UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

Sistemas Distribuídos

Prof. Sérgio T. Carvalho
sergio@inf.ufg.br

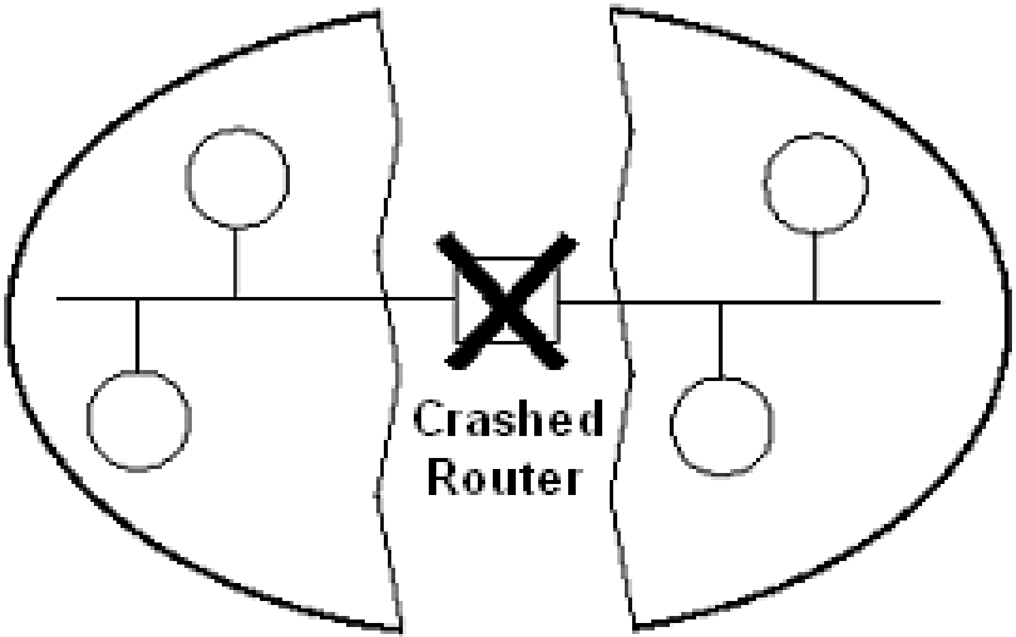
Coordenação

Based on Minh, Thái, Vũ

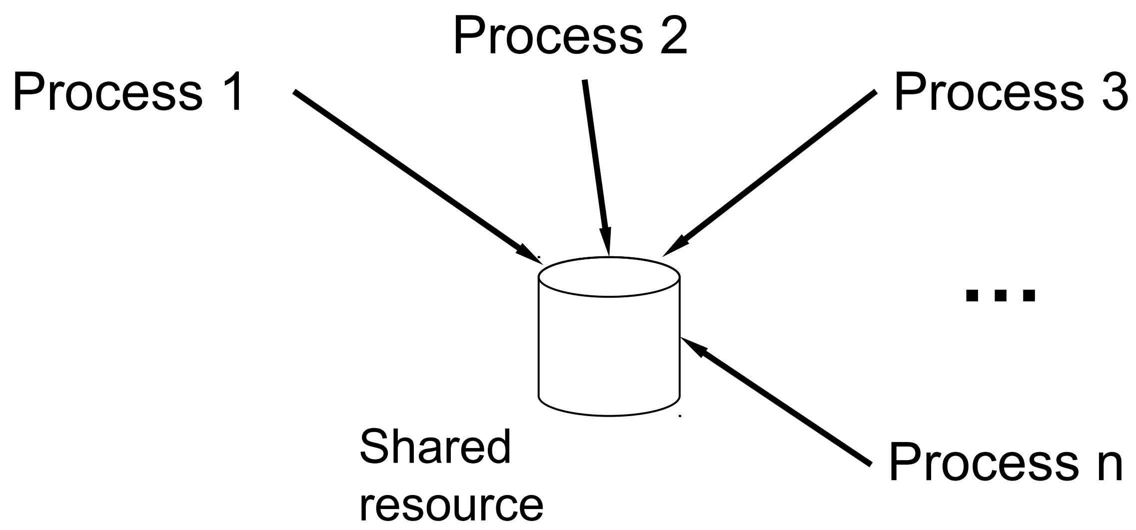
Main Assumptions

- 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

Main Assumptions



Distributed Mutual Exclusion



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

Distributed Mutual Exclusion

- 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*

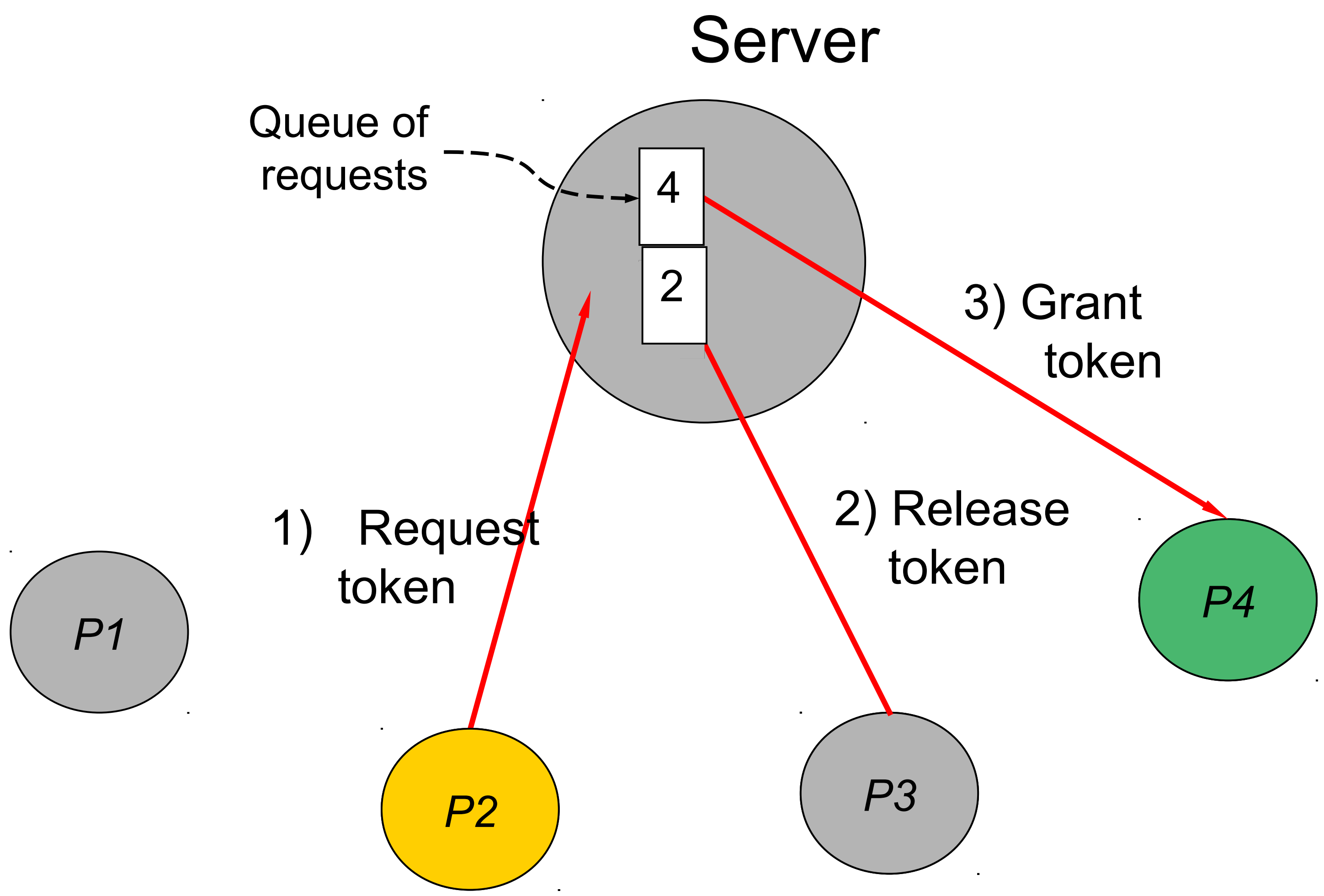
Distributed Mutual Exclusion

- 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

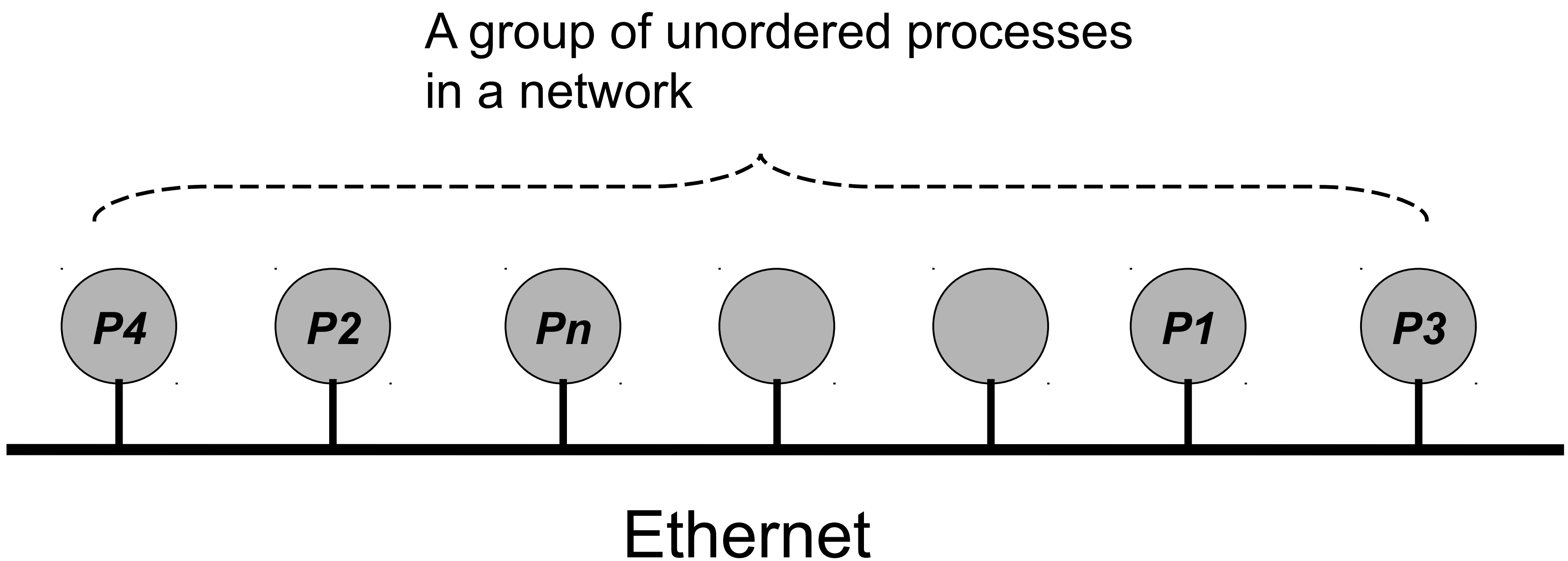
      No deadlock and no starvation

    - **Ordering:** If one request to enter the CS happened-before another, then entry to the CS is granted in that order

