]: **if** $state_{Pi}$ = *TOKEN-PRESENT* **then**

P_i releases the resource (see rule RR2).

end if

Rule for releasing a CS:

/* performed by P_i after it finished work in a CS, or when it holds the token without using it and got a request */

[RR1]: $state_{Pi}$ = *TOKEN-PRESENT*

[RR2]: **for** k = [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] **do**

if $request_{Pi}[k] > token[k]$ **then** // pass the token to P_k :

$state_{Pi}$:= *NO-TOKEN*

$token[i] := C_{Pi}$, the value of the local logical clock

P_i **sends** the *token* to P_k

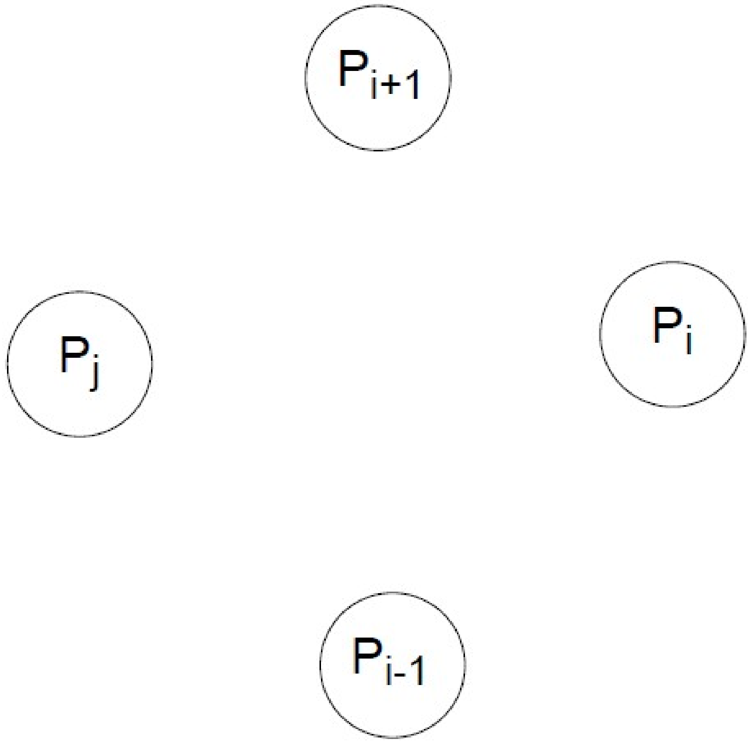
break /* leave the for loop */

end if

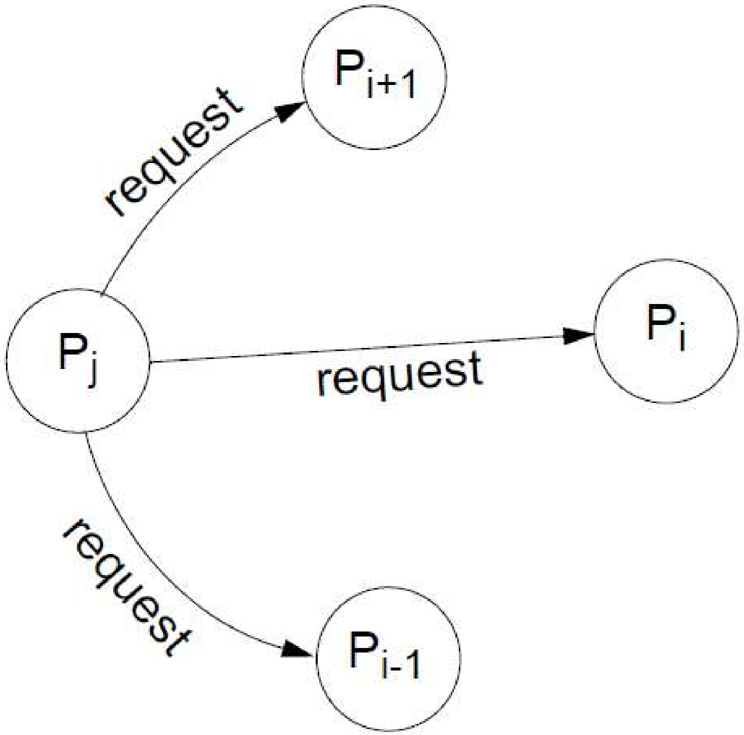
end for

A process P_k with a pending request is searched for in the **round-robin** order [i+1, i+2,..., n, 1, 2,..., i-2, i-1]. **This in order to avoid starvation!**

