Coordination and Agreement

Outline

- Introduction
- Distributed Mutual Exclusion
- Election Algorithms
- Group Communication
- Consensus and Related Problems

Introduction

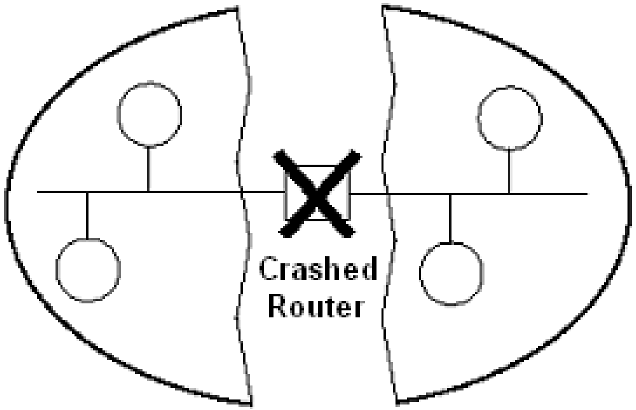
- Collection of algorithms that share an aim

  for a set of processes to coordinate their actions or to agree on one or more values.

- For example, Spaceship:
  - **Agreement:** it is essential that the computers controlling it agree on such conditions as whether the spaceship's mission is proceeding or has been aborted
  - **Coordination:** the computers must coordinate their actions correctly with respect to shared resources (the spaceship's sensors and actuators)

Main Assumptions

- Each pair of processes is connected by reliable channels
- Processes independent from each other
- Network: don't disconnect



- Processes fail only by crashing
- Local failure detector

Failure Detector

- Is a service that processes queries about whether a particular process has crashed.
- It is often implemented by a local object known as a *Local Failure Detector*.
- Failure detectors are not necessarily accurate.
- For example:
  - a process that timed-out after 255 seconds might have succeeded if allowed to proceed for 256 seconds.
- Two types of failure detector:
  - Unreliable failure detector
  - Reliable failure detector

Unreliable Failure Detector

- Produce one of two values when given the identity of a process: *Unsuspected* or *Suspected*.
  - **Unsuspected:** detector has recently received evidence suggesting that the process has not failed.
  - **Suspected:** failure detector has some indication that the process may have failed.
- Implement:
  - each process sends *alive* message to everyone else
  - not receiving *alive* message after timeout, report *Suspected*
  - if it subsequently receives, reports OK *(Unsuspected)*
- Most practical systems

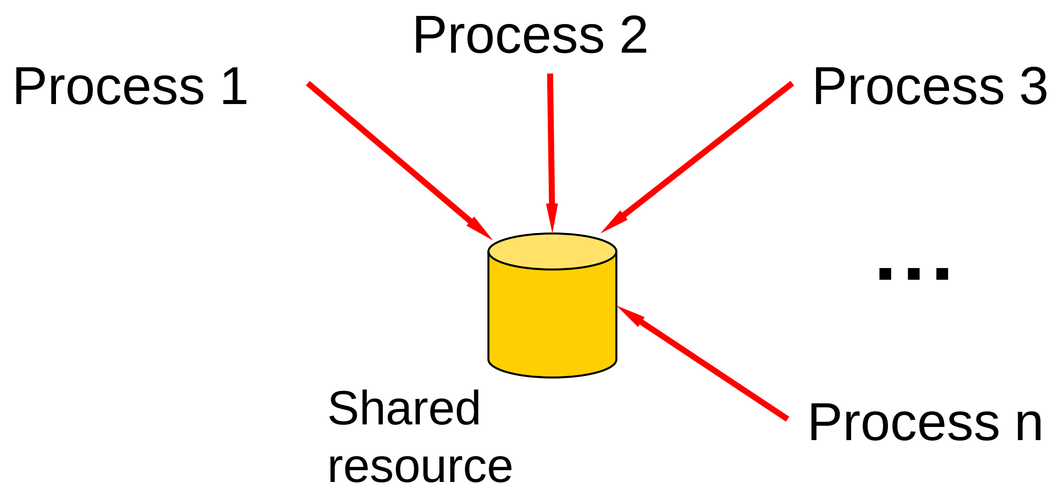
Reliable Failure Detector

- Is always accurate in detecting a process's failure.
- It answers processes' queries with either a response of *Unsuspected* or *Failed*.
  - **Unsuspected:** as before, can only be a hint that the process has not failed.
  - **Failed:** detector has determined that the process has crashed.
- Implement needs *synchronous* system
- Few practical systems