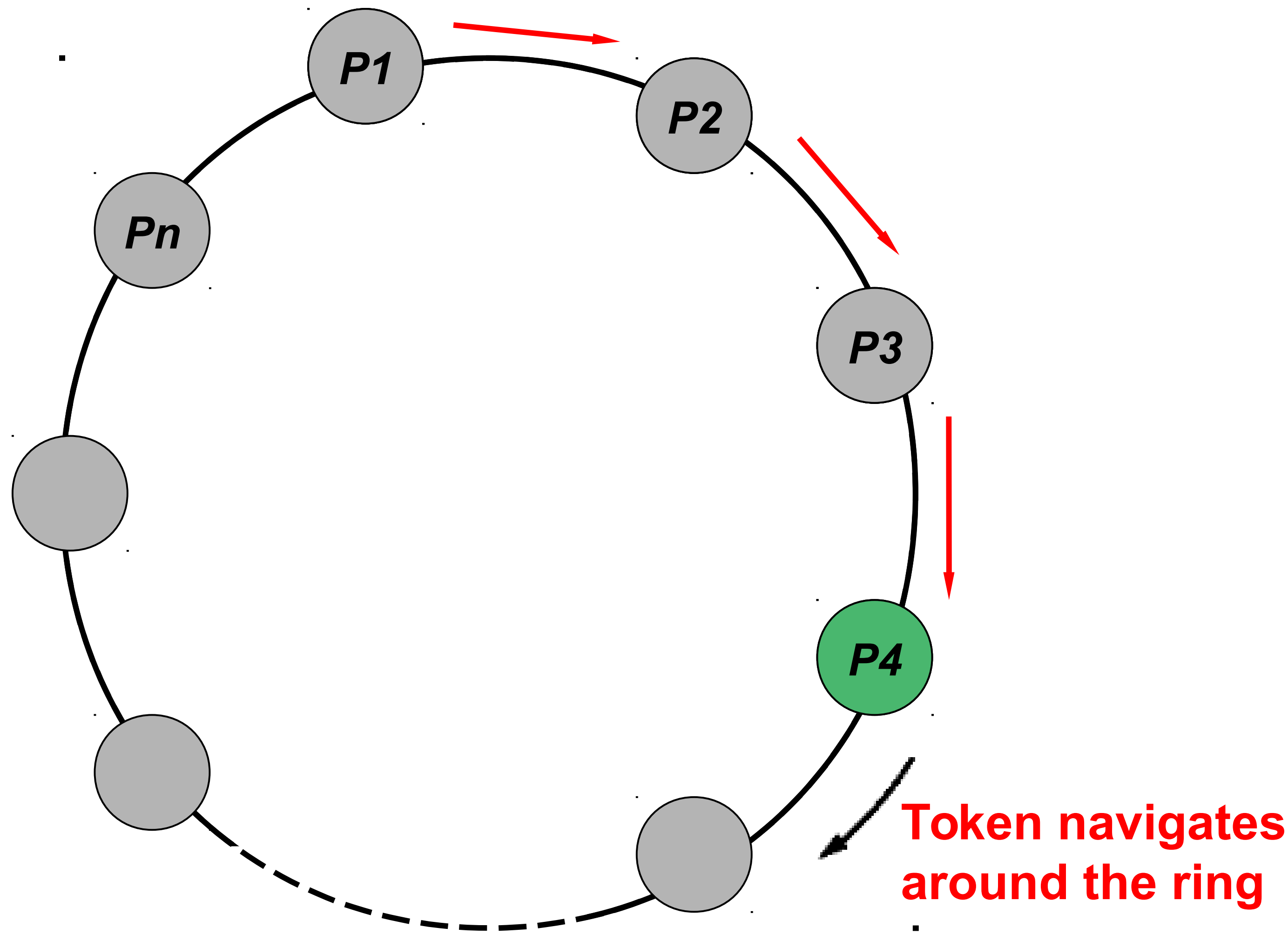
Central Server Algorithm



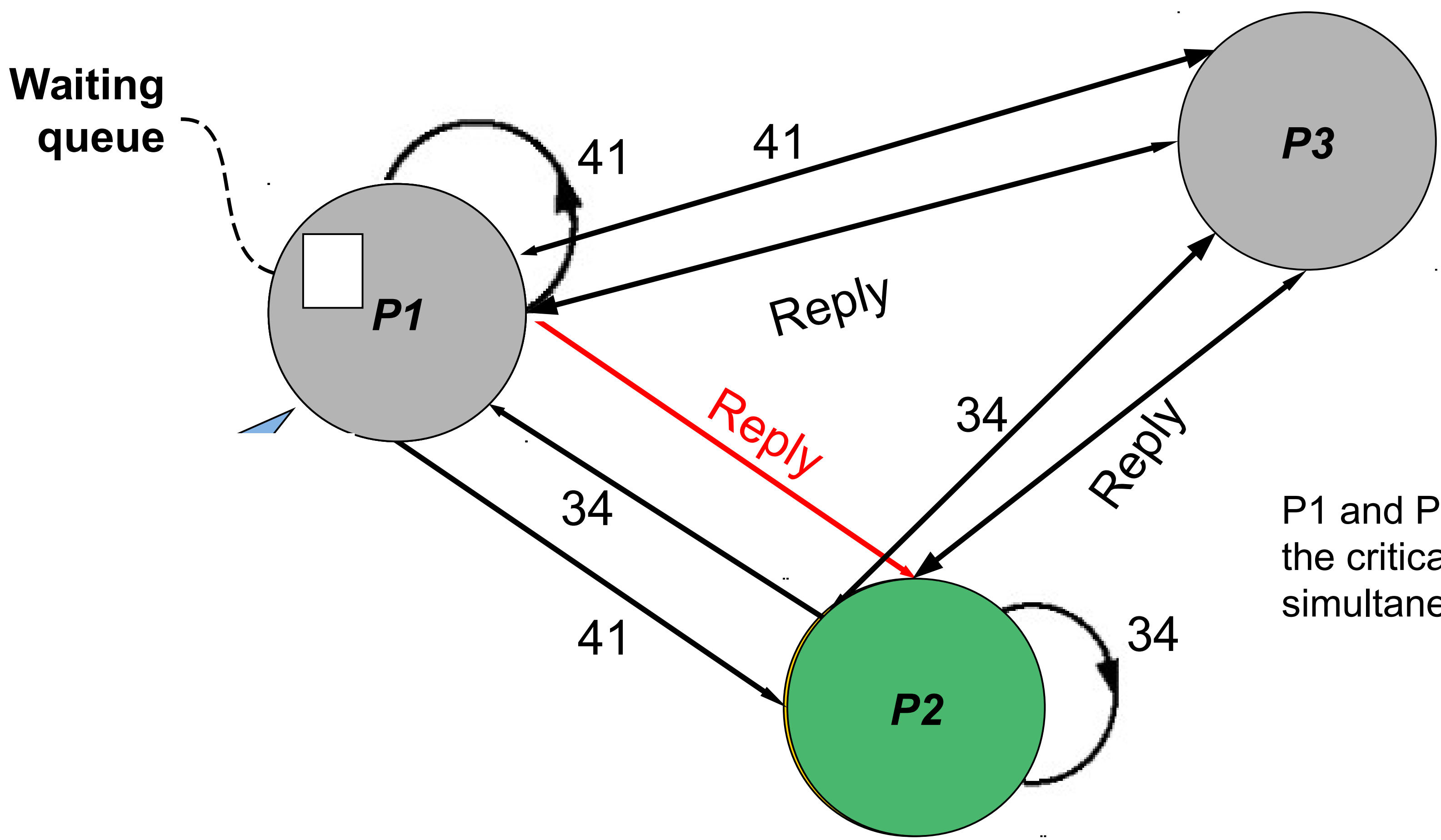
Ring Based Algorithm



Ring Based Algorithm



Mutual Exclusion using Multicast and Logical Clocks



P1 and P2 request entering the critical section simultaneously

Mutual Exclusion using Multicast and Logical Clocks

- 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

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

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

<u>If</u> (state = HELD) OR

(state = WANTED AND $(T, p_j) < (T_i, p_i)$)

<u>Then</u> queue request from p_i without replying;

<u>Else</u> reply immediately to p_i;

To quit the critical section

state := RELEASED;

Reply to any queued requests;

Maekawa's Voting Algorithm

- 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
  - $p_i \in V_i$