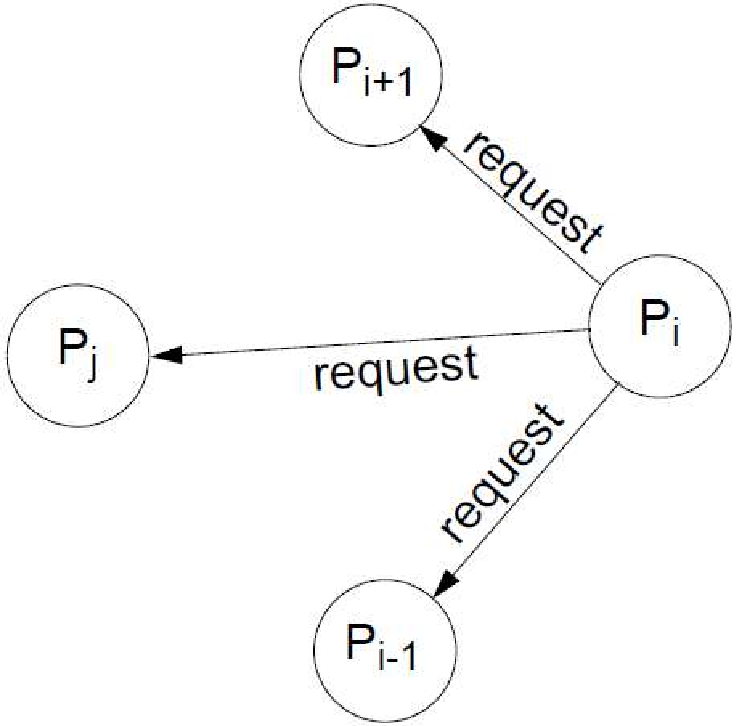
Ricart-Agrawala Second Algorithm



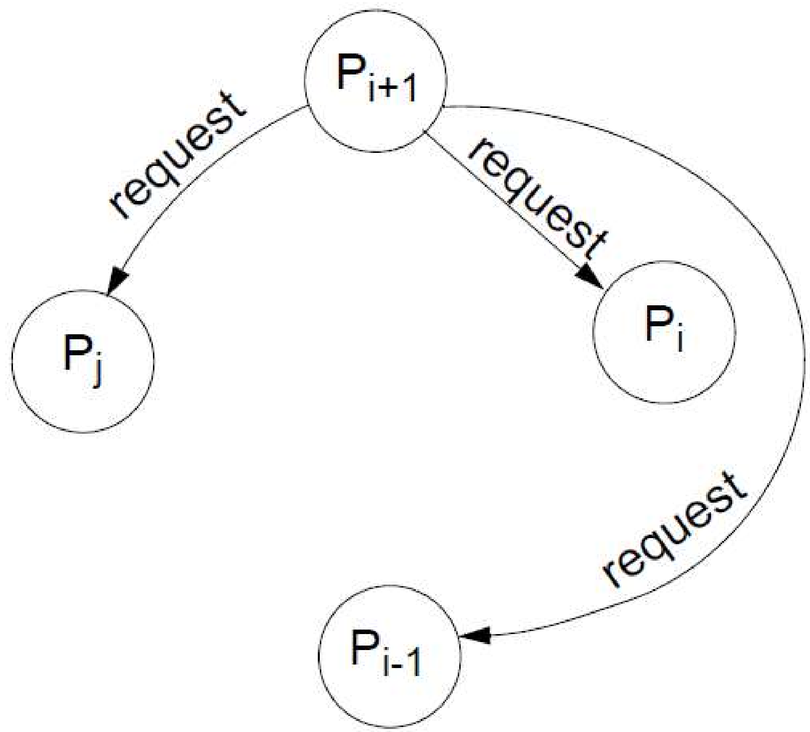
Ricart-Agrawala Second Algorithm



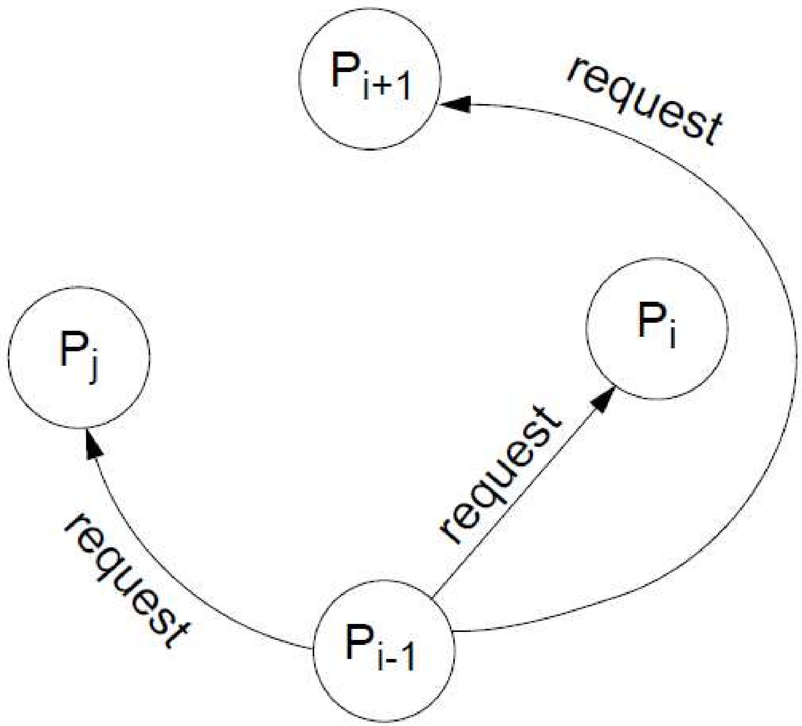
Ricart-Agrawala Second Algorithm



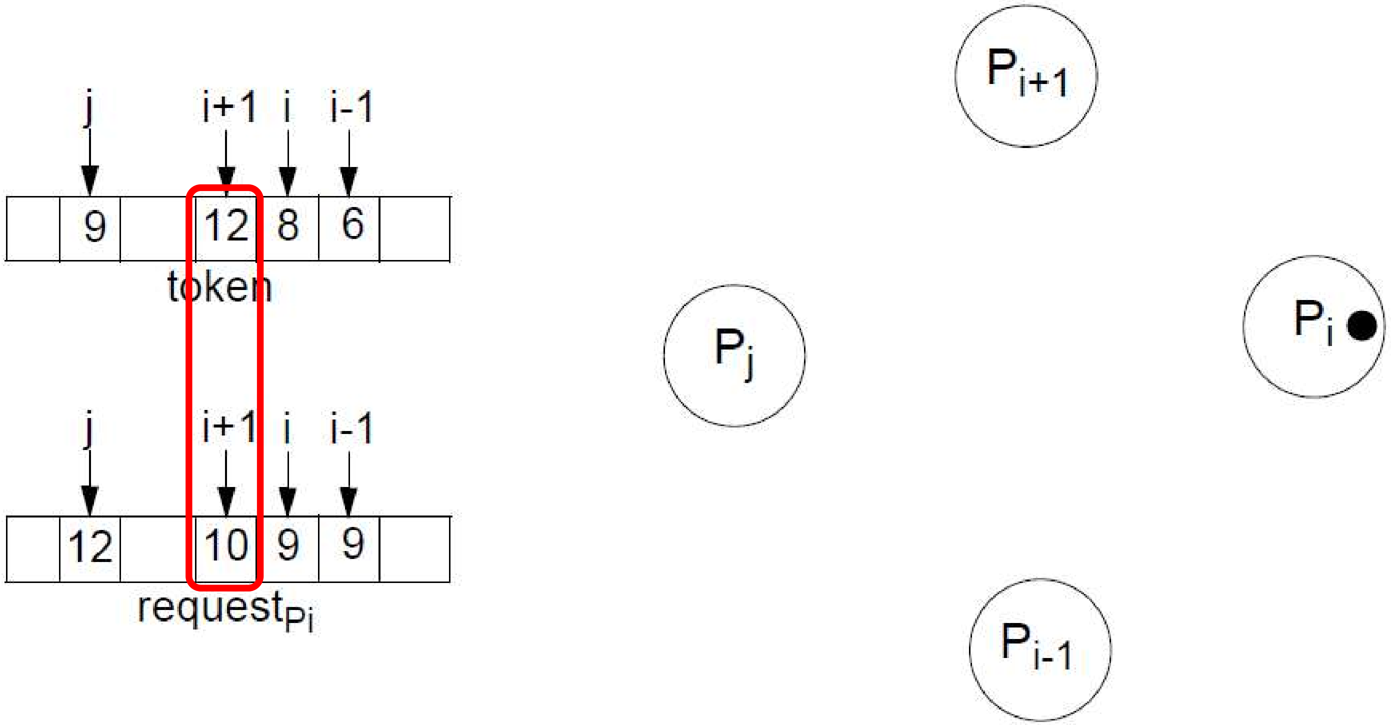
Ricart-Agrawala Second Algorithm