Outline

- Introduction
- **Distributed Mutual Exclusion**
- Election Algorithms
- Group Communication
- Consensus and Related Problems

Distributed Mutual Exclusion (1)



- Mutual exclusion very important
  - Prevent interference
  - Ensure consistency when accessing the resources

Distributed Mutual Exclusion (2)

- Mutual exclusion useful when the server managing the resources don't use locks
- Critical section

Enter()	*enter critical section – blocking*
• • •	*Access shared resources in critical section*
Exit()	*Leave critical section*

Distributed Mutual Exclusion (3)

- Distributed mutual exclusion: no shared variables, only message passing
- Properties:
  - **Safety:** At most one process may execute in the critical section at a time
  - **Liveness:** Requests to enter and exit the critical section eventually succeed

    → No deadlock and no starvation
  - **Ordering:** If one request to enter the CS happened-before another, then entry to the CS is granted in that order

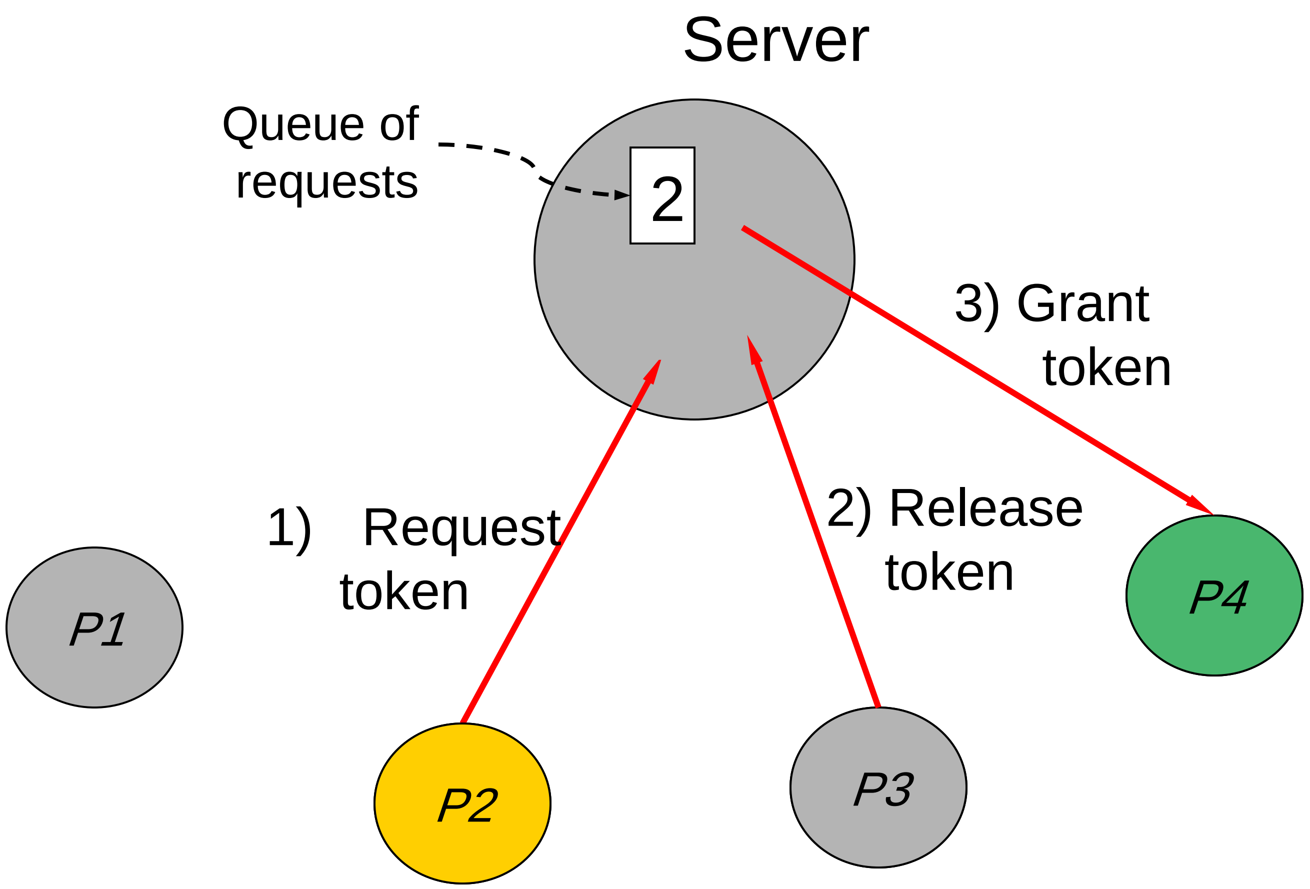
Mutual Exclusion Algorithms

- Basic Hypothesis:
  - System: asynchronous
  - Critical section: only one
  - Processes: don’t fail
  - Message transmission: reliable
- Central Server Algorithm
- Ring-Based Algorithm
- Mutual Exclusion using Multicast and Logical Clocks
- Maekawa’s Voting Algorithm
- Mutual Exclusion Algorithms Comparison

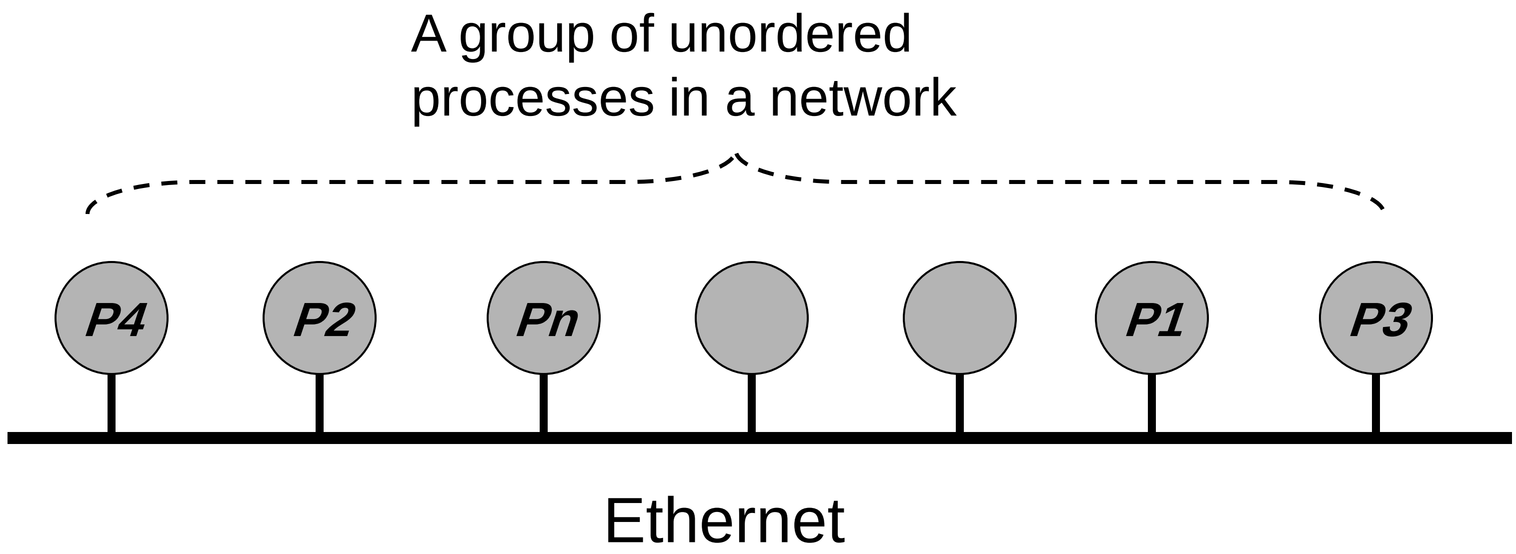
Evaluation of the performance alg.

