



Module-4

Coordination and agreement

Department of
Computer Science & Engineering

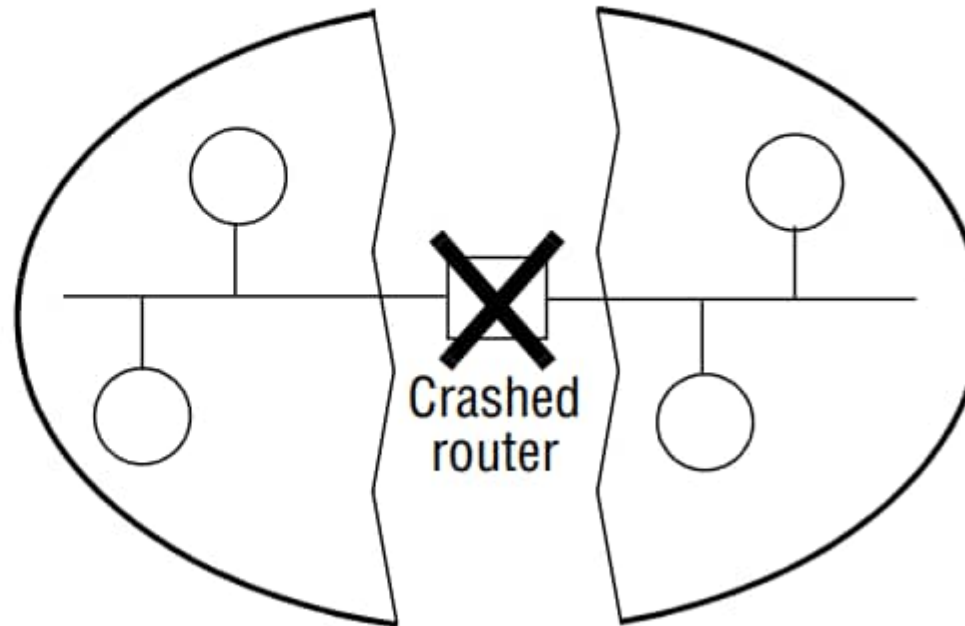
notesforvtu.in

www.cambridge.edu.in



Failure assumptions and failure detectors

A network partition



Failure assumptions and failure detectors

Over a point-to-point network such as the Internet, complex topologies and independent routing choices mean that connectivity may be

- Asymmetric connectivity: communication is possible from process p to process q , but not vice versa.
- Intransitive connectivity: communication is possible from p to q and from q to r , but p cannot communicate directly with r .

Failure assumptions and failure detectors

- One of the problems in the design of algorithms that can overcome process crashes is that of deciding when a process has crashed.
- A **failure detector** is a service that processes queries about whether a particular process has failed.
- It is often implemented by an object local to each process (on the same computer) that runs a **failure-detection algorithm** in conjunction with its counterparts at other processes.
- The object local to each process is called a **local failure detector**.

Failure assumptions and failure detectors

- An **unreliable failure** detector may produce one of two values when given the identity of a process:
 - ✓ A result of **Unsuspected** signifies that the detector has recently received evidence suggesting that the process has not failed;
 - ✓ A result of **Suspected** signifies that the failure detector has some indication that the process may have failed.

Failure assumptions and failure detectors

- A reliable failure detector is one that is always accurate in detecting a process's failure.

Distributed Mutual Exclusion

- If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources.

Algorithms for mutual exclusion

The application-level protocol for executing a critical section is as follows:

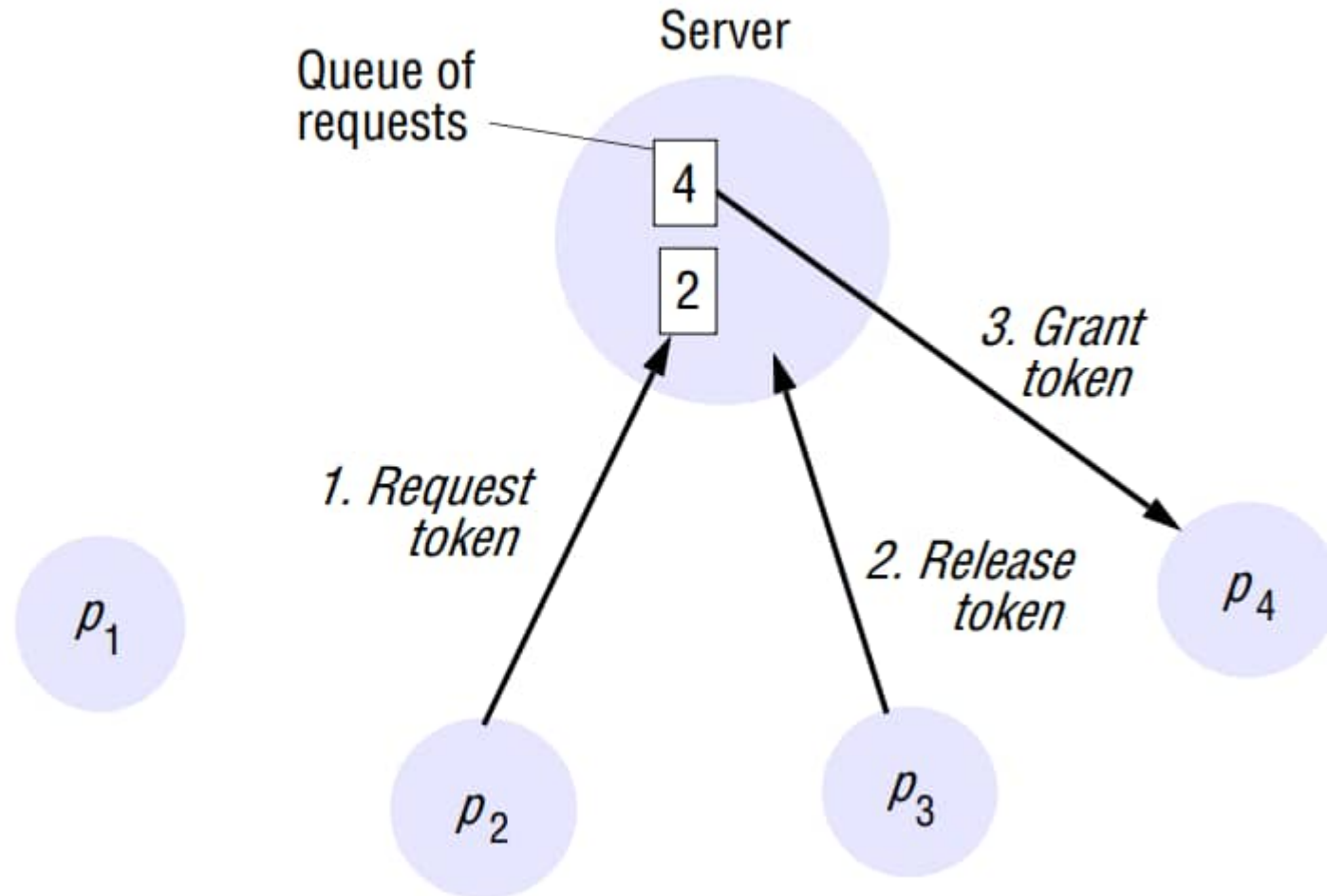
```
enter()           // enter critical section – block if necessary  
resourceAccesses() // access shared resources in critical section  
exit()          // leave critical section – other processes may now enter
```