Ricart-Agrawala Second Algorithm



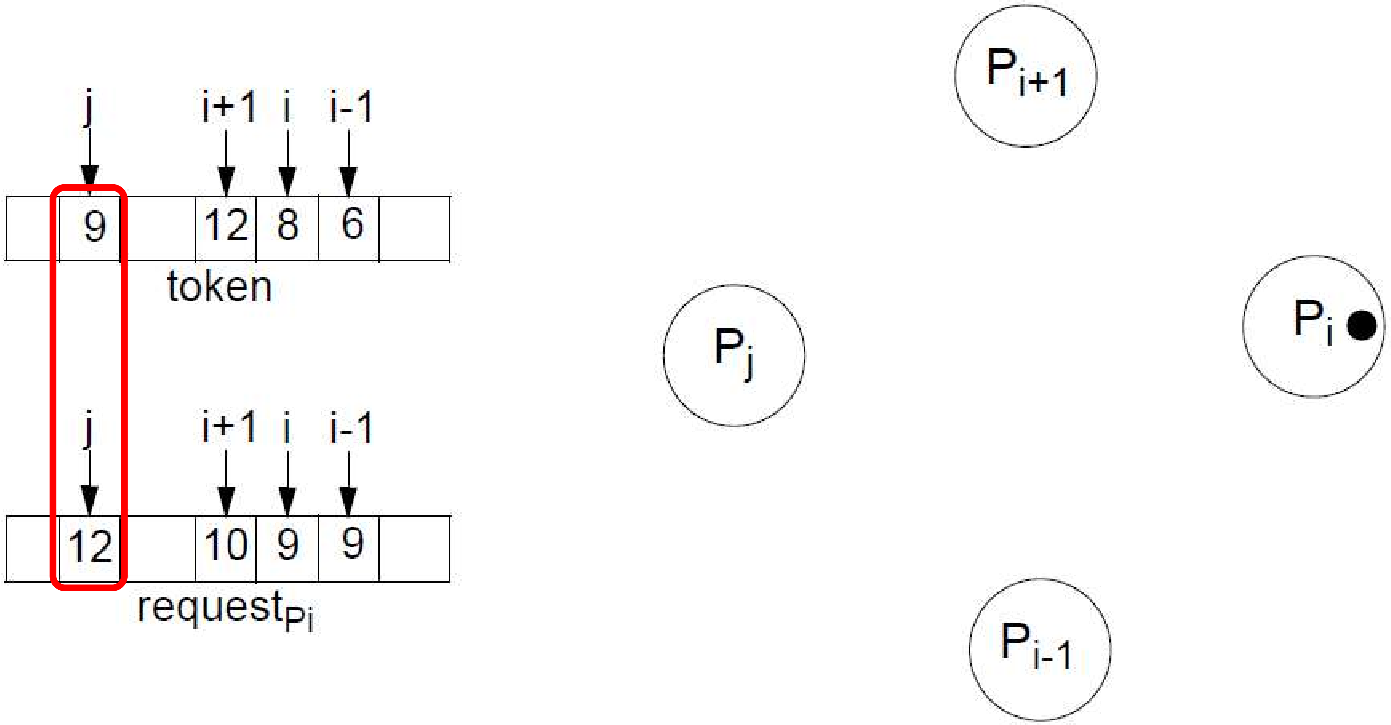
Ricart-Agrawala Second Algorithm



Eventually, P_i got the token, executed its CS and now it is ready to give the token away. (Its local logical clock shows 11)

To whom should it pass the token?