- Bandwidth
  - The number of message sent in each entry and exit operation
- Client Delay
- Throughput

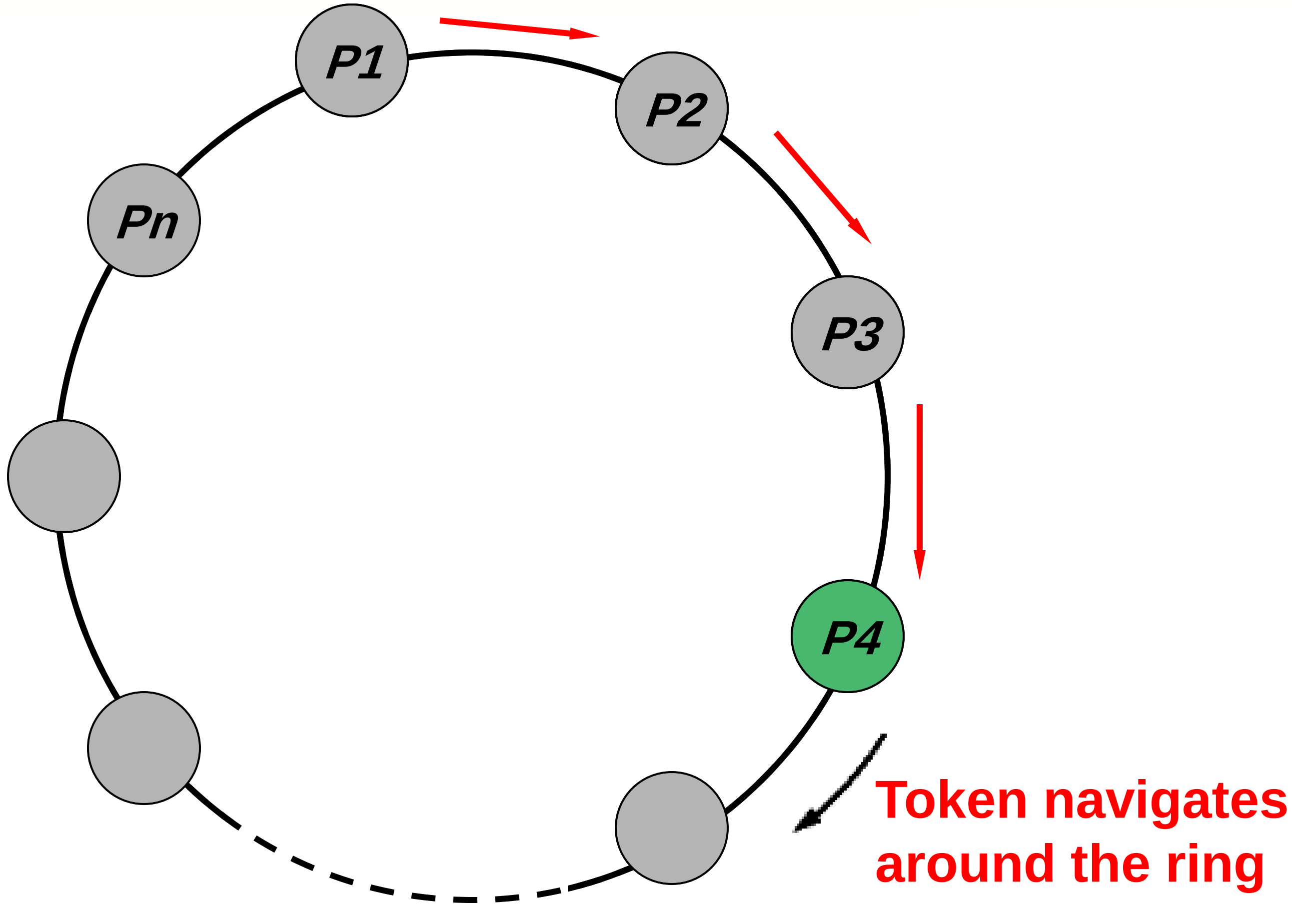
Central Server Algorithm



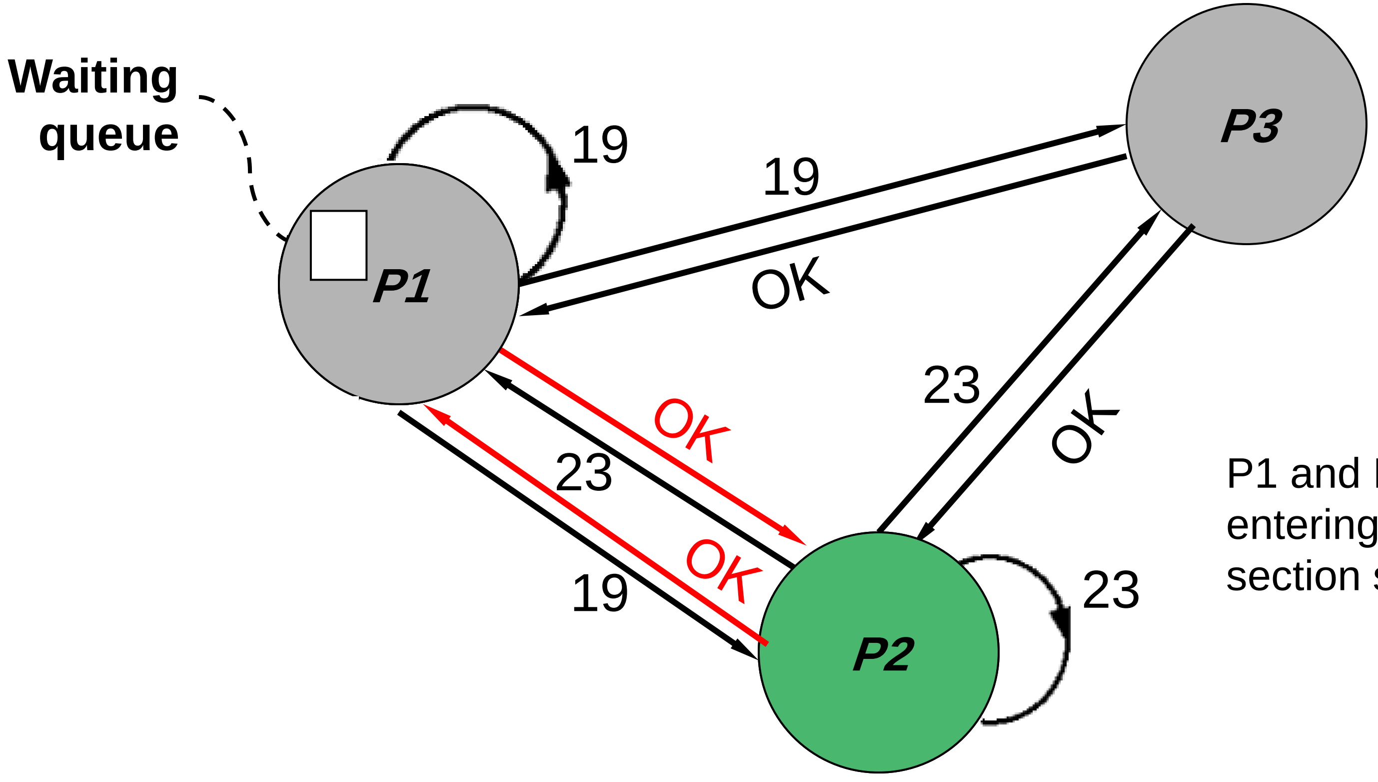
Ring-Based Algorithm (1)



Ring-Based Algorithm (2)



Mutual Exclusion using Multicast and Logical Clocks (1)



P1 and P2 request entering the critical section simultaneously

Mutual Exclusion using Multicast and Logical Clocks (2)

- Main steps of the algorithm:

Initialization

State := RELEASED;

Process p_i request entering the critical section

State := WANTED;

T := request's timestamp;

Multicast request <T, p_i> to all processes;

Wait until (Number of replies received = (N – 1));

State := HELD;

Mutual Exclusion using Multicast and Logical Clocks (3)

- Main steps of the algorithm (cont'd):

On receipt of a request <T_i, p_i> at p_j ($i \neq j$)

If (state = HELD) OR
(state = WANTED AND $(T, p_j) < (T_i, p_i)$)
Then queue request from p_i without replying;
Else reply immediately to p_i;

To quit the critical section

state := RELEASED;
Reply to any queued requests;

Maekawa's Voting Algorithm (1)

- Candidate process: must collect sufficient votes to enter to the critical section
- Each process p_i maintain a *voting set* V_i (i=1, …, N), where $V_i \subseteq \{p_1, \ldots, p_N\}$
- Sets V_i: chosen such that $\forall$ i,j
  - $p_i \in V_i$
  - $V_i \cap V_j \neq \varnothing$ **(at least one common member of any two voting sets)**
  - $|V_i| = k$ **(fairness)**
  - Each process p_j is contained in M of the voting sets V_i

Maekawa's Voting Algorithm (2)

