Unit 3:
Synchronization, Coordination and Agreement

References:

1.G. Coulouris, J. Dollimore and T. Kindberg; **Distributed Systems Concepts and Design,4th Edition.**

2.Andrew S. Tanenbaum and Maarten van Steen**; Distributed Systems: Principles and Paradigms, 2nd Edition.**

Synchronization, Coordination and Agreement

3.1 Time Synchronization

3.1.1 Time and time synchronization

3.1.2 Physical Clock Synchronization

3.1.2.1 Berkeley Algorithm

3.1.2.2 Cristian's Algorithm

3.1.2.3 Network Time Protocol (NTP)

3.1.3 Logical Clocks Synchronization

3.1.3.1 Events and ordering

3.1.3.2 Partical, causal and total ordering

3.1.3.3 Global State and State Recording

3.1.3.4 Lamport logical clock

3.1.3.5 Vector clock

3.2 Distributed Co-ordination

3.2.1 Distributed mutual exclusion

3.2.2 Mutual Exclusion Algorithms

3.2.2.1 Central Coordinator Algorithm

3.2.2.2 Token Ring Algorithm

3.2.2.3 Lamport's Algorithm

3.2.2.4 Ricart-Agrawala
(Non-Token based & token based)

3.2.3 Distrbuted Leader Election

3.2.3.1 Bully algorithm

3.2.3.2 Ring Algorithms

Mutual Exclusion in DS

Mutual exclusion in DS

There are three major communication scenarios:

1. One-way Communication usually do not need mutual exclusion.
2. Client/server communication is for multiple clients making service request to a shared server. If co-ordination is required among the clients, it is handled by the server and there is no explicit interaction among client process – and hence no need of mutual exclusion
3. Inter process communication: processes need to exchange information to reach some conclusion about the system or some agreement among the cooperating processes.

Background of Mutual Exclusion

- Inter Process Communication (IPC)
- Need to Access a Shared Resource (Memory)
- Critical Section (Region)
  - A *critical section is a section code or region in which a process(or thread) competes in a potentially* destructive way with another process(thread) for access to a shared data item or file
- Race Condition
  - a problem caused because of accessing the Critical Section by two or more processes at the same time.
- Mutual Exclusion
  - a solution to the Race Condition

Definition of Mutual Exclusion

- If two processes are allowed to concurrently be in competing critical sections, then incorrect results may be computed – called a **Race Condition**

- The process of ensuring that this destructive interaction does no occur is called ***mutual exclusion*** **(mutex).**

- In other words, **mutual exclusion** is the process of making only one process to enter into the critical section at the same time.

Mutual Exclusion

- Very well-understood in shared memory systems

- Requirements:
  - at most one process in critical section (safety)
  - if more than one requesting process, someone enters (liveness)
  - a requesting process enters within a finite time (no starvation)
  - requests are granted in order (fairness)

Mutual Exclusion Algorithms

There are two basic approaches to distributed mutual exclusion:

1. Non-token-based (permission based):
   - each process freely and equally competes for the right to use the shared resource; requests are arbitrated 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.

Mutual Exclusion Algorithms

1. Non-token based (permission based) Algorithms:
   A. Permission from a central Coordinator
      I. **Central Coordinator Based Algorithm**
   B. Permission from all (distributed) processes
      I. **Lamport Algorithm**
      II. **Ricart Agarwala Algorithm**

2. Token Based Algorithm
   I. **Token Ring Algorithm**
   II. **Ricart Agarwala Second Algorithm**
   III. *Suzuki Kasami Algorithm*
   IV. *Raymond Algorithm*
   V. *Singhal Algorithm etc.*

Central Coordinator Based Algorithm

- A central coordinator grants permission to enter a CS
- In this algorithm, a process that wants to enter into a CS has to take a permission from a central coordinator
- The scheme is simple and easy to implement.
- The strategy requires only three messages per use of a CS *(request, OK, release).*

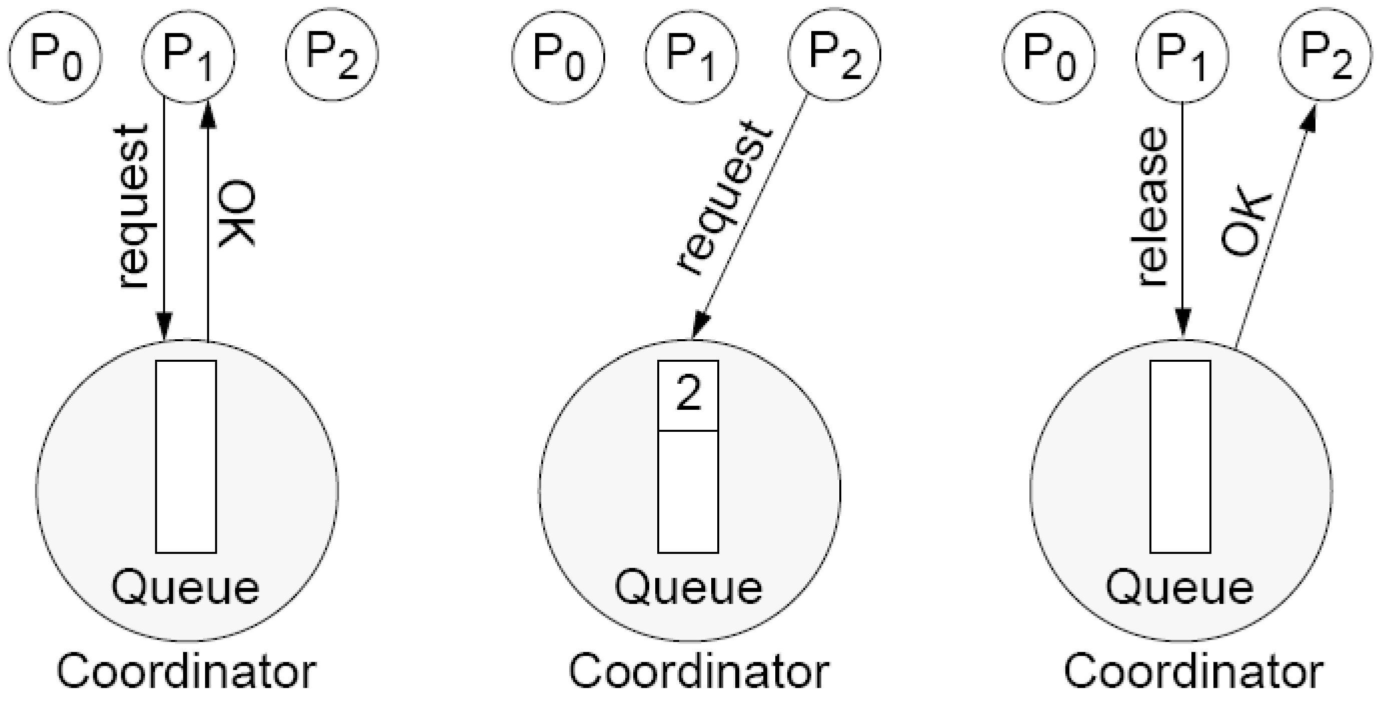
Algorithm:

1. 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).