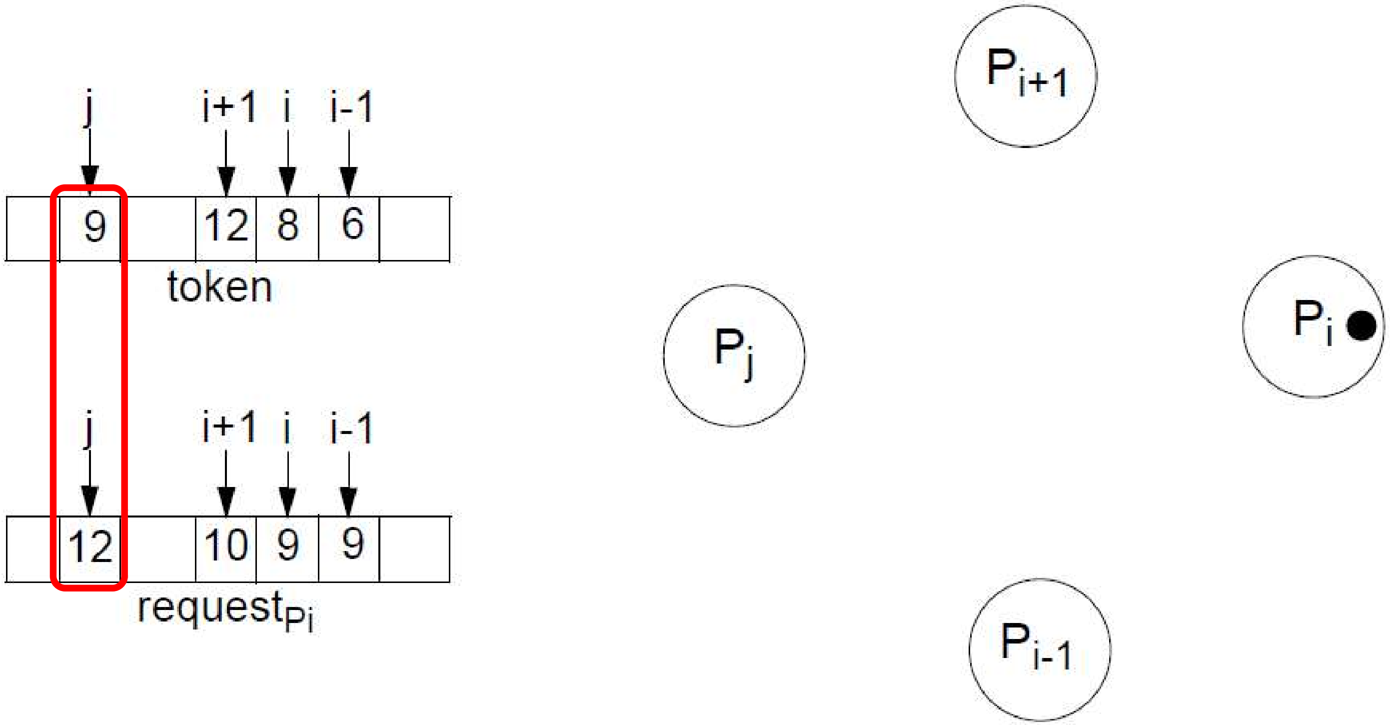
Ricart-Agrawala Second Algorithm



Eventually, P_i got the token, executed its CS and now it is ready to give the token away. (Its local logical clock shows 11)

To whom should it pass the token?