- Main steps of the algorithm:

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;

p_i enter the critical section only after collecting K votes

Maekawa's Voting Algorithm (3)

- Main steps of the algorithm (cont'd):

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 Reply immediately to p_i;

voted := TRUE;

For p_i to exit the critical section

state := RELEASED;

Multicast release to all processes V_i ;

Maekawa's Voting Algorithm (4)

- Main steps of the algorithm (cont'd):

On a receipt of a release from p_i at p_j

If (queue of requests is non-empty)

Then remove head of queue, e.g., p_k;

send reply to p_k;

voted := TRUE;

Else voted := FALSE;

M. E. Algorithms Comparison

Algorithm	Number of messages		Problems
	Enter()/Exit	Before Enter()	
Centralized	3	2	Crash of server
Virtual ring	1 to N	0 to N-1	Crash of a process Token lost Ordering non-satisfied
Logical clocks	3(N-1)	2(N-1)	Crash of a process
Maekawa's Alg.	$3\sqrt{N}$	$2\sqrt{N}$	Crash of a process who **votes**

Outline

- Introduction
- Distributed Mutual Exclusion
- **Election Algorithms**
- Group Communication
- Consensus and Related Problems

Election Algorithms (1)

- **Objective:** Elect one process p_i from a group of processes $p_1 \ldots p_N$

Even if multiple elections have been started simultaneously

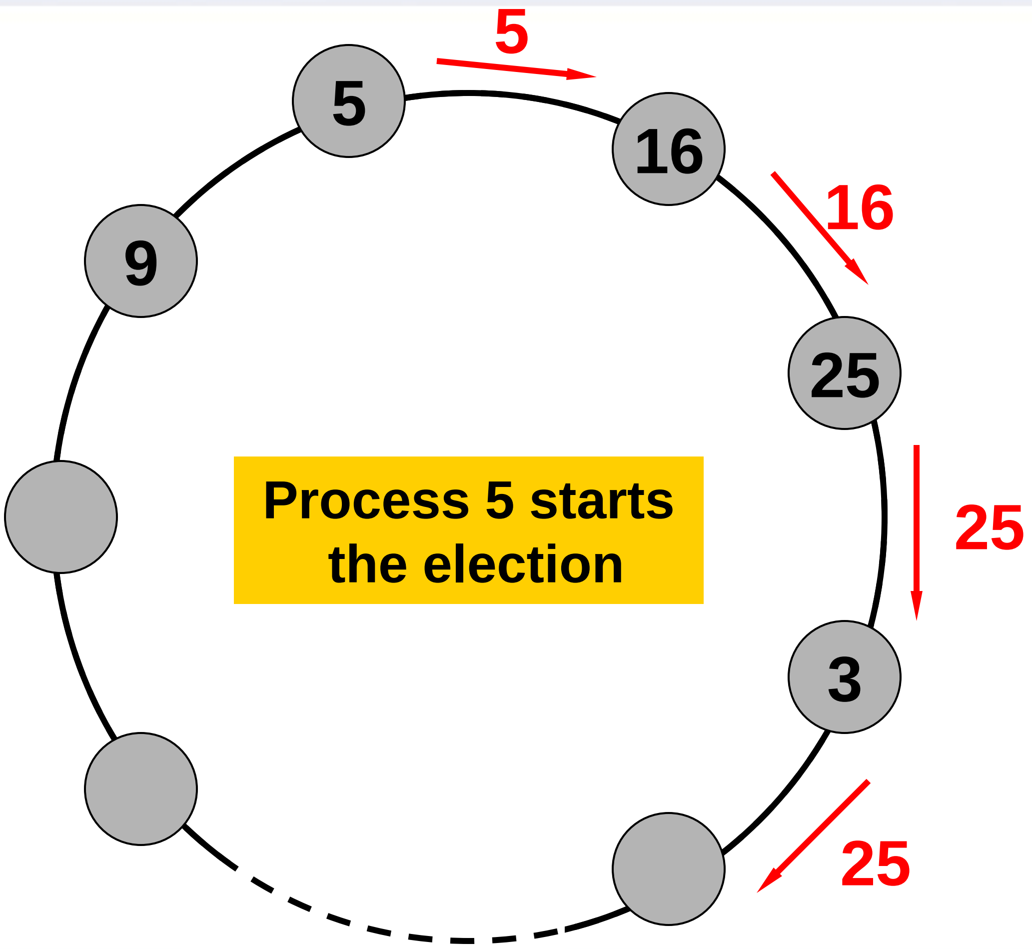
- At any point in time, a process p_i is either a participant or a non-participant
- Each process p_i maintains the identity of the elected in the variable *$Elected_i$* (NIL '⊥' if it isn't defined yet)
- **Properties to satisfy:** $\forall\ p_i$
  - Safety: *$Elected_i$* = NIL or *Elected* = P
  - Liveness: p_i participates and sets *$Elected_i$* ≠ NIL, or crashes