2. The reply from the coordinator gives the right to enter the CS.

3. After finishing work in the CS the process notifies the coordinator with a release message.

Example:



P0
P1
P2
request
OK
Queue
Coordinator
P0
P1
P2
request
2
Queue
Coordinator
P0
P1
P2
release
OK
Queue
Coordinator

Central Coordinator Based Algorithm: Pseudo code

- Coordinator

```
loop
  receive(msg);
    case msg of
      REQUEST: if nobody in CS
               then reply GRANTED
               else queue the REQ;
                    reply DENIED
      RELEASE: if queue not empty then
                remove 1st on the queue
                reply GRANTED //modified algo
    end case
end loop
```

- Client

```
send(REQUEST);
receive(msg);
while msg != GRANTED then receive(msg);
enter CS;
send(RELEASE)
```

Central Coordinator Based Algorithm

- Advantages
  - guarantees mutual exclusion
  - It is fair (First Come First Served)
  - Easy to implement
- Disadvantages
  - 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 h*as to be run in order to choose one and only one new coordinator.
    - if no explicit DENIED message, then cannot distinguish permission denied from a dead coordinator

Mutual Exclusion Algorithms

1. Non-token based (permission based) Algorithms:
   A. Permission from a central Coordinator
      I. **Central Coordinator Based Algorithm**
   B. Permission from all (distributed) processes
      I. **Lamport Algorithm**
      II. **Ricart Agarwala Algorithm**

2. <u>Token Based Algorithm</u>
   I. **Token Ring Algorithm**
   II. **Ricart Agarwala Second Algorithm**
   III. *Suzuki Kasami Algorithm*
   IV. *Raymond Algorithm*
   V. *Singhal Algorithm etc.*

Permission from all (distributed) processes

- In a distributed environment it seems more natural to implement mutual exclusion, based upon distributed agreement - not on a central coordinator. The basic algorithm goes like this:

1. Two processes want to enter the same critical region at the same moment.
2. Both send request messages to all processes
3. All events are time-stamped by the global ordering algorithm
4. The process whose request event has smaller time-stamp wins
5. Every process must respond to request messages

Permission from all (distributed) processes – Distributed Algorithm

I. **Lamport's Algorithm**

II. **Ricart Agarwala Algorithm**

Lamport's Algorithm

- Every node *i* has a request queue q_i
  - keeps requests sorted by logical timestamps (total ordering enforced by including process id in the timestamps)

- To request critical section:
  - send timestamped REQUEST(*tsi*, *i*) to all other nodes
  - put (*tsi*, *i*) in its own queue

- On receiving a request (*tsi*, *i*): at j
  - send timestamped REPLY to the requesting node i
  - put request (*tsi*, *i*) in the queue q_j REPLY(ts j, j)

Lamport's Algorithm contd..

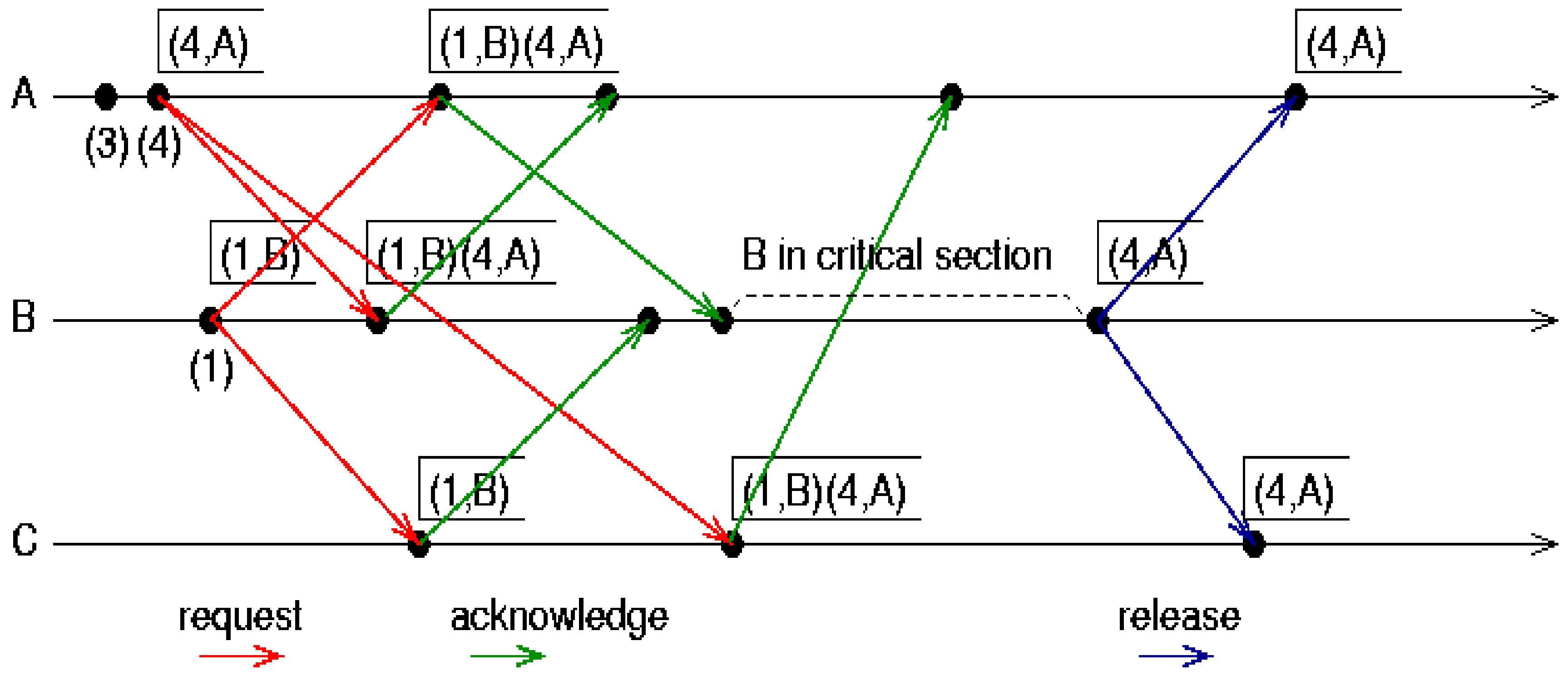
- To enter critical section:
  - Process *i* enters critical section if:
    - (*tsi*, *i*) is at the top if its own queue, and
    - Process *i* has received a message (any message) with timestamp larger than (*tsi*, *i*) from ALL other nodes.

- To release critical section:
    - Process *i* removes its request from its own queue and sends a timestamped RELEASE message to all other nodes
    - On receiving a RELEASE message from *i*, *i*'s request is removed from the local request queue

Some notable points

- Purpose of REPLY messages from node *i* to *j* is to ensure that *j* knows of all requests of *i* prior to sending the REPLY (and therefore, possibly any request of *i* with timestamp lower than *j*'s request)
- Requires FIFO channels.
- $3(n – 1)$ messages per critical section invocation
- Synchronization delay = max mesg transmission time
- Requests are granted in order of increasing timestamps

Example


(4,A)
(1,B)(4,A)
(4,A)
A
(3) (4)
(1,B)
(1,B)(4,A)
B in critical section
(4,A)
B
(1)
(1,B)
(1,B)(4,A)
(4,A)
C
request
acknowledge
release

Permission from all (distributed) processes – Distributed Algorithm

I. **Lamport's Algorithm**

II. **Ricart Agarwala Algorithm**

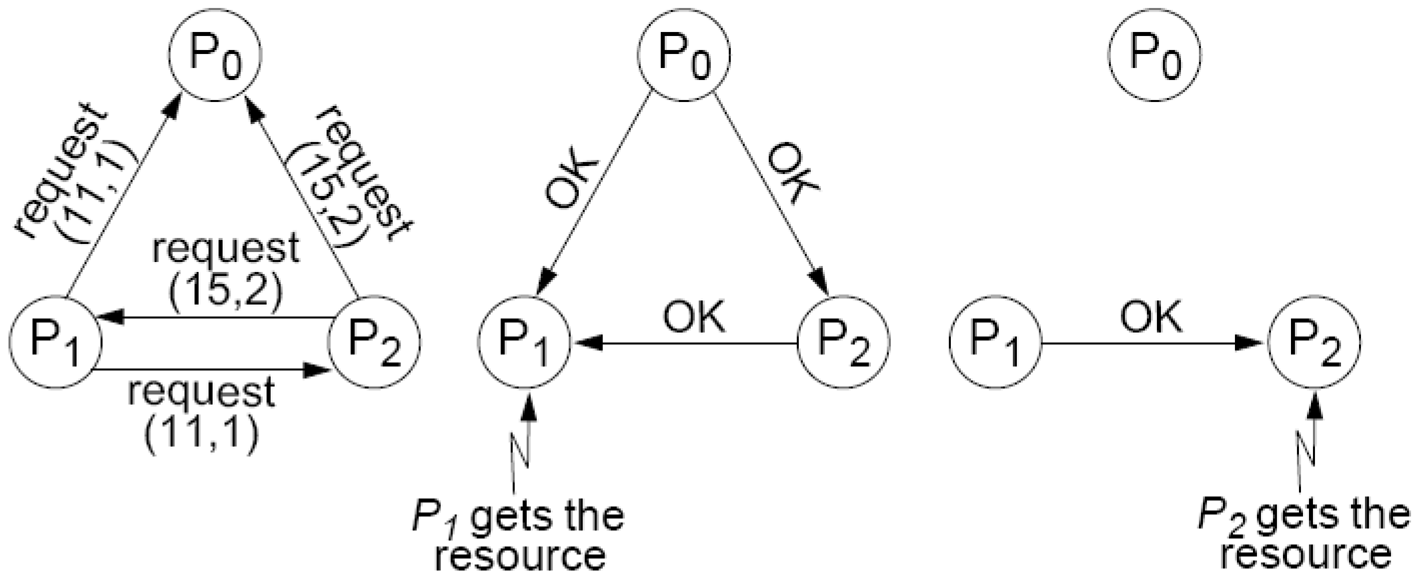
The Ricart-Agrawala Algorithm

- Improvement over Lamport's
- Main Idea:
  - node *j* need not send a REPLY to node *i* if *j* has a request with timestamp lower than the request of *i* (since *i* cannot enter before *j* anyway in this case)
- Does not require FIFO
- 2(n – 1) messages per critical section invocation
- Synchronization delay = max. message transmission time
- Requests granted in order of increasing timestamps