  - $V_i \cap V_j \neq \emptyset$ **(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

- 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 – {p_i};

Wait until (number of replies received = K – 1);

state := HELD;

p_i enter the critical section only after collecting K-1 votes

Maekawa's Voting Algorithm

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

On receipt of a request from p_i at p_j ($i \neq j$)

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 – $\{p_i\}$;

Maekawa's Voting Algorithm

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

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

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;

Algorithms Comparison

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

Election Algorithms

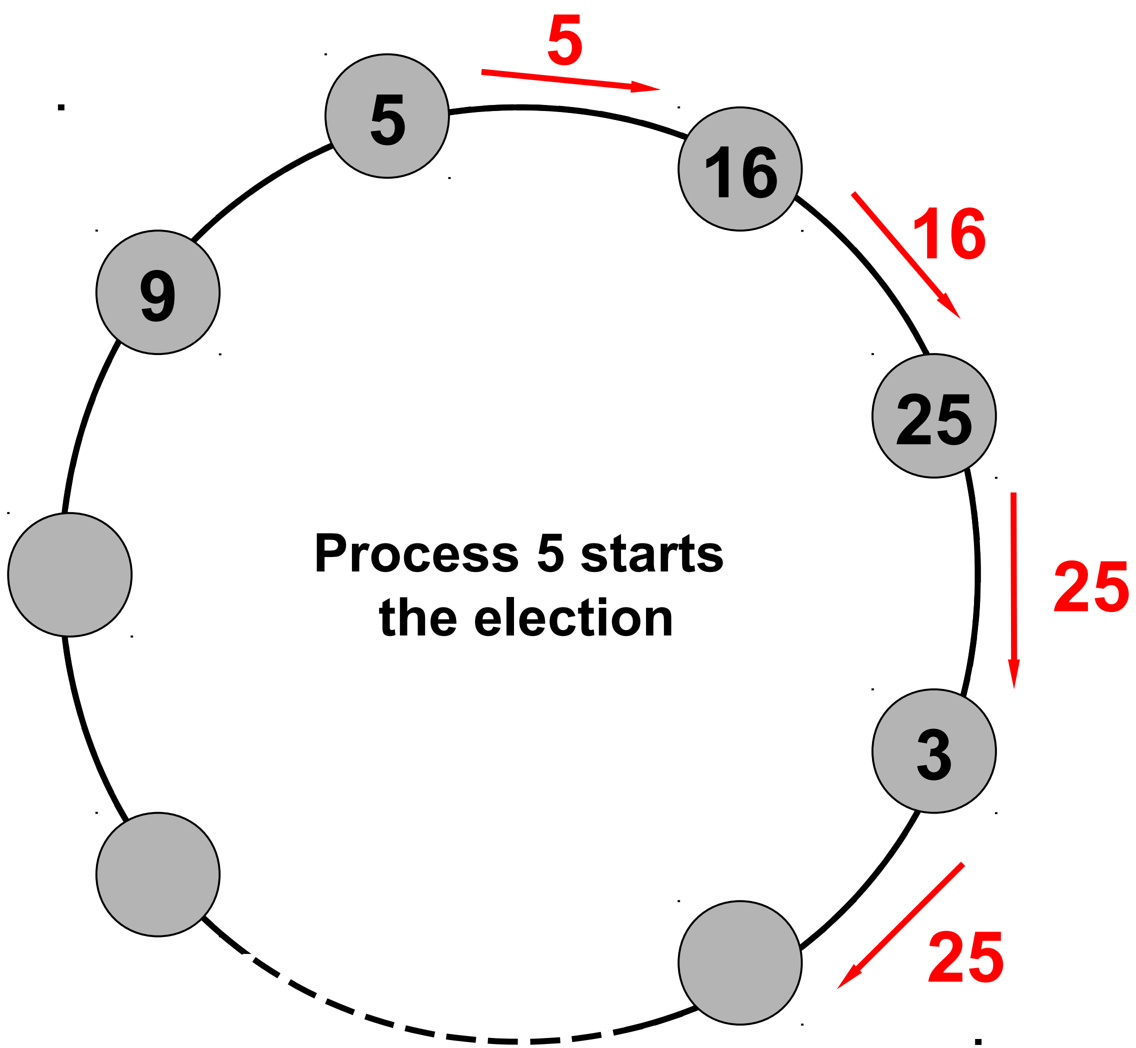
- **Objective:** Elect one process p_i from a group of processes $p_1 \ldots p_N$