A non-crashed process with the largest identifier

Election Algorithms (2)

- Ring-Based Election Algorithm
- Bully Algorithm
- Election Algorithms Comparison

Ring-Based Election Algorithm (1)



Ring-Based Election Algorithm (2)

Initialization

Participant$_i$:= FALSE;
Elected$_i$:= NIL

P_i starts an election

Participant$_i$:= TRUE;

Send the message <*election, p_i*> to its neighbor

Receipt of a message <*elected, p_j*> *at p_i*

If $p_i \neq p_j$

Then **Participant$_i$:= FALSE;**
Elected$_i$:= p_j;
Send the message <*elected, p_j*> to its neighbor

Ring-Based Election Algorithm (3)

Receipt of the election's message <*election*, p_i> at p_j

If $p_i > p_j$

Then Send the message <*election*, p_i> to its neighbor
Participant$_j$:= TRUE;

Else If $p_i < p_j$ AND Participant$_j$ = FALSE

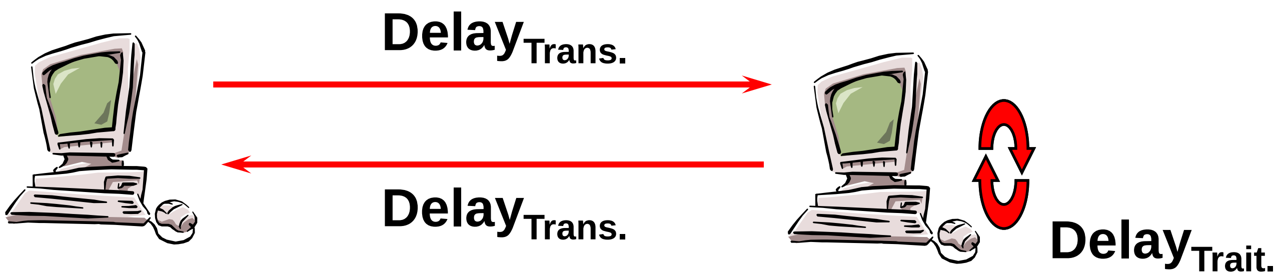
Then Send the message <*election*, p_j> to its neighbor
Participant$_j$:= TRUE;

Else If $p_i = p_j$

Then Elected$_j$:= p_j;
Participant$_j$:= FALSE;
Send the message <*elected*, p_j> to its neighbor

Bully Algorithm (1)

- **Characteristic:** Allows processes to crash during an election
- **Hypothesis:**
  - Reliable transmission
  - Synchronous system