The Ricart-Agrawala Algorithm

- To request critical section:
  - send timestamped REQUEST message (*tsi*, *i*)
- On receiving request (*tsi*, *i*) at *j*:
  - send REPLY to *i* if *j* is neither requesting nor executing critical section or
  - if *j* is requesting and *i*'s request timestamp is smaller than *j*'s request timestamp. Otherwise, defer the request.
- To enter critical section:
  - *i* enters critical section on receiving REPLY from all nodes
- To release critical section:
  - send REPLY to all deferred requests

Example:


P0
P1
P2
request (11,1)
request (15,2)
request (15,2)
request (11,1)
OK
OK
OK
P1 gets the resource
OK
P2 gets the resource

Mutual Exclusion Algorithms

1. Non-token based (permission based) Algorithms:
   - A. Permission from a central Coordinator
     - I. **Central Coordinator Based Algorithm**
   - B. Permission from all (distributed) processes
     - I. **Lamport Algorithm**
     - II. **Ricart Agarwala Algorithm**

2. <u>Token Based Algorithm</u>
   - I. **Token Ring Algorithm**
   - II. **Suzuki Kasami Algorithm**
   - III. **Raymond Algorithm**
   - IV. **Singhal Algorithm** etc.

Token based Algorithms

- Single token circulates, enter CS when token is present
- Mutual exclusion obvious
- Algorithms differ in how to find and get the token
- Uses sequence numbers rather than timestamps to differentiate between old and current requests

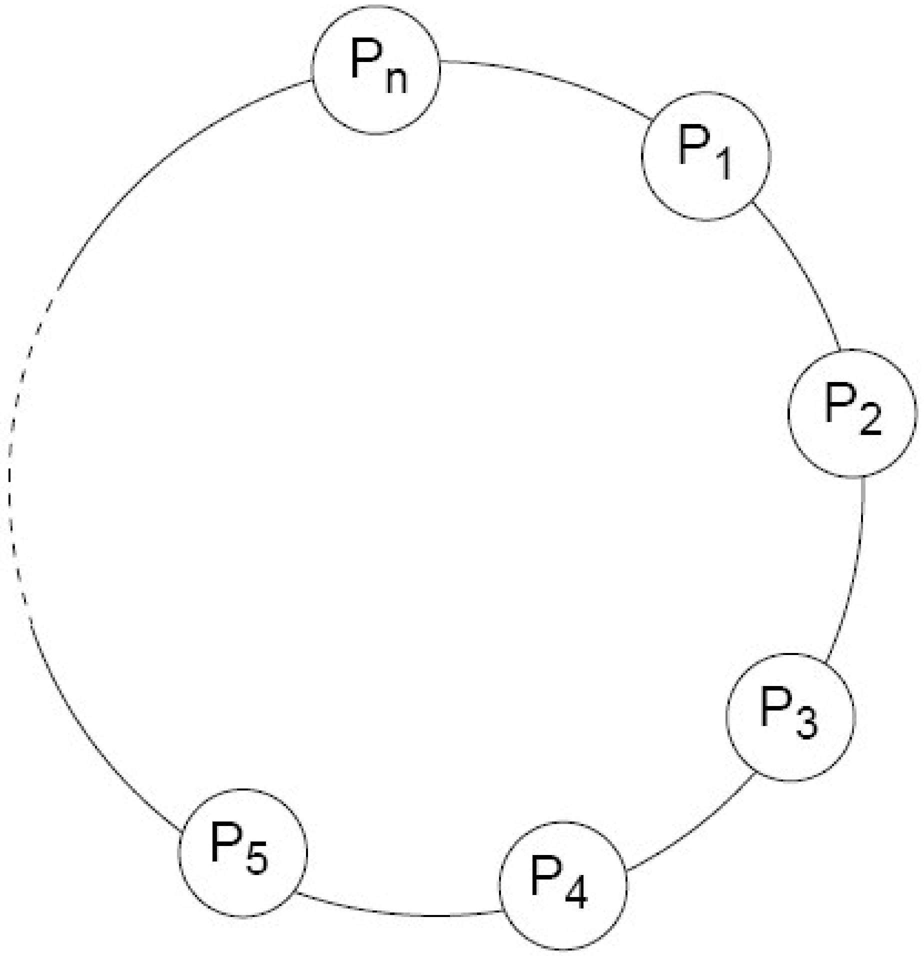
Token Based Algorithms

- A token is circulated in a logical ring.
- A process enters its CS if it has the token.
- Issues:
  - If the token is lost, it needs to be regenerated.
  - Detection of the lost token is difficult since there is no bound on how long a process should wait for the token.
  - If a process can fail, it needs to be detected and then by-passed.
  - When nobody wants to enter, processes keep on exchanging messages to circulate the token

Token Ring Algorithm

- A very simple way to solve mutual exclusion ⇒ arrange the *n* processes P_1, P_2, ... 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



Token Ring Algorithm

The 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 then 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* (see later) has to be run for this purpose.

Mutual Exclusion Algorithms

1. Non-token based (permission based) Algorithms:
   A. Permission from a central Coordinator
      I. **Central Coordinator Based Algorithm**
   B. Permission from all (distributed) processes
      I. **Lamport Algorithm**
      II. **Ricart Agarwala Algorithm**

2. <u>Token Based Algorithm</u>
   I. **Token Ring Algorithm**
   II. **Ricart Agarwala Second Algorithm**
   III. *Suzuki Kasami Algorithm*
   IV. *Raymond Algorithm*
   V. *Singhal Algorithm etc.*

Ricart-Agrawala Second Algorithm

- A process is allowed to enter the critical section when it got the token. In order to get the token it 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.
- Initially the token is assigned arbitrarily to one of the processes.
- When a process Pi leaves a critical section it passes the token to one of the processes which are waiting for it; this will be the first process Pj, where j is searched in order [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] for which there is a pending request.
- If no process is waiting, Pi 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 Pi find out if there is a pending request?
  - Each process Pi records the timestamp corresponding to the last request it got from process Pj, in requestPi[j]. In the token itself, token[j] records the timestamp (logical clock) of Pj's last holding of the token. If requestPi[j] > token[j] then Pj has a pending request.

Ricart-Agrawala Second Algorithm

The Algorithm

Rule for process initialization

/* performed at initialization */

- [RI1]: statePi:= NO-TOKEN for all processes Pi, except one single process Px for which statePx := TOKEN-PRESENT.
- [RI2]: token[k] initialized 0 for all elements k= 1 .. n.

Request Pi[k] initialized 0 for all processes Pi and all elements k= 1 .. n.

Rule for access request and execution of the CS

/* performed whenever process Pi requests an access to the CS and when it finally gets it; in particular Pi can already possess the token */

- [RA1]: **if** statePi = NO-TOKEN **then**

    Pi sends a request message to all processes; the message is of the form (TPi, i), where TPi= Cpi is the value of the local logical clock, and i is an identifier of Pi.

    Pi waits until it receives the token.

    end if.

    statePi := TOKEN-HELD.

    Pi enters the CS .