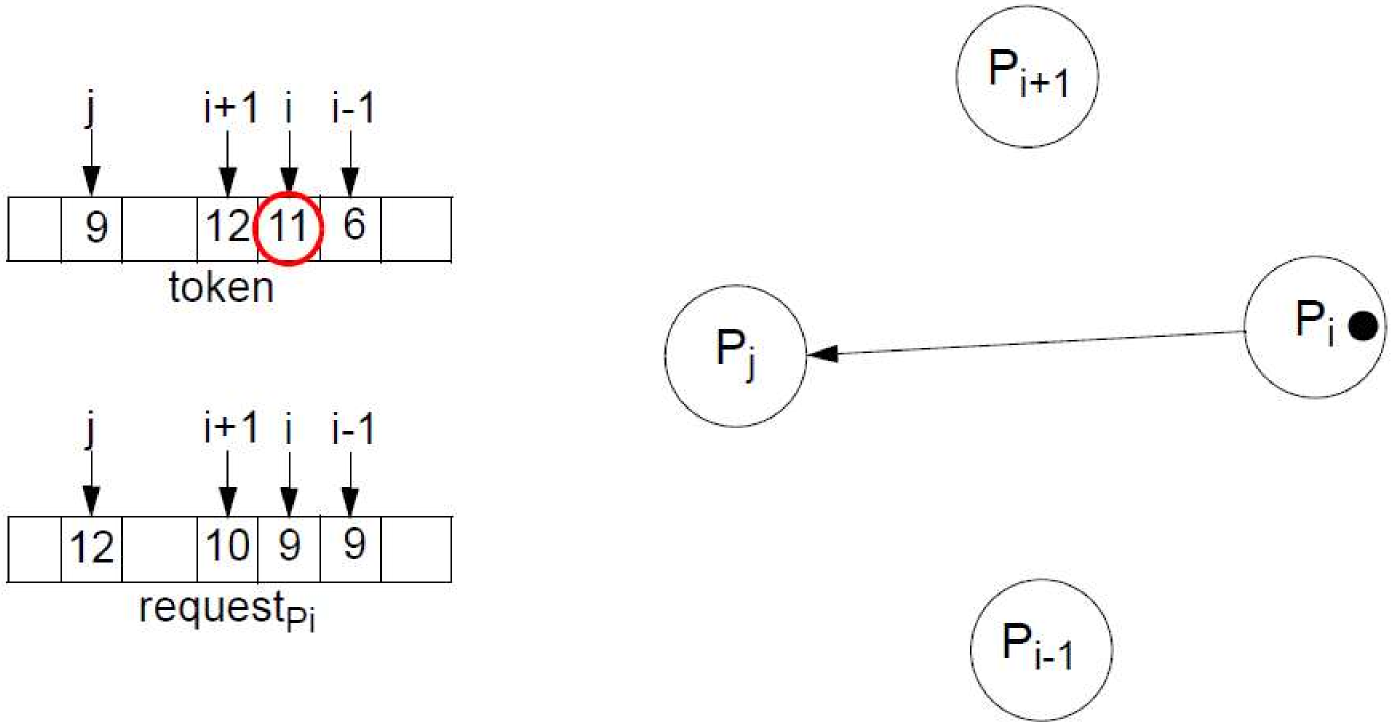
Ricart-Agrawala Second Algorithm



Eventually, P_i got the token, executed its CS and now it is ready to give the token away. (Its local logical clock shows 11)

To whom should it pass the token?

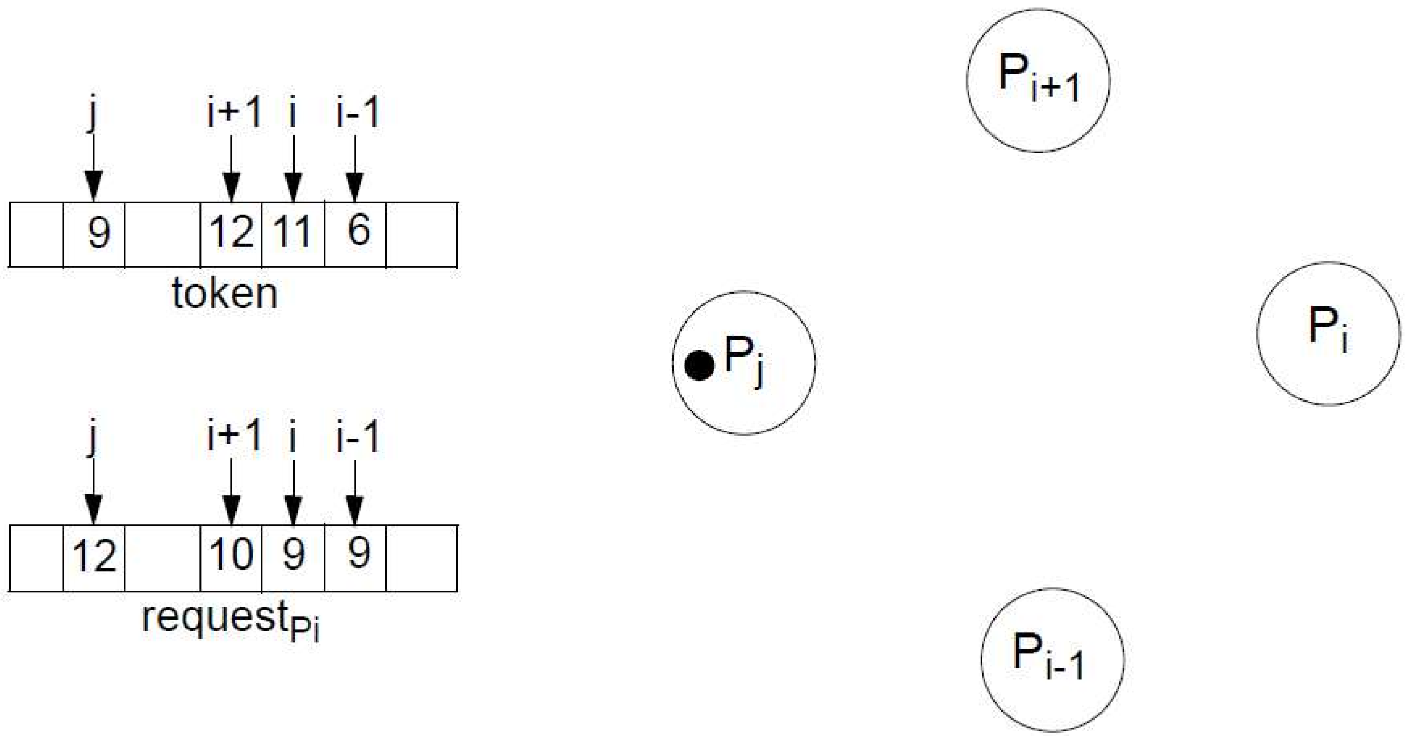
Ricart-Agrawala Second Algorithm



Eventually, P_i got the token, executed its CS and now it is ready to give the token away. (Its local logical clock shows 11)

To whom should it pass the token?