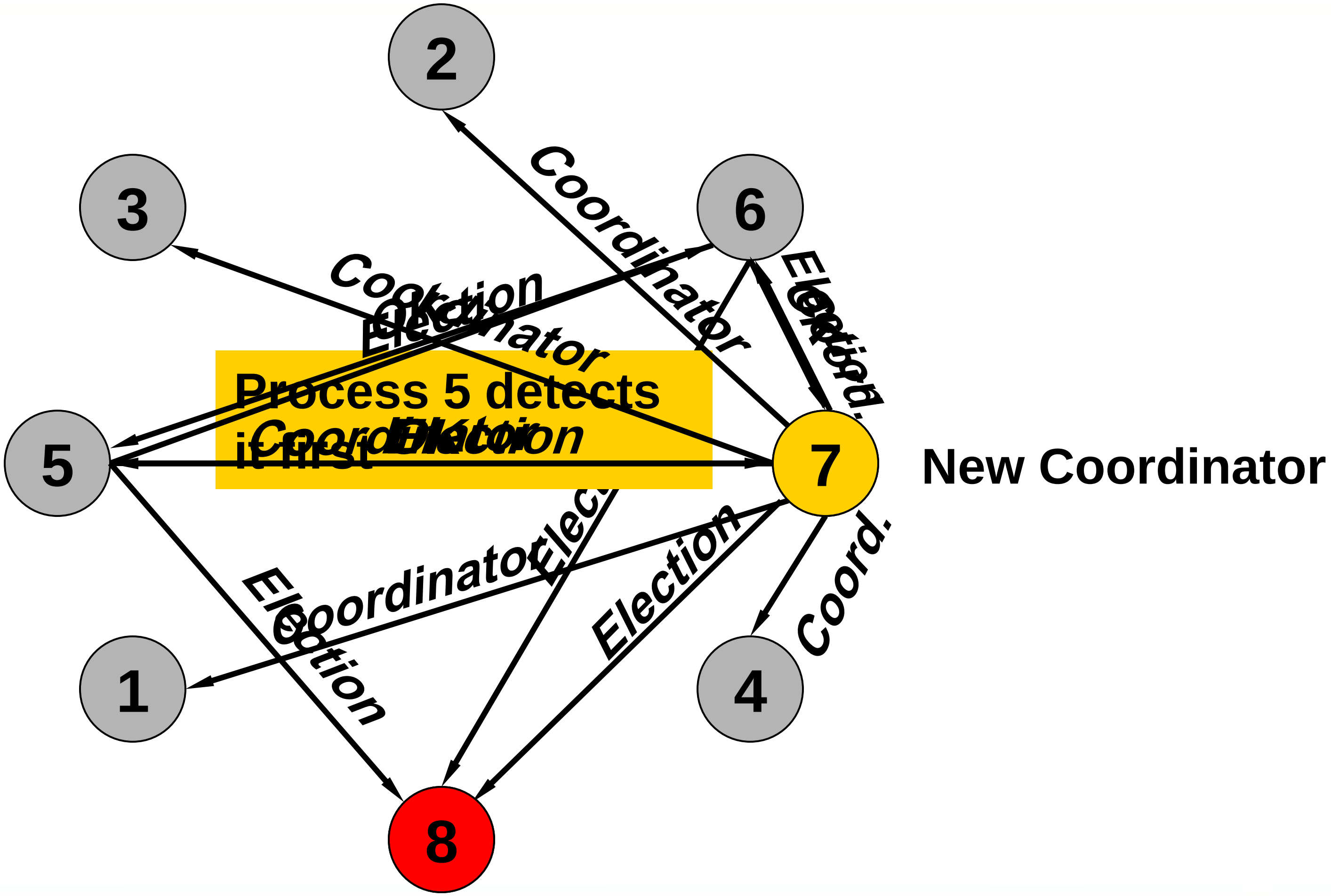


$$T = 2\ Delay_{Trans.} + Delay_{Trait.}$$

Bully Algorithm (2)

- **Hypothesis (cont'd):**
  - Each process knows which processes have higher identifiers, and it can communicate with all such processes
- **Three types of messages:**
  - *Election*: starts an election
  - *OK*: sent in response to an election message
  - *Coordinator*: announces the new coordinator
- Election started by a process when it notices, through timeouts, that the coordinator has failed

Bully Algorithm (3)



Bully Algorithm (4)

Initialization

Elected$_i$:= NIL

p_i starts the election

Send the message (*Election, p_i*) to p_j , i.e., $p_j > p_i$

Waits until message (*OK, p_j*) from p_j are received;

<u>If</u> no message (*OK, p_j*) arrives during T

<u>Then</u> Elected$_i$:= p_i;
Send the message (*Coordinator, p_i*) to p_j , i.e., $p_j < p_i$

<u>Else</u> waits until receipt of the message (*coordinator)*
(if it doesn't arrive during another timeout T', it begins another election)

Bully Algorithm (5)

Receipt of the message (*Coordinator, p_j*) at p_i

$Elected_i := p_j$;

Receipt of the message (*Election, p_j*) at p_i

Send the message *(OK, p_i)* to p_j

Start the election unless it has begun one already

- When a process is started to replace a crashed process: it begins an election

Election Algorithms Comparison

Election algorithm	Number of messages	Problems
Virtual ring	2N to 3N-1	Don't tolerate faults
Bully	N-2 to $O(N^2)$	System must be synchronous