Ricart-Agrawala Second Algorithm

Rule for handling incoming requests

```
        /* performed by Pi whenever it received a request ( TPj, j)
from Pj */
• [RH1]: request[ j] := max( request[ j], TPj).
• [RH2]: if statePi = TOKEN-PRESENT then Pi releases the
  resource (see rule RR2).
        end if.
```

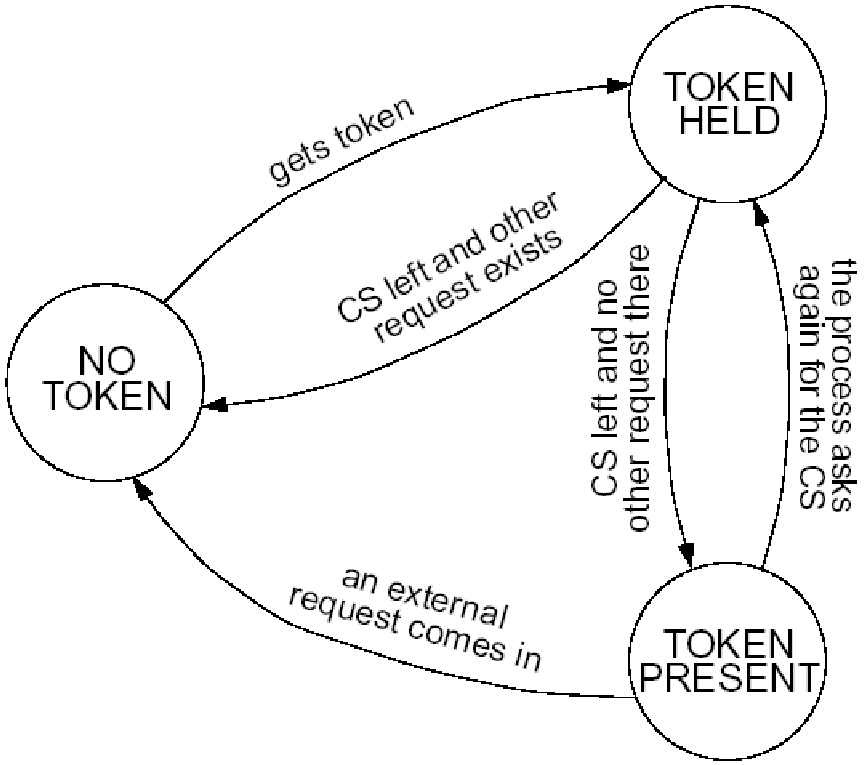
Rule for releasing a CS

```
/* performed by Pi after it finished work in a CS or when it holds a
token without using it and it got arequest */
• [RR1]: statePi = TOKEN-PRESENT.
• [RR2]: for k = [ i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] do
        if request[k] > token[k] then
        statePi := NO-TOKEN.
        token[ i] := CPi, the value of the local logical clock.
        Pi sends the token to Pk.
break. /* leave the for loop */
end if.
end for.
```

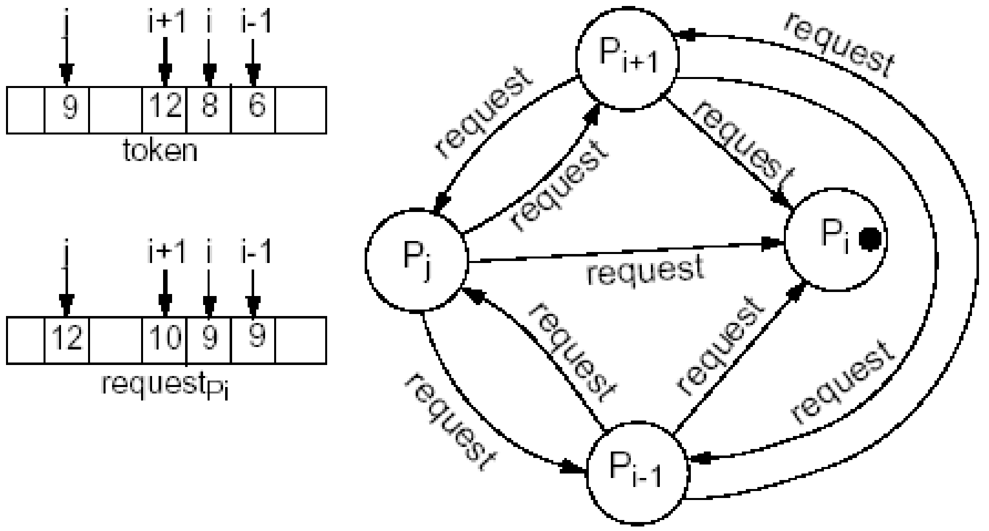
Ricart-Agrawala Second Algorithm

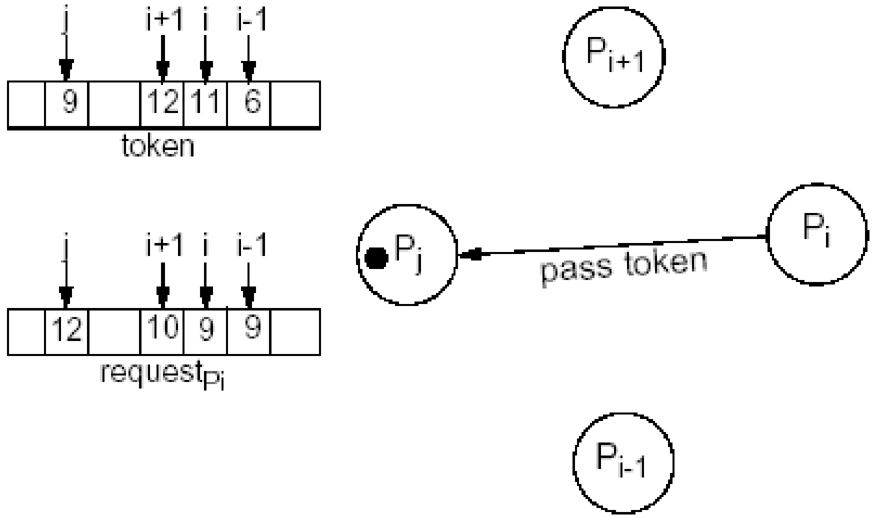
Each process keeps its state with respect to the token:

NO-TOKEN, TOKEN-PRESENT, TOKEN-HOLD.


TOKEN HELD
NO TOKEN
TOKEN PRESENT
gets token
CS left and other request exists
CS left and no other request there
the process asks again for the CS
an external request comes in

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, doesn’t prevent progress.

Election Algorithm

Election

- In distributed computing, leader election is the process of designating a single process as the organizer of some task distributed among several computers (nodes).
- Before the task is begun, all network nodes are unaware which node will serve as the "leader," or coordinator, of the task.
- After a leader election algorithm has been run, however, each node throughout the network recognizes a particular, unique node as the task leader.

Election Algorithm

Many distributed algorithms require one process to act as a coordinator or, in general, perform some special role, and that coordinator is selected using an Election Algorithm

Need of Election Algorithm: Few Examples

1. Clock Synchronization
   - Berkeley Algorithm: To select a leader or master to take the responsibility of averaging the time.

2. Mutual Exclusion
   - Central coordinator algorithm: at initialization or whenever the coordinator crashes, a new coordinator has to be elected
   - Token ring algorithm: when the process holding the token fails, a new process has to be elected which generates the new token

3. Any Distributed Computing
   - A **distributed algorithm** is an algorithm, run on a distributed system, that does not assume the previous existence of a central coordinator.
   - Need to select a master to take the responsibility of distributing the sub-problems among the slaves and collecting the partial results from them

Election Algorithm: Basic Concepts (1)

- Determine where a new copy of the coordinator should be restarted.
- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process P_i is i.
- Assume a one-to-one correspondence between processes and sites.
- The coordinator is always the process with the largest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number.

Election Algorithm: Basic Concepts (2)

☞ We consider that it doesn't 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 $\Rightarrow$ a process that gets no response for a period of time suspects a failure and initiates an election process.

Phases of Any Election Algorithm

☞ An election process is typically performed in two phases:

1. **Select a leader with the highest priority.**
2. **Inform all processes about the winner**

Election Algorithms

Here we will discuss the following two basic Election algorithms

1. Bully Algorithm

2. Chang and Roberts algorithm (Ring Algorithm)

Bully Algorithm

- This algorithm is applicable to elect a leader in a distributed system connected with each other (say in a mesh topology).

- The bully algorithm is a method in distributed computing for dynamically selecting a coordinator by process ID number.