Even if multiple elections have been started simultaneously

- **Utility:** Elect a primary manager, a master process, a coordinator or a central server
- 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

    A non-crashed process with the largest identifier
  - Liveness: p_i participates and sets $Elected_i \neq$ NIL, or crashes

Ring-based Election Algorithm



Ring-based Election Algorithm

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*

$Participant_i$:= FALSE;

If $p_i \neq p_j$

Then Send the message <*elected, p_j*> to its neighbor

Ring-based Election Algorithm

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

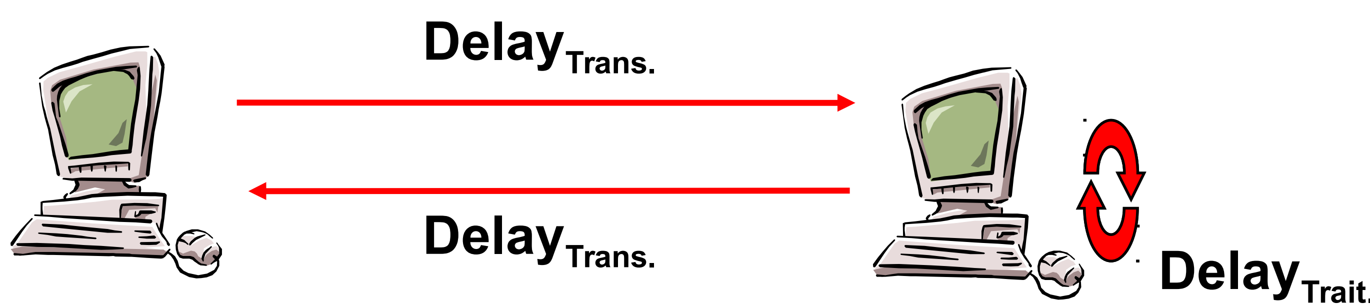
Then Send the message <*election*, p_j> to its neighbor
Participant$_j$:= TRUE;

Else If $p_i = p_j$

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

Bully Algorithm

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

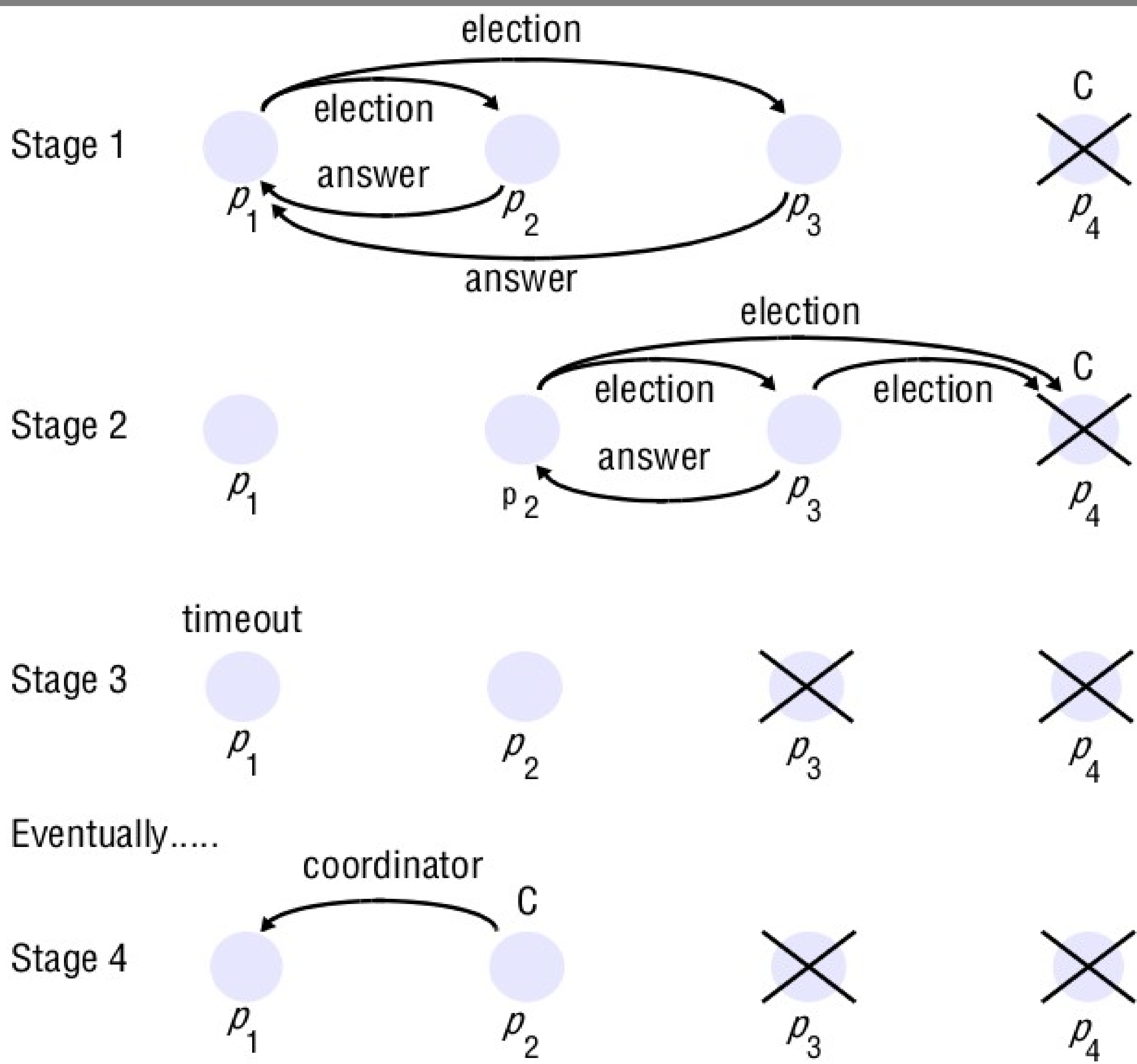


$$T = 2\ Delay_{Trans.} + Delay_{Trait.}$$

Bully Algorithm

- **Hypotheses (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



Bully Algorithm

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 all messages (*OK, p_j*) from p_j are received;