Ricart-Agrawala Second Algorithm

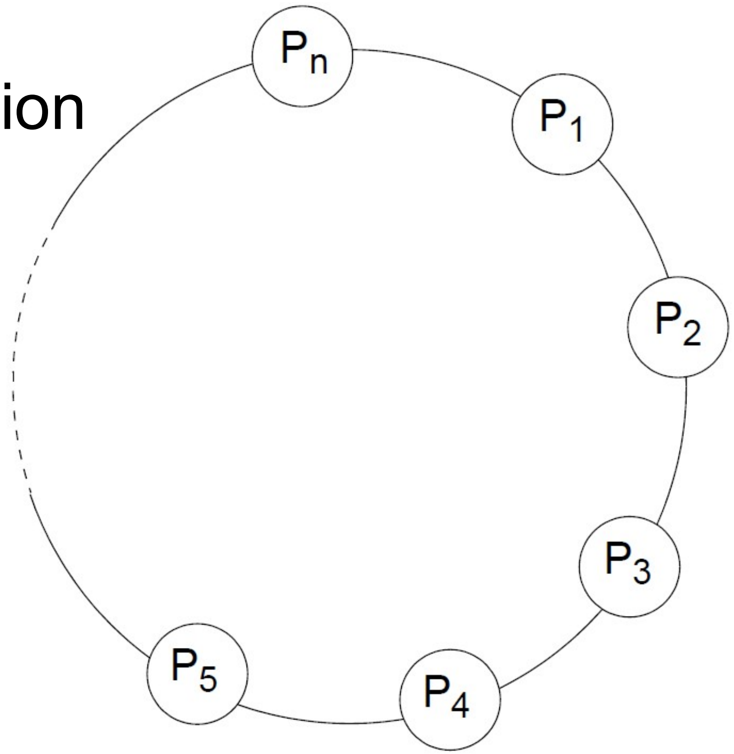


Ricart-Agrawala Second Algorithm

- The complexity is reduced compared to the (first) Ricart-Agrawala algorithm:
  - it requires *n* messages for entering a CS:
    (*n*-1) requests and one reply.
- The failure of a process, except the one which holds the token, does not prevent progress.

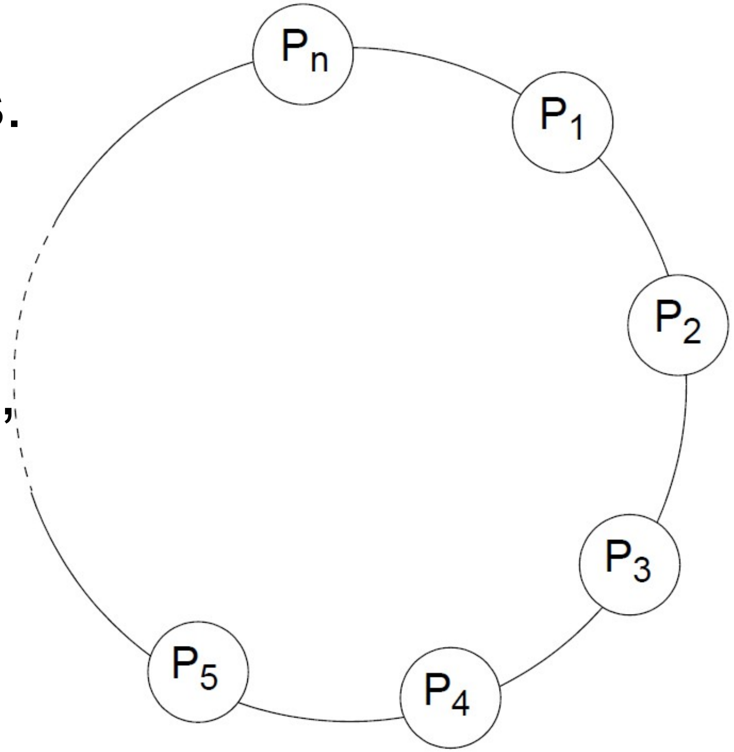
Token Ring Algorithm

- A very simple way to solve mutual exclusion
- Arrange the *n* processes $P_1, P_2, \ldots, P_n$ in a **logical ring**.
  - The logical ring topology is created by giving each process the address of one other process, which is its **neighbour** in the clockwise direction.
  - The logical ring topology is **unrelated to the physical interconnections between the computers**.



Token Ring Algorithm

- The token is initially given to one process.
- The token is passed from one process to its neighbour round the ring.
- When a process requires to enter the CS, it waits until it receives the token from its left neighbour and it retains it;
  - after it got the token, it enters the CS;
  - after it left the CS, it passes the token to its neighbour in clockwise direction.



- When a process receives the token but does not require to enter the critical section, it immediately passes the token over along the ring.

Token Ring Algorithm

- It can take from 1 to *n*-1 messages to obtain a token.
- Messages are sent around the ring even when no process requires the token → additional load on the network.

- The algorithm works well in heavily loaded situations, when there is a high probability that the process which gets the token wants to enter the CS.
  It works poorly in lightly loaded cases.

- If a process fails, no progress can be made until a **reconfiguration** is applied to extract the process from the ring.
- If the process holding the token fails, a unique process has to be picked, which will regenerate the token and pass it along the ring
  - An **election algorithm** has to be run for this purpose.

Distributed Election

Election

- Many distributed algorithms require one process to act as a **coordinator** or, in general, perform some **special role**.
- Examples with mutual exclusion:
  - *Central coordinator algorithm*:
    at initialisation or whenever the coordinator crashes, a new coordinator has to be elected.
  - *Token based algorithms*:
    when the process holding the token fails, a new process has to be elected which generates the new token.

Election

- We consider that it does not matter *which* process is elected.
  - What is important is that **one and only one process is chosen** (we call this process the *coordinator*)
    and **all processes agree on this decision**.

- We assume that each process has a unique number (identifier);
  - in general, election algorithms attempt to locate the process with the highest number, among those which currently are up.

- **Election** is typically started after a failure occurs.
  The detection of a failure (e.g. the crash of the current coordinator) is normally based on time-out

  → a process that gets no response for a period of time suspects a failure and initiates an election process.

- An election process is typically performed in **two phases**:
  1. Select a leader with the highest priority.
  2. Inform all processes about the winner.