- **Assumptions**
    - The system is synchronous and uses timeout for identifying process failure

Bully Algorithm

- Message types
  - Election Message : Sent to announce the election
  - Answer Message : Respond to the election message
  - Coordinator message: sent to announce the identity elected process

- Compare with Ring algorithm:
  - Assumes that system is synchronous
  - Uses timeout to detect process failure/crash
  - Each processor knows which processor has the higher identifier number and communicates with that

Bully Algorithm: Basic Steps

- When a process P determines that the current coordinator is down because of message timeouts or failure of the coordinator to initiate a handshake, it performs the following sequence of actions:
  1. P broadcasts an election message (inquiry) to all other processes with higher process IDs.
  2. If P hears from no process with a higher process ID than it, it wins the election and broadcasts victory.
  3. 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.
  4. 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.
- 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.

Bully Algorithm: Detailed Algorithm

- Applicable to systems where every process can send a message to every other process in the system.
- If process P_i sends a request that is not answered by the coordinator within a time interval *T*, assume that the coordinator has failed; P_i tries to elect itself as the new coordinator.
- P_i sends an election message to every process with a higher priority number, P_i then waits for any of these processes to answer within *T*.

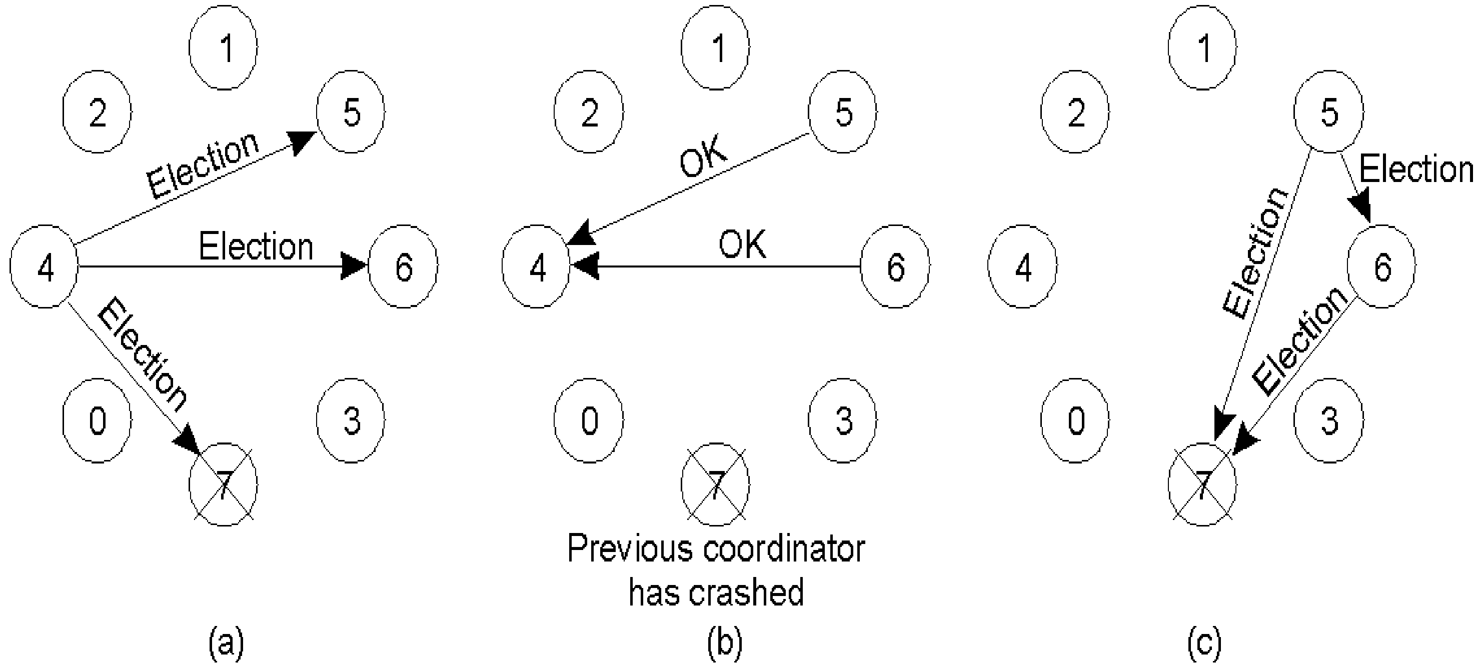
Bully Algorithm (Cont.)

- If no response within *T*, assume that all processes with numbers greater than i have failed; P_i elects itself the new coordinator.
- If answer is received, P_i begins time interval T', waiting to receive a message that a process with a higher priority number has been elected.
- If no message is sent within T', assume the process with a higher number has failed; P_i should restart the algorithm

Bully Algorithm (Cont.)

- If P_i is not the coordinator, then, at any time during execution, P_i may receive one of the following two messages from process P_j.
  - P_j is the new coordinator ($j > i$). P_i, in turn, records this information.
  - P_j started an election ($j > i$). P_i, sends a response to P_j and begins its own election algorithm, provided that *Pi* has not already initiated such an election.
- After a failed process recovers, it immediately begins execution of the same algorithm.
- If there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number.

Bully Algorithm: Example (1)

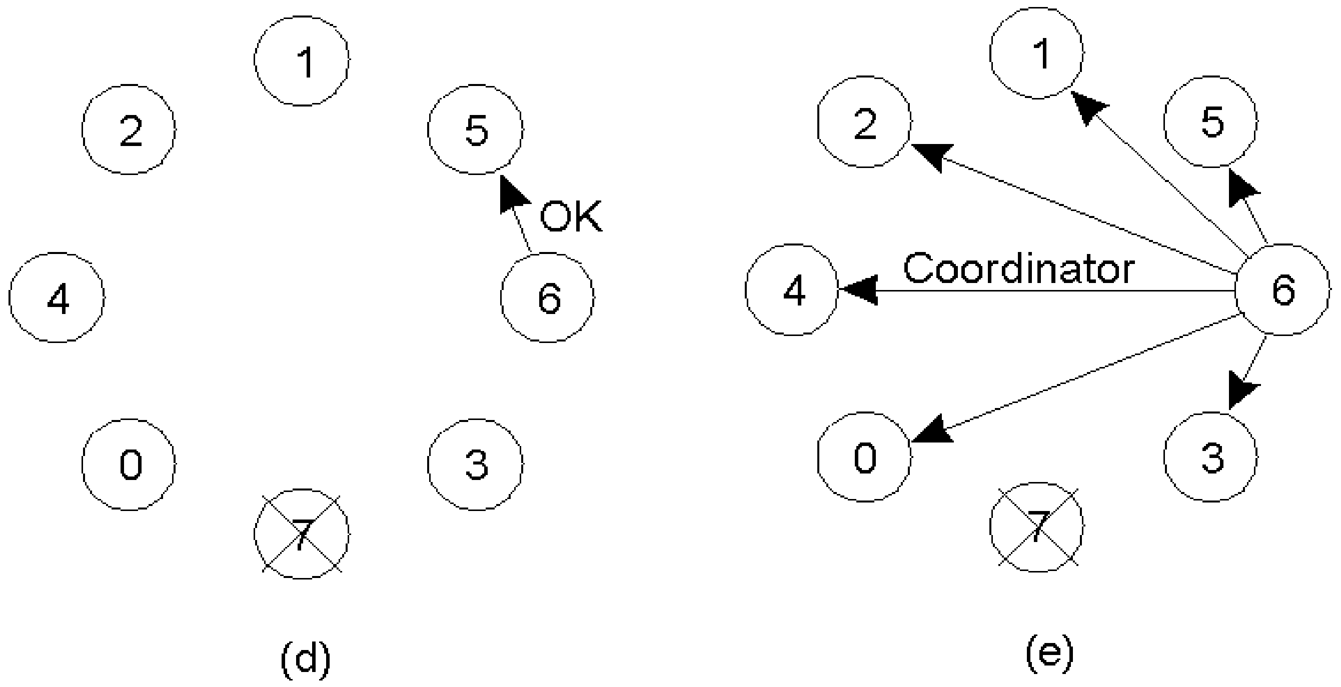


a) **Process 4 holds an election**

b) **Process 5 and 6 respond, telling 4 to stop**

c) **Now 5 and 6 each hold an election**

Bully Algorithm: Example (2)



d) **Process 6 tells 5 to stop**
e) **Process 6 wins and tells everyone**

Election Algorithms

Here we will discuss the following two basic Election algorithms

1. Bully Algorithm
2. Chang and Roberts algorithm (Ring Algorithm)

Chang and Roberts algorithm

- Chang and Roberts is a ring-based election algorithm used to find a process with the largest identification.
- It is a useful method of election in decentralized distributed computing where the systems are connected in a logical or physical ring.
- The algorithm works for any number of processes N, and does not require any process to know how many processes are in the ring.
- Is often referred as a **ring algorithm**.