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

<u>Then</u> Elected := 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

Receipt of the message (*Coordinator, p_j*)

Elected := 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

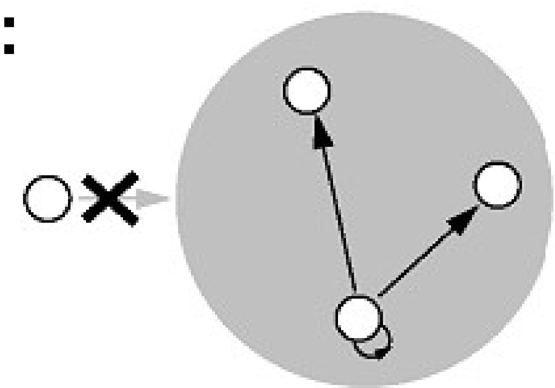
Election algorithm	Number of messages	Problems
Virtual ring	2N to 3N-1	Don’t tolerate faults
Bully	N-2 to $O(N^2)$	System must be synchronous

Group Communication

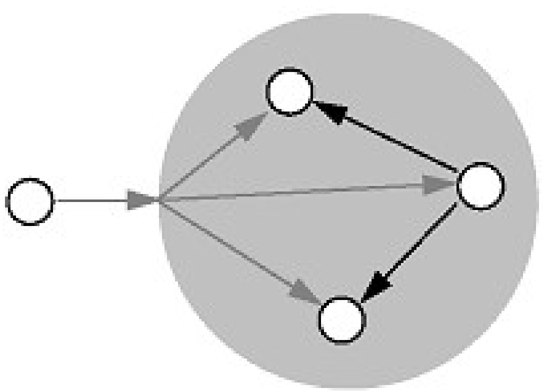
- **Objective:** each of a group of processes must receive copies of the messages sent to the group
- **Group communication requires:**
  - Coordination
  - Agreement: on the set of messages that is received and on the delivery ordering
- We study multicast communication of processes whose membership is known (static groups)

Group Communication

- **System:** contains a collection of processes, which can communicate **reliably** over **one-to-one** channels
- **Processes:** members of groups, may fail only by crashing
- **Groups:**



Closed group



Open group

Group Communication

- **Primitives:**
  - ***multicast(g, m)***: sends the message *m* to all members of group g
  - ***deliver(m)*** **:** delivers the message *m* to the calling process
  - ***sender(m)*** **:** unique identifier of the process that sent the message *m*
  - ***group(m)***: unique identifier of the group to which the message *m* was sent

Basic Multicast

- **Objective:** Guarantee that a correct process will eventually deliver the message as long as the multicaster does not crash
- **Primitives:** B_multicast, B_deliver
- **Implementation:** Use a reliable one-to-one communication

To B_multicast(g, m)

For each process p $\in$ g, send(p, m);

On receive(m) at p

threads to perform the send operations simultaneously

B_deliver(m) to p

- **Unreliable:** Acknowledgments may be dropped

Reliable Multicast

- **Properties to satisfy:**
  - **Integrity:** A correct process P delivers the message m at most once
  - **Validity:** If a correct process multicasts a message m, then it will eventually deliver m
  - **Agreement:** If a correct process delivers the message m, then all other correct processes in group(m) will eventually deliver m
- **Primitives:** R_multicast, R_deliver

Reliable Multicast

- **Implementation using B-multicast:**

Initialization

msgReceived := {};

R-multicast(g, m) by p

B-multicast(g, m); // $p \in g$

B-deliver(m) by q with g = group(m)

If (m $\notin$ msgReceived)

Then msgReceived := msgReceived $\cup$ {m};
If ($q \neq p$) Then B-multicast(g, m);
R-deliver(m);

Correct algorithm, but inefficient

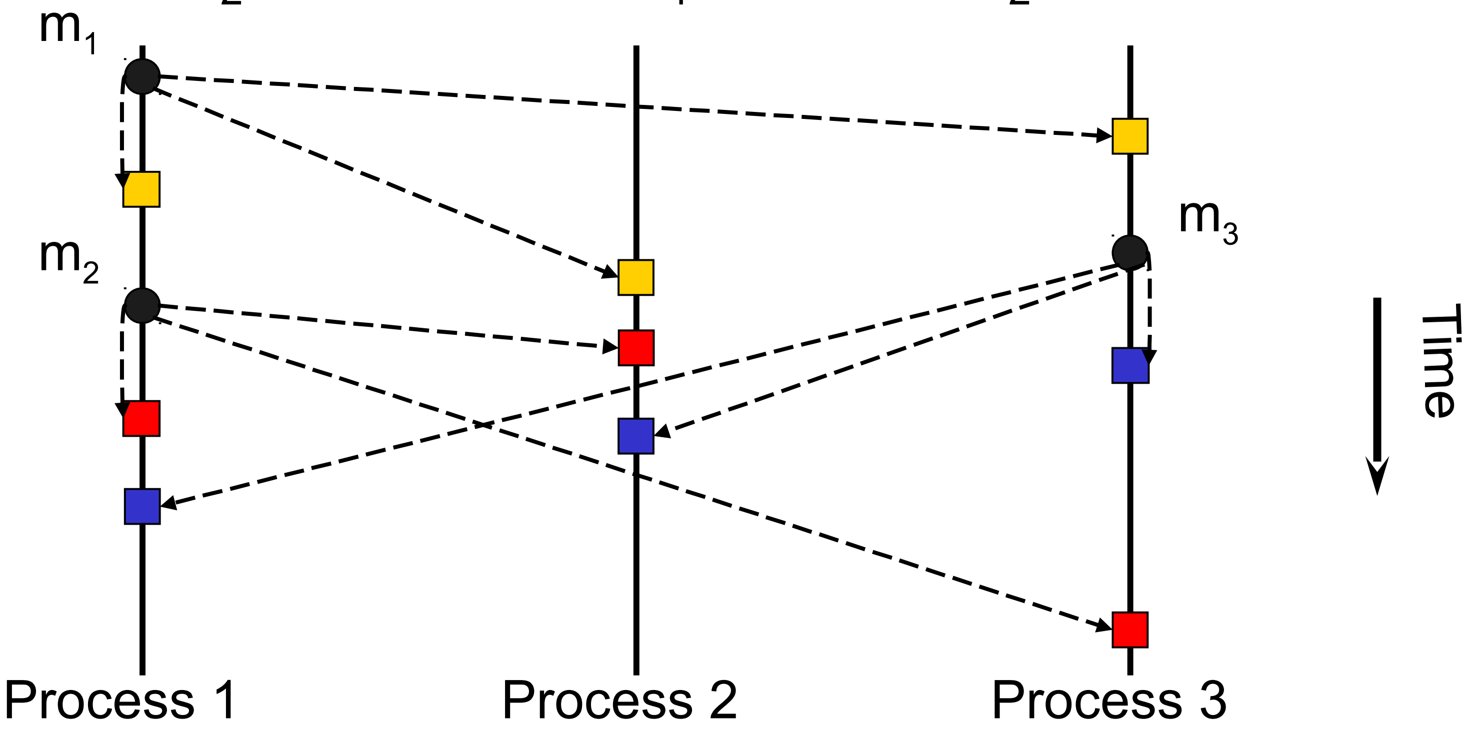
(each message is sent |g| times to each process)

Ordered Multicast

- **Ordering categories:**
  - FIFO Ordering
  - Total Ordering
  - Causal Ordering
  - Hybrid Ordering: Total-Causal,
    Total-FIFO