The Bully Algorithm

- Each process knows the identifiers of all processes (but not which one is still up!); **the process alive with the highest identifier is selected**.
- Any process could fail even *during* the election procedure.
- P_i detects a failure → coordinator has to be elected:
  - it **send**s an *election* message to all processes with higher identifier and **wait**s for an *answer* message:
    - If no answer arrives within a time limit, P_i becomes the coordinator (as all processes with higher identifier are down)

      → it **broadcast**s a *coordinator* message to all processes to let them know.
    - If an *answer* message arrives, P_i knows that another process has to become the coordinator

      → it **wait**s in order to receive the *coordinator* message. If this message fails to arrive within a time limit (which means that a potential coordinator crashed after sending the *answer* message), P_i resends the *election* message.
- When receiving an *election* message, a process P_j replies with an *answer* message and starts an election procedure itself, unless it has already started one

  → it **send**s an *election* message to all processes with higher identifier.
- Eventually, all processes get an *answer* message, except the one which becomes the coordinator.

The Bully Algorithm

By default, the state of a process is *ELECTION-OFF*

Rule for election process initiator:

/* performed by a process P_i, that triggers the election procedure, or that starts an election after receiving itself an *election* message for the first time */

[RE1]: $state_{P_i}$:= *ELECTION-ON*.

P_i sends an *election* message to all processes with a higher identifier.

P_i waits for *answer* message.

if no *answer* message arrives before time-out **then**

  P_i is coordinator and sends a *coordinator* message to all processes

else

  P_i waits for a *coordinator* message to arrive.

  if no *coordinator* message arrives before time-out **then**

    restart election procedure according to RE1

  end if

end if

The Bully Algorithm

Rule for handling an incoming *election* message:

/* performed by a process P_i at reception of an *election* message from P_j */

[RH1]: P_i replies with an *answer* message to P_j.

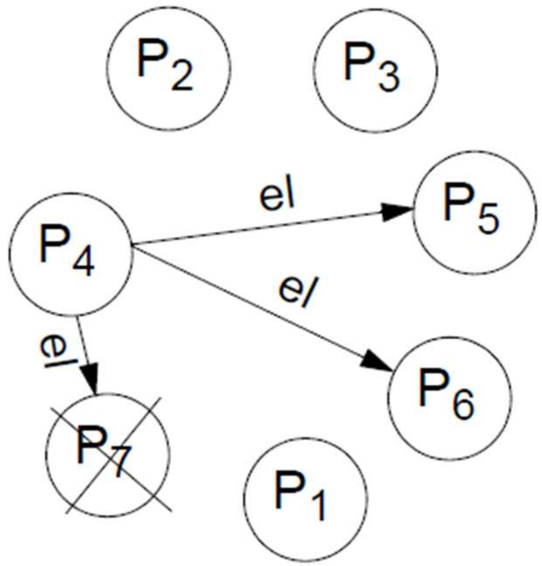
[RH2]: **if** $state_{P_i}$:= *ELECTION-OFF* **then**

start election procedure according to RE1

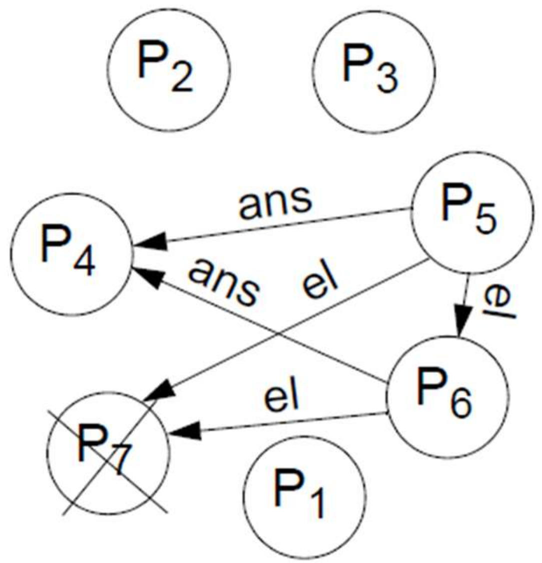
end if

The Bully Algorithm

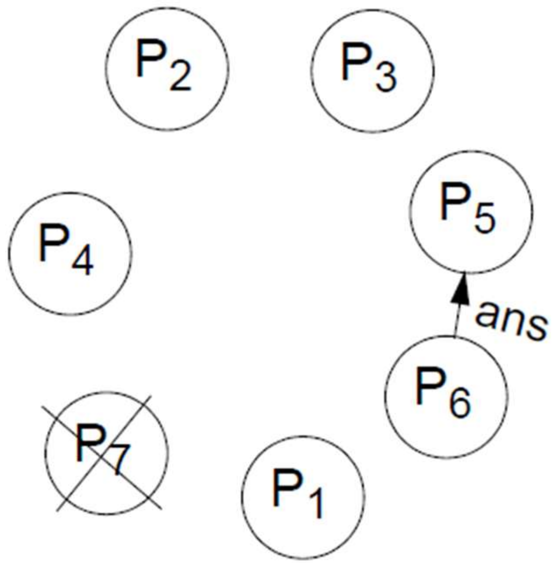
Example:
P_4 suspects that the previous coordinator P_7 has crashed, and starts the election process.