Ring Algorithm

- Applicable to systems organized as a ring (logically or physically).
- Assumes that the links are unidirectional, and that processes send their messages to their right neighbors.
- Each process maintains an *active list*, consisting of all the priority numbers of all active processes in the system when the algorithm ends.
- If process P_i detects a coordinator failure, P_i creates a new active list that is initially empty. It then sends a message *elect(i)* to its right neighbor, and adds the number *i* to its active list.

Ring Algorithm (Cont.)

- If P_i receives a message elect(j) from the process on the left, it must respond in one of three ways:
  - ✦ If this is the first *elect* message it has seen or sent, P_i creates a new active list with the numbers i and j
    - It then sends the message *elect(i)*, followed by the message *elect(j)*
  - ✦ If $i \neq j$, then the active list for P_i now contains the numbers of all the active processes in the system
    - P_i can now determine the largest number in the active list to identify the new coordinator process
  - ✦ If $i = j$, then P_i receives the message *elect(i)*
    - The active list for P_i contains all the active processes in the system
    - P_i can now determine the new coordinator process.

Ring Algorithm: Example1

- Suppose that we have four processes arranged in a ring: P1 , P2 , P3 , P4 , P1 …
- P4 is coordinator
- Suppose P1 + P4 crash
- Suppose P2 detects that coordinator P4 is not responding
- P2 sets active list to [ ]
- P2 sends “Elect(2)” message to P3; P2 sets active list to [2]
- P3 receives “Elect(2)”
- This message is the first message seen, i.e., P3 is not yet a participant, so P3 sets its active list to [2,3]
- P3 sends “Elect(3)” towards P4 and then sends “Elect(2)” towards P4
- The messages pass P4 + P1 and then reach P2
- P2 adds 3 to active list [2,3]
- P2 forwards “Elect(3)” to P3
- P2 receives the “Elect(2) message
- P2 chooses P3 as the highest process in its list [2, 3] and sends an “Elected(P3)” message
  P3 receives the “Elect(3)” message
- P3 chooses P3 as the highest process in its list [2, 3] + sends an “Elected(P3)” message

The Ring-Based Algorithm

- We assume that the processes are arranged in a logical ring; each process knows the address of one other process, which is its neighbour in the clockwise direction.

- The algorithm elects a single coordinator, which is the process with the highest identifier.

- 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 $\Longrightarrow$ this will be the coordinator. The new coordinator sends an *elected message* through the ring.

The Ring-Based Algorithm (cont'd)

☞ An optimization

- Several elections can be active at the same time. Messages generated by later elections should be killed as soon as possible.

⇩

Processes can be in one of two states: *participant* and *non-participant*. Initially, a process is *non-participant*.

- The process initiating an election marks itself *participant*.
- A *participant* process, in the case that the identifier in the *election message* is smaller than the own, does not forward any message (it has already forwarded it, or a larger one, as part of another simultaneously ongoing election).
- When forwarding an *election message*, a process marks itself *participant*.
- When sending (forwarding) an *elected message*, a process marks itself *non-participant*.

Algorithm

1. Initially each process in the ring is marked as *non-participant*.
2. A process that notices a lack of leader starts an election. It creates an *election message* containing its UID. It then sends this message clockwise to its neighbor.
3. Every time a process sends or forwards an *election message*, the process also marks itself as a *participant.*
4. When a process receives an *election message* it compares the UID in the message with its own UID.

   a) If the UID in the election message is larger, the process unconditionally forwards the *election message* in a clockwise direction.

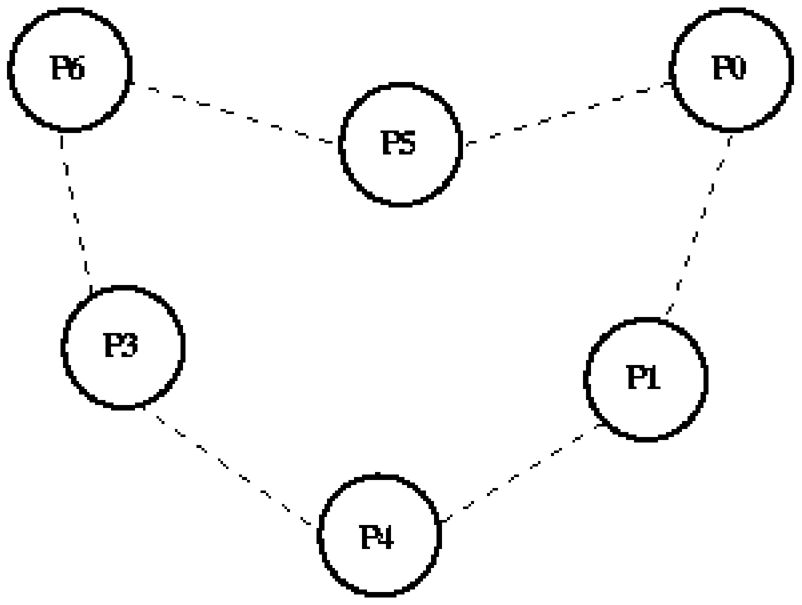
   b) If the UID in the election message is smaller, and the process is not yet a participant, the process replaces the UID in the message with its own UID, sends the updated *election message* in a clockwise direction.

   c) If the UID in the election message is smaller, and the process is already a *participant* (i.e., the process has already sent out an election message with a UID at least as large as its own UID), the process discards the election message.

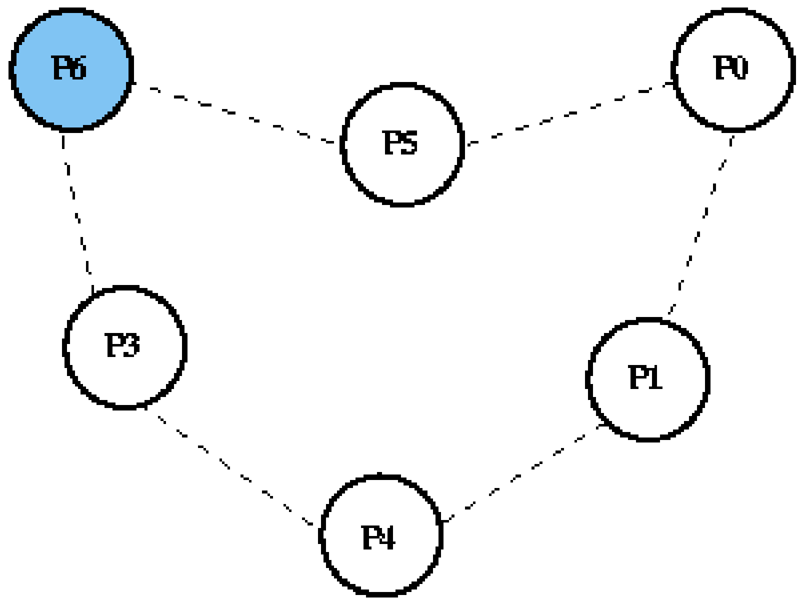
   d) If the UID in the incoming election message is the same as the UID of the process, that process starts acting as the leader.

Algorithm (contd..)

5. When a process starts acting as the leader, it begins the second stage of the algorithm.
   a) The leader process marks itself as *non-participant* and sends an *elected message* to its neighbor announcing its election and UID.
   b) When a process receives an *elected message*, it marks itself as *non-participant*, records the elected UID, and forwards the *elected message* unchanged.
   c) When the *elected message* reaches the newly elected leader, the leader discards that message, and the election is over.