FIFO Ordering

- If a correct process issues multicast(g, m_1) and then multicast(g, m_2), then every correct process that delivers m_2 will deliver m_1 before m_2

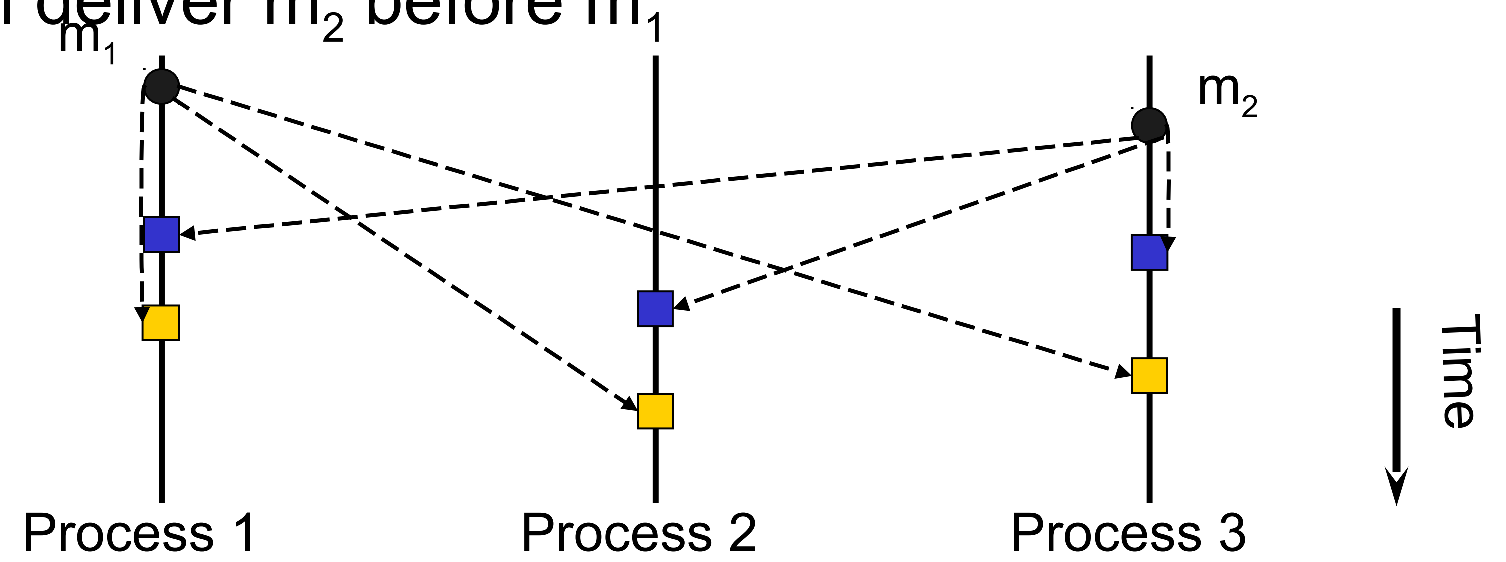


FIFO Ordering

- **Primitives:** FO_multicast, FO_deliver
- **Implementation:** Use of sequence numbers
- Variables maintained by each process p:
  - S_g^p: Number of messages sent by p to group g
  - R_g^q: sequence number of the latest message p has delivered from process q that was sent to the group
- Algorithm
- FIFO Ordering is reached only under the assumption that groups are non-overlapping

Total Ordering

- If a correct process delivers message m_2 before it delivers m_1, then any correct process that delivers m_1 will deliver m_2 before m_1



- **Primitives:** TO_multicast, TO_deliver

Total Ordering

- **Implementation:** Assign totally ordered identifiers to multicast messages
- Each process makes the same ordering decision based upon these identifiers
- **Methods for assigning identifiers to messages:**
  - Sequencer process
  - Processes collectively agree on the assignment of sequence numbers to messages in a distributed fashion

Total Ordering

- **Sequencer process:** Maintains a group-specific sequence number S_g

Initialization

$S_g := 0;$

B-deliver(<m, Ident.>) with g = group(m)

B-multicast(g, <"order", Ident., S_g>);

$S_g = S_g + 1;$

- Algorithm for group member $p \in g$

Initialization

$R_g := 0;$

Total Ordering

Unique identifier of m

TO-multicast(g, m) by p

B-multicast(g ∪ Sequencer(g), <m, Ident.>);

B-deliver(<m, Ident.>) by p, with g = group(m)

Place <m, Ident.> in hold-back queue;

B-deliver(m_{order}= <“order”, Ident., S>) by p, with g = group(m_{order})

Wait until (<m, Ident.> in hold-back queue AND S = R_g);

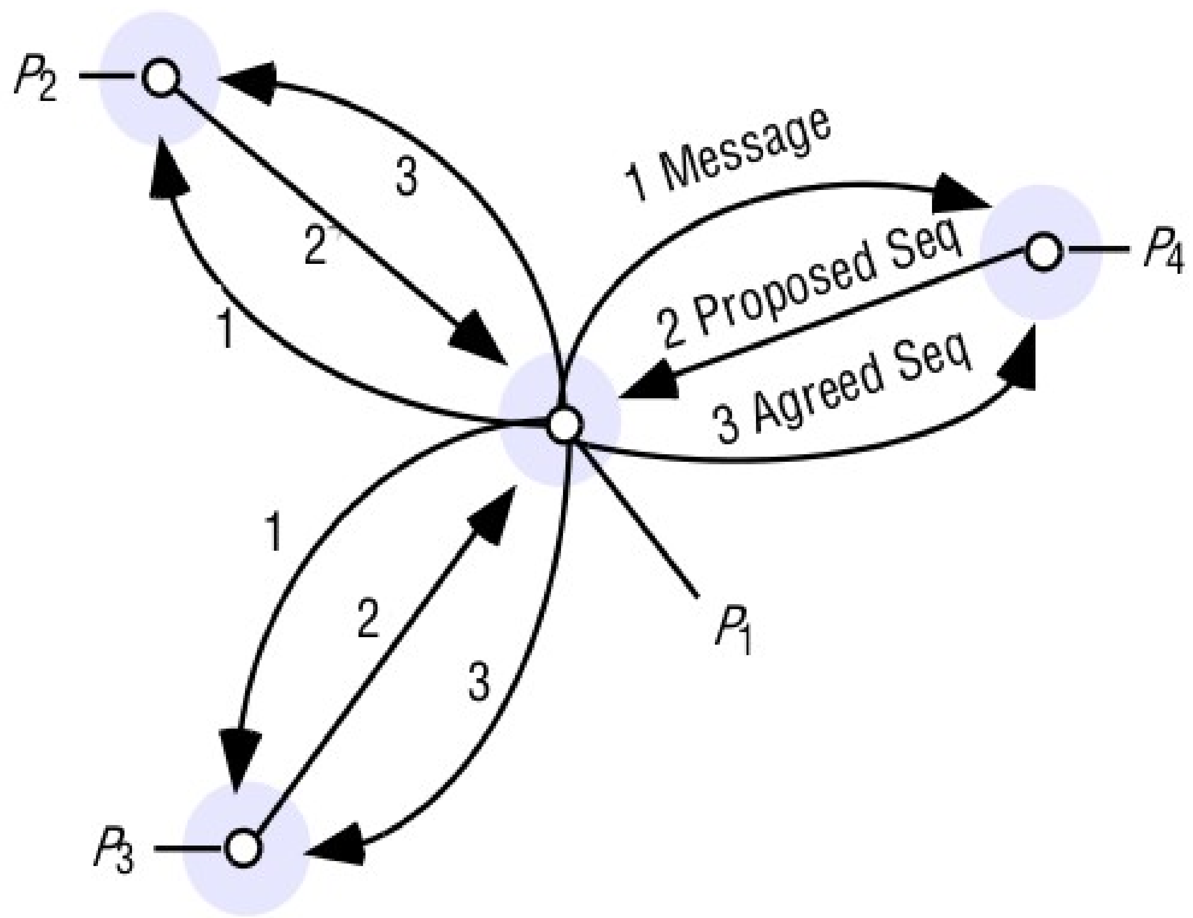
TO-deliver(m);