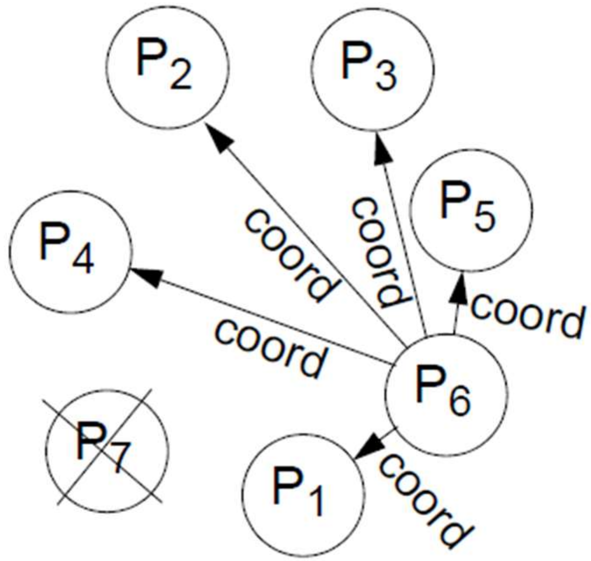
The Bully Algorithm



The Bully Algorithm



The Bully Algorithm

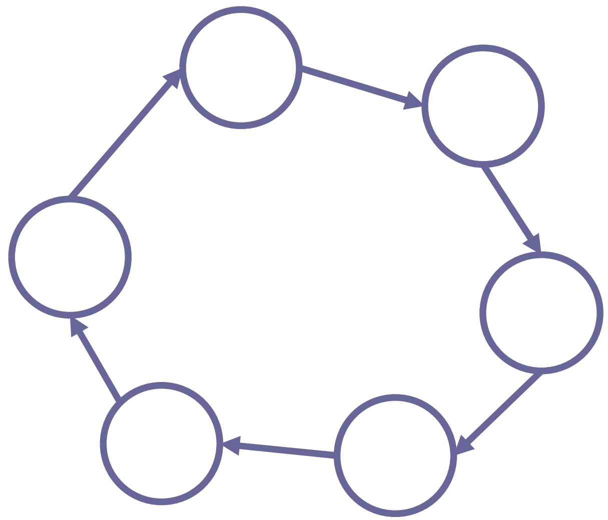


If P_6 crashes before sending the *coordinator* message,
P_4 and P_5 restart the election process.

The best case: the process with the second-highest identifier notices the coordinator's failure.
It can immediately select itself and then send *n*-2 *coordinator* messages.

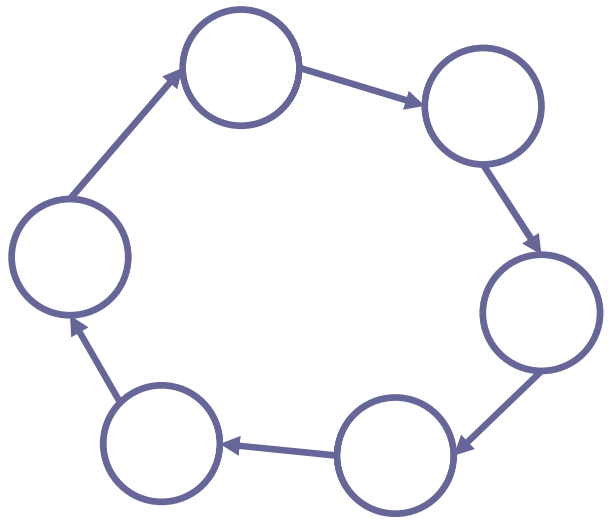
The worst case: the process with the lowest identifier initiates the election.
It sends *n*-1 election messages to processes which themselves initiate each one an election $\Rightarrow$ $O(n^2)$ messages.

The Ring-Based Algorithm



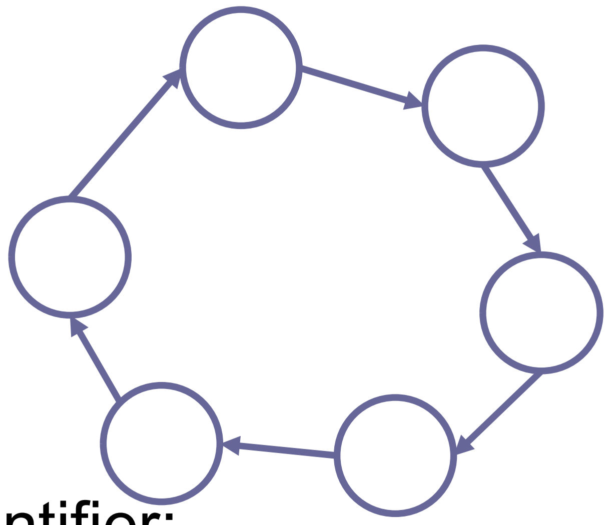
- We assume that the processes are arranged in a **logical ring**
  - Overlay network – only used for election
  - Each process knows the address of one other process, which is its neighbour in the clockwise direction.
  - Each process must also know the next neighbor of that neighbor, so that a crashed process (e.g. coordinator) can be bypassed and the ring be closed again.
    → Exercise: extend the algorithm below to update this information
- The algorithm elects a single coordinator, which is the process in the ring with the highest identifier.

The Ring-Based Algorithm



- Election is started by a process
  which has noticed that the current coordinator has failed.
  The process places its identifier in an *election* message that is passed to the following process.
- When a process receives an *election* message,
  it compares the identifier in the message with its own.
  - If the arrived identifier is greater, it forwards the received *election* message to its neighbour;
  - If the arrived identifier is smaller, it substitutes its own identifier in the *election* message before forwarding it.
  - If the received identifier is that of the receiver itself
    → the receiver process is the coordinator.
    The process sends a *coordinator* message through the ring.

The Ring-Based Algorithm



For an election:

- **On average:**
  - *n*/2 (*election*) messages needed to reach the node with maximal identifier;
  - *n* (*election*) messages to return to this node;
  - *n* messages to rotate *coordinator* message.

  → Number of messages: 2*n* + *n*/2.

- **Worst case:**
  - *n*-1 messages needed to reach the maximal node;

  → Number of messages: 3*n* - 1.

The ring algorithm is more efficient on average than the bully algorithm.

General Observation

Ring-based algorithms

- e.g. token ring mutex algorithm, ring election

vs. Multicast-based algorithms

- e.g. bully algorithm, Ricard-Agrawala first algorithm:

Ring-based algorithms trade fewer total messages for longer latency

Acknowledgments

- Most of the slide contents is based on a previous version by Petru Eles, IDA, Linköping University.