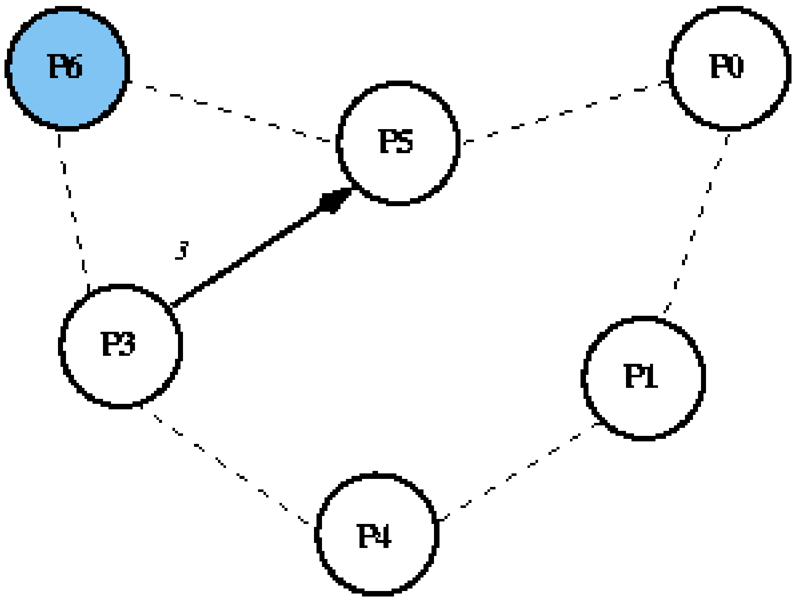
Token Ring Election Algorithm: Step 0



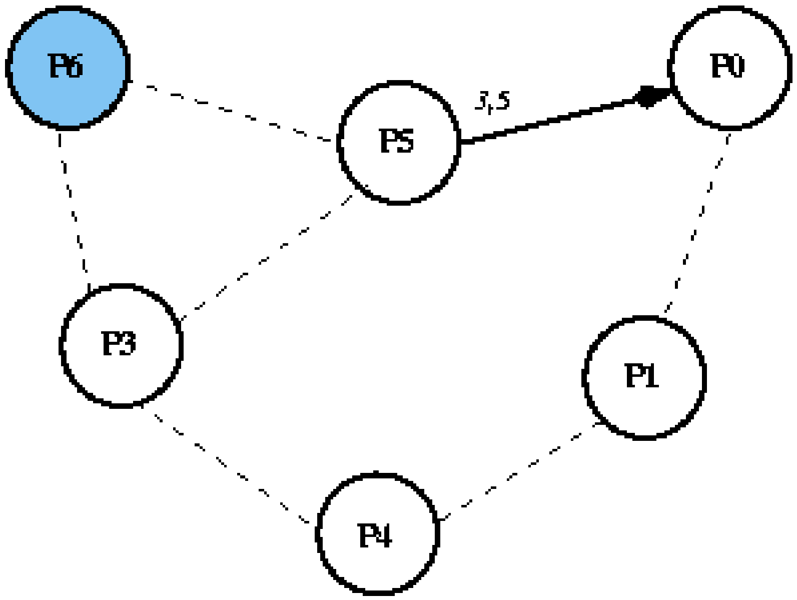
Token Ring Election Algorithm: Step 1

We start with 6 processes, connected in a logical ring. Process 6 is the leader, as it has the highest number.

Process 6 fails



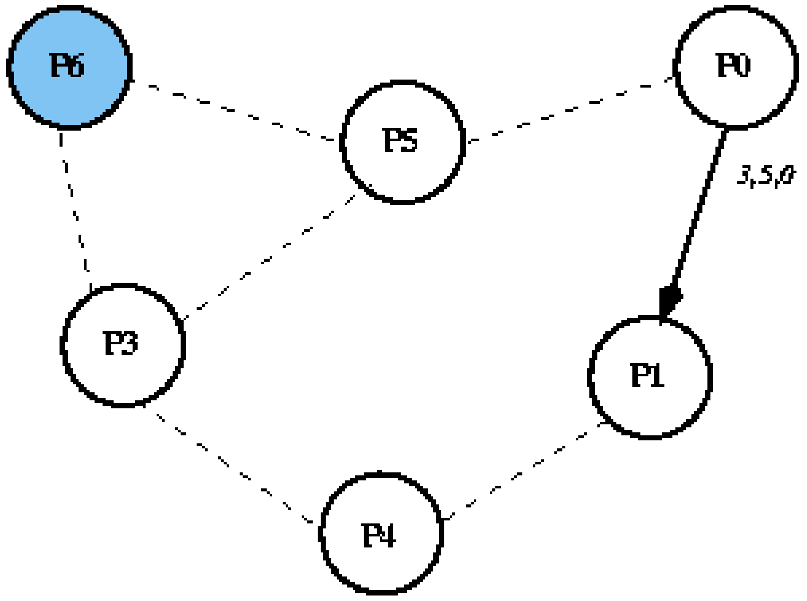
Token Ring Election Algorithm: Step 2



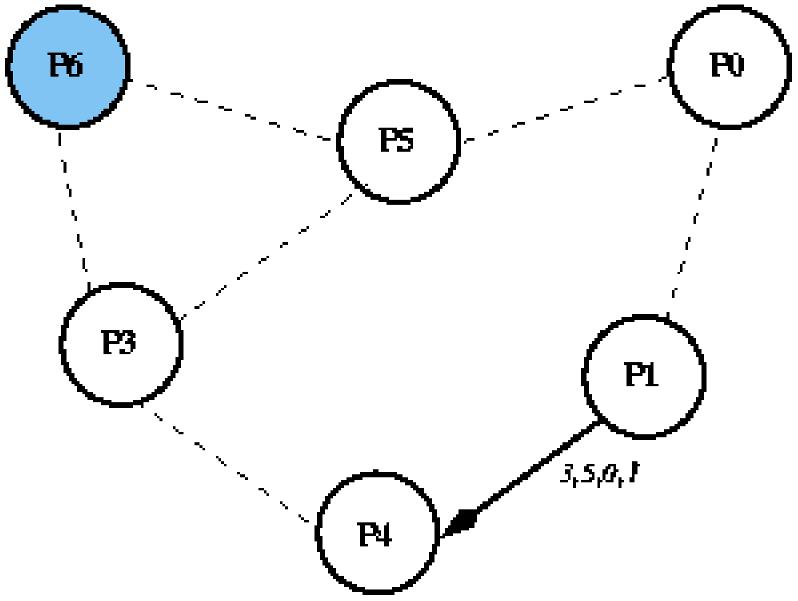
Token Ring Election Algorithm: Step 3

Process 3 notices that Process 6 does not respond So it starts an election, sending a message containing its id to the next node in the ring.

Process 5 passes the message on, adding its own id to the message.



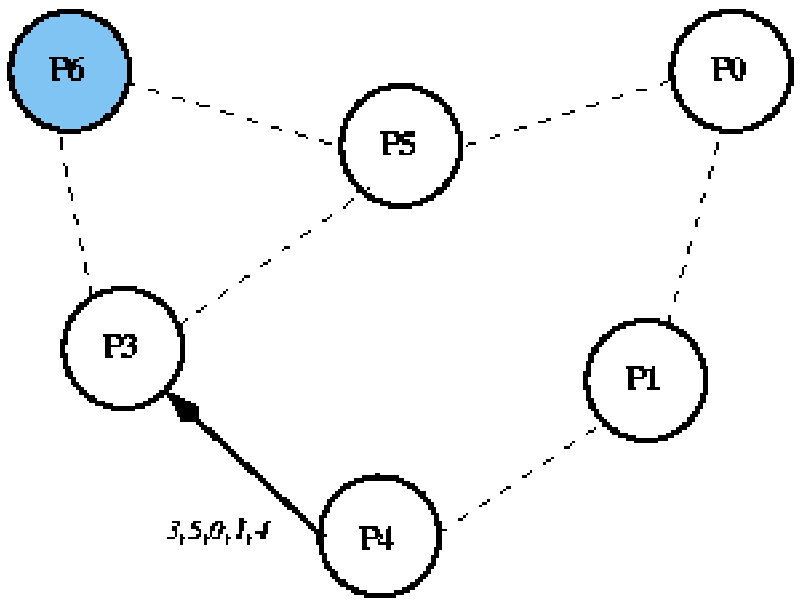
Token Ring Election Algorithm: Step 4



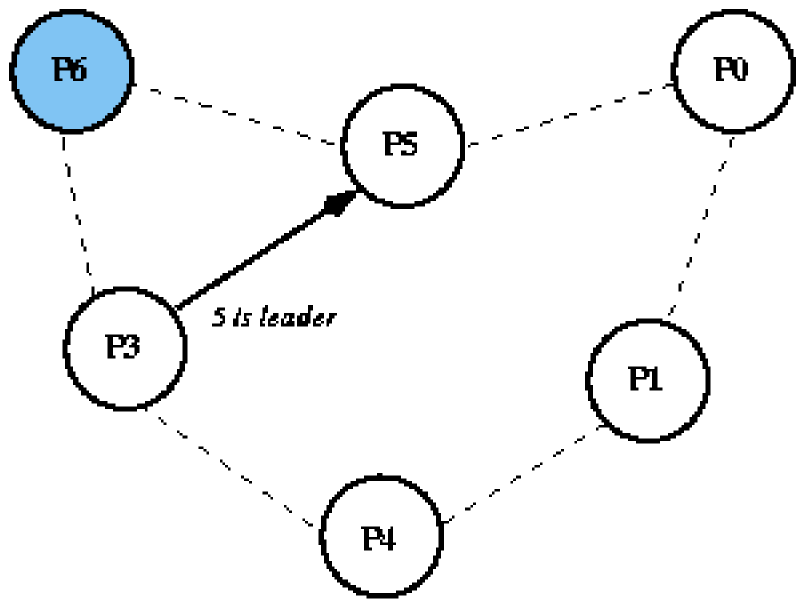
Token Ring Election Algorithm: Step 5

Process 0 passes the message on, adding its own id to the message.