R_g = S + 1;

Total Ordering

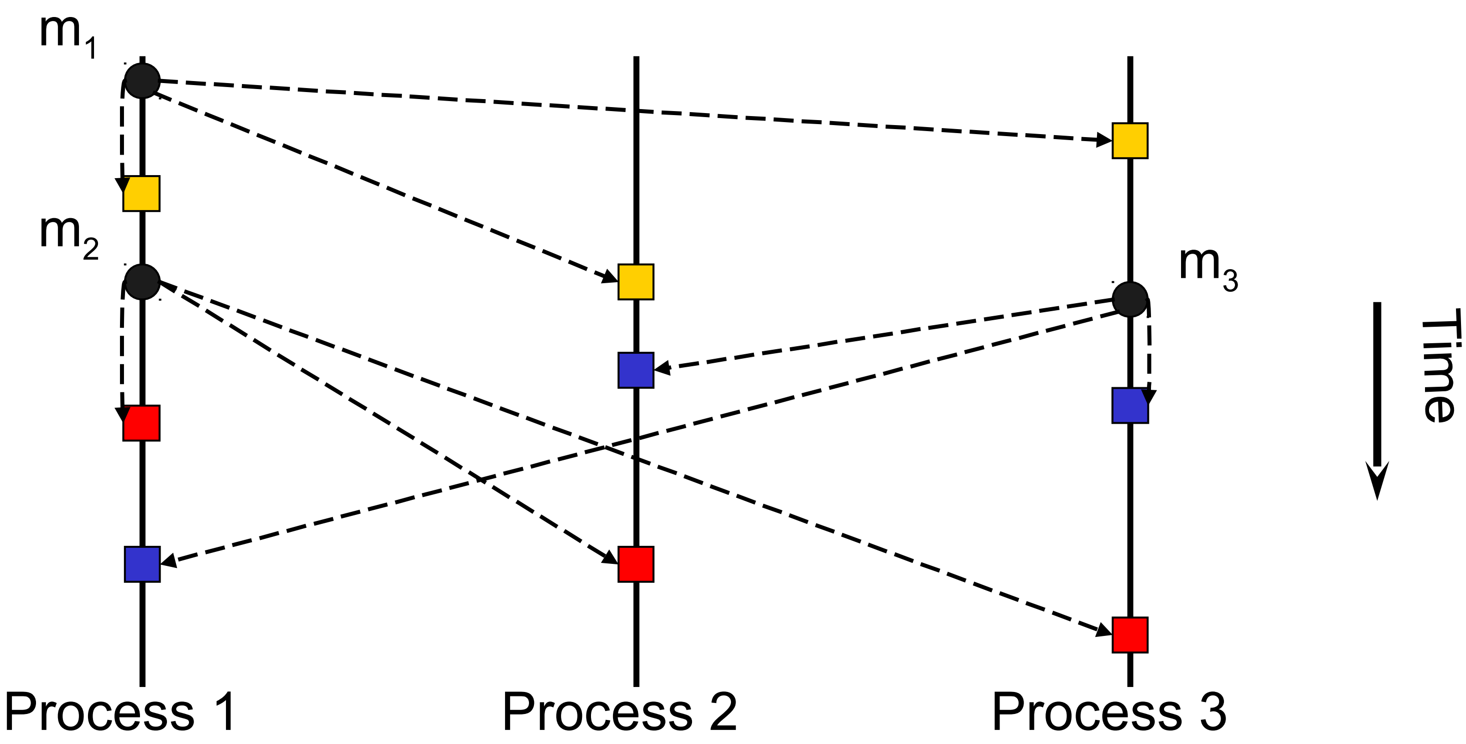
- Processes collectively agree on the assignment of sequence numbers to messages in a distributed fashion
- Variables maintained by each process p:
  - P_g^q: largest sequence number proposed by q to group g
  - A_g^q: largest agreed sequence number q has observed so far for group g

Total Ordering



Causal Ordering

- If multicast(g, m_1) → multicast(g, m_3), then any correct process that delivers m_3 will deliver m_1 before m_3



Causal Ordering

- **Primitives:** CO_multicast, CO_deliver
- Each process p_i of group g maintains a timestamp vector

  V_i^g

  $V_i^g[j]$ = Number of multicast messages received from p_j that happened-before the next message to be sent

- **Algorithm for group member p_i:**

  Initialization

  $V_i^g[j] := 0 \ (j = 1, \ldots, N);$

Causal Ordering

CO-multicast(g, m)

$V_i^g[i] := V_i^g[i] + 1;$

B-multicast(g, $<m, V_i^g>$);

B-deliver($<m, V_j^g>$) of p_j, with g = group(m)

Place $<m, V_j^g>$ in a hold-back queue;

Wait until ($V_j^g[j] = V_i^g[j] + 1$) AND ($V_j^g[k] \leq V_i^g[k]$);
$(k \neq j)$

CO-deliver(m);