Process 1 passes the message on, adding its own id to the message.



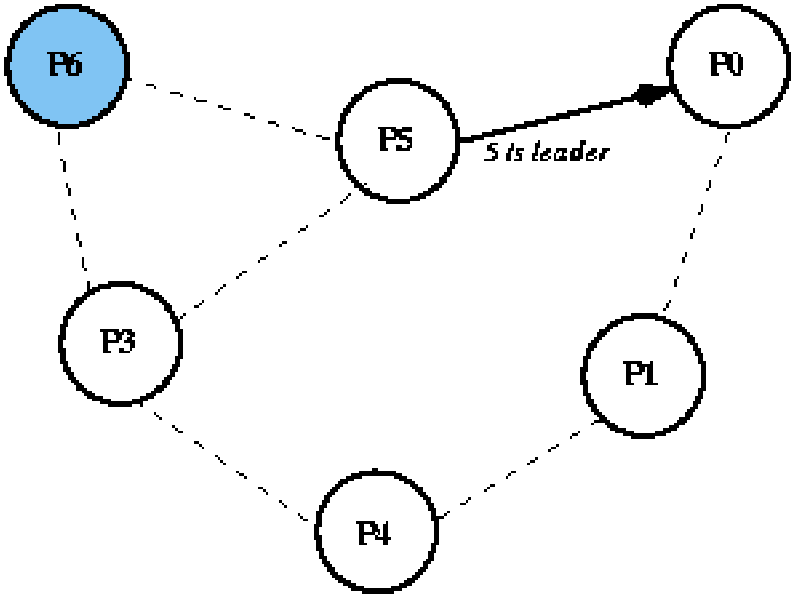
Token Ring Election Algorithm: Step 6



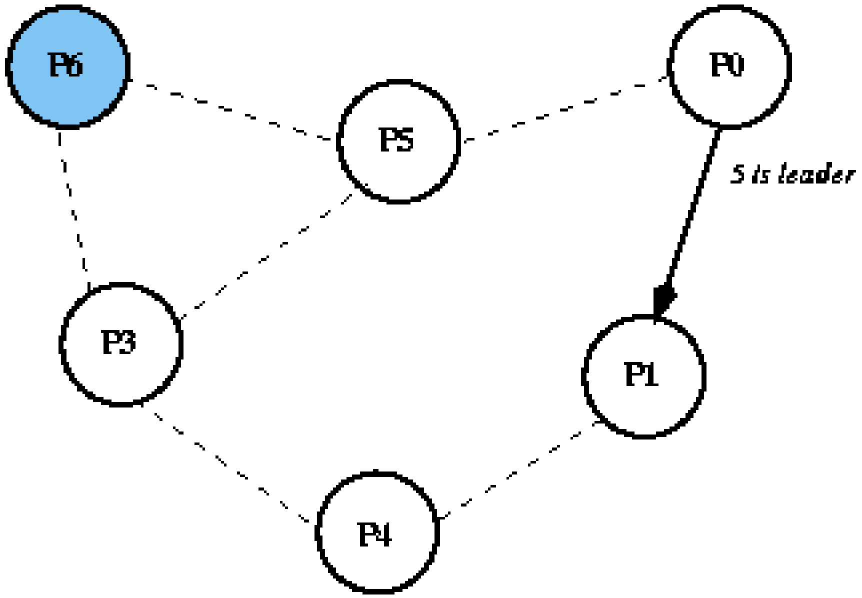
Token Ring Election Algorithm: Step 7

Process 4 passes the message on, adding its own id to the message.

When Process 3 receives the message back, it knows the message has gone around the ring, as its own id is in the list.Picking the highest id in the list, it starts the coordinator message "5 is the leader" around the ring.



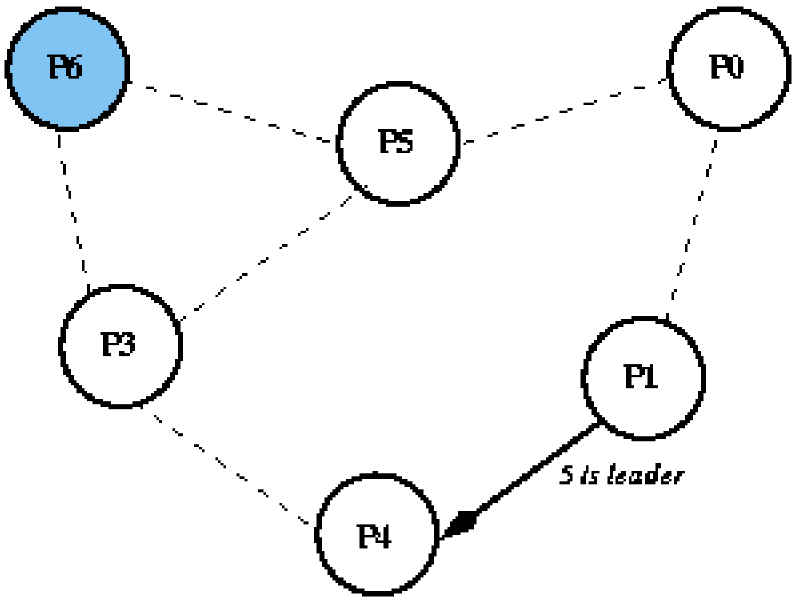
Token Ring Election Algorithm: Step 8



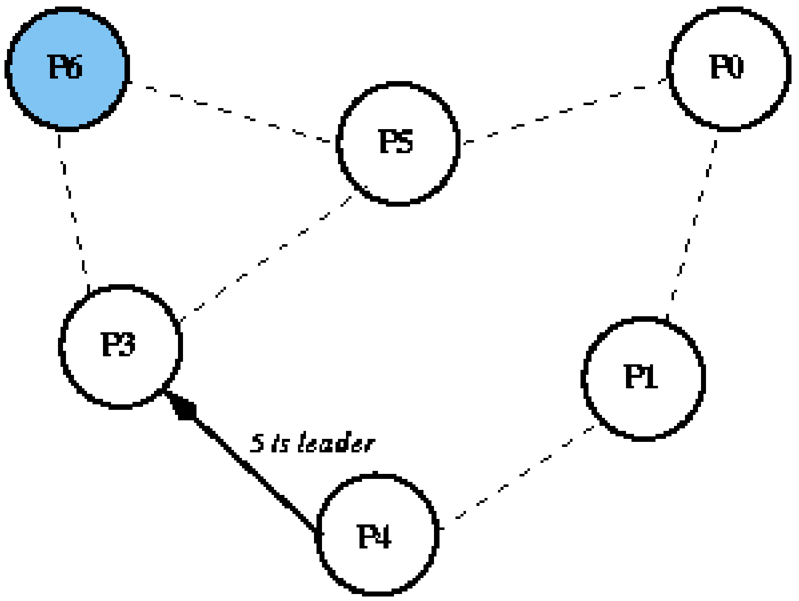
Token Ring Election Algorithm: Step 9

Process 5 passes on the coordinator message.

Process 0 passes on the coordinator message.



Token Ring Election Algorithm: Step 10



Token Ring Election Algorithm: Step 11

Process 1 passes on the coordinator message.

Process 3 receives the coordinator message, and stops it.