$V_i^g[j] := V_i^g[j] + 1;$

Consensus introduction

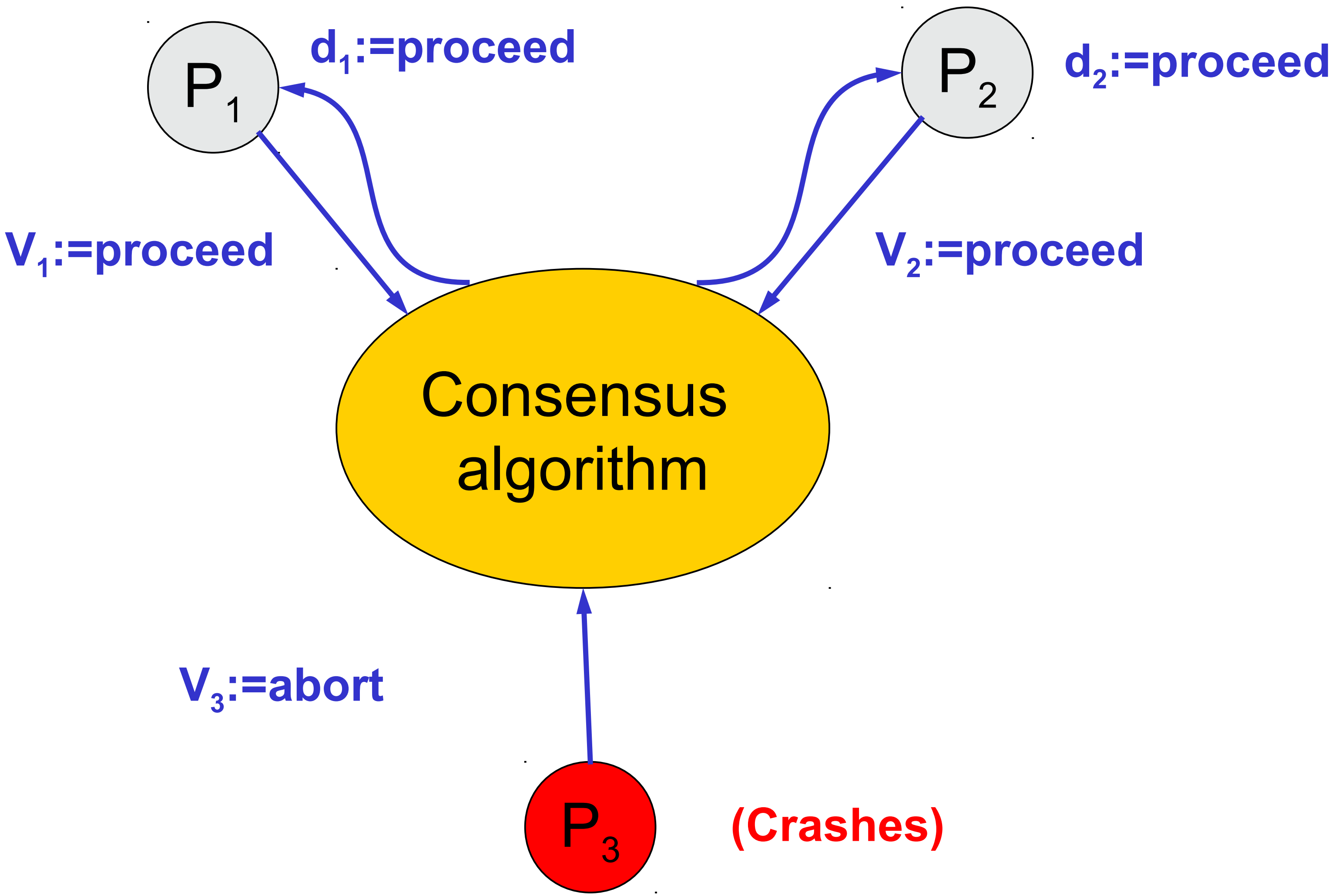
- **Make agreement in a distributed manner**
  - Mutual exclusion: who can enter the critical region
  - Totally ordered multicast: the order of message delivery
  - Byzantine generals: attack or retreat?
- **Consensus problem**
  - Agree on a value after one or more of the processes has proposed what the value should be

Consensus

- **Objective:** processes must agree on a value after one or more of the processes has proposed what that value should be
- **Hypotheses:** reliable communication, but processes may fail
- **Consensus problem:**
  - Every process P_i begins in the *undecided* state
  - Proposes a value $V_i \subset D$ (i=1, …, N)
  - Processes communicate with one another, exchanging values
  - Each process then sets the value of a decision variable d_i

Enters the state *decided*, in which it may no longer change d_i (i=1, …, N)

Consensus



P1
d1:=proceed
P2
d2:=proceed
V1:=proceed
V2:=proceed
Consensus algorithm
V3:=abort
P3
(Crashes)

Consensus

- **Proprieties to satisfy:**
  - **Termination:** Eventually each correct process sets its decision variable
  - **Agreement:** the decision value of all correct processes is the same:
    P_i and P_j are correct $\rightarrow d_i = d_j$ (i,j=1, …, N)
  - **Integrity:** If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value

Consensus

- Co
Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds
  - *On initialization*
  - $Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;
  - *In round* r $(1 \le r \le f + 1)$
  - *B-multicast*$(g, Values_i^r - Values_i^{r-1})$;

    $Values_i^{r+1} := Values_i^r$;

    while (in round r)

    {

    On B-deliver(V_j) *from some* p_j

    $Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

    }

After $(f + 1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

$Values_i^r$: set of proposed values known to process p_i at the beginning of round r

Consensus

- **Interactive consistency problem:** variant of the consensus problem
- **Objective:** correct processes must agree on a *vector* of values, one for each process
- **Proprieties to satisfy:**
  - **Termination:** Eventually each correct process sets its decision variable
  - **Agreement:** the decision vector of all correct processes is the same
  - **Integrity:** If P_i is correct, then all correct processes decide on V_i as the i^{th} component of their vector

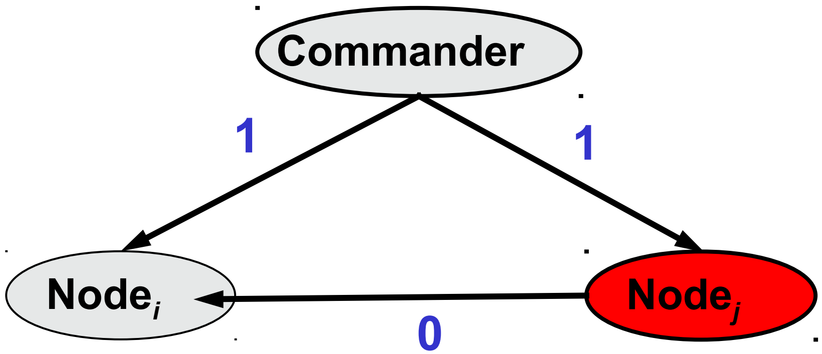
Consensus

- **Byzantine generals problems:** variant of the consensus problem
- **Objective:** a distinguished process supplies a value that the others must agree upon
- **Proprieties to satisfy:**
  - **Termination:** Eventually each correct process sets its decision variable
  - **Agreement:** the decision value of all correct processes is the same:
    Pi and Pj are correct → di = dj (i,j=1, …, N)
  - **Integrity:** If the commander is correct, then all correct processes decide on the value that the commander proposed

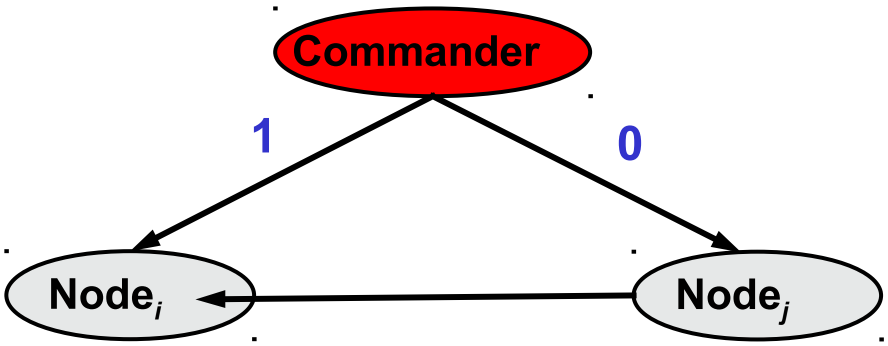
Consensus

- **Byzantine agreement in a synchronous system:**
  - **Example :** a system composed of three processes (must agree on a binary value 0 or 1)

Scenario 1: process *j* is faulty



Scenario 2: Commander is faulty



Number of faulty processes must be bounded

Consensus

- For *m* faulty processes, ***$n \geq 3m+1$****, where n* denotes the total number of processes
- **Interactive Consistency Algorithm:** *ICA(m), m>0, m* denotes the maximal number of processes that may fail simultaneously
  - **Sender:** all nodes must agree upon its value
  - **Receivers:** all other processes

- If a process doesn't send a message, the receiving process will use a default value $\perp$
- *ICA* Algorithm requires ***$m+1$*** rounds in order to achieve the consensus

Consensus

- **Interactive Consistency Algorithm:**

Algorithm ***ICA(0)***

1. The sender sends its value to all the other *n-1* processes
2. Each process uses the value received from the sender,

   or use the default value if no message is received

End

it sends the value V_i to the *n-2* other processes

3. $\forall$ *i*, Let V_j be the value received from process *j* ($j \neq i$)

The process *i* uses the value $Choice(V_1, \ldots, V_n)$

End