Our essential requirements for mutual exclusion are as follows:

- | | |
|-----------------|---|
| ME1: (safety) | At most one process may execute in the critical section (CS) at a time. |
| ME2: (liveness) | Requests to enter and exit the critical section eventually succeed. |

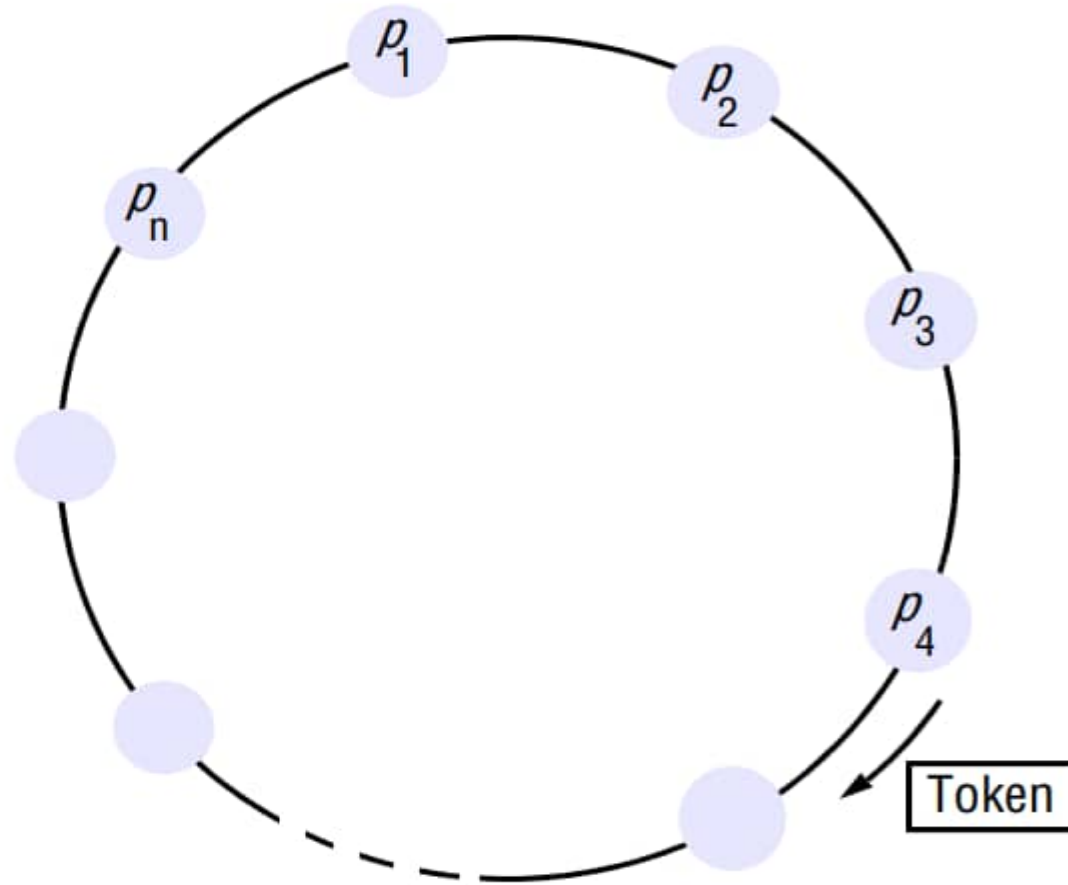
The central server algorithm

Server managing a mutual exclusion token for a set of processes



A ring-based algorithm

A ring of processes transferring a mutual exclusion token



Ricart and Agrawala's algorithm

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;

Request processing deferred here

On receipt of a request $\langle T_i, p_i \rangle$ 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 ;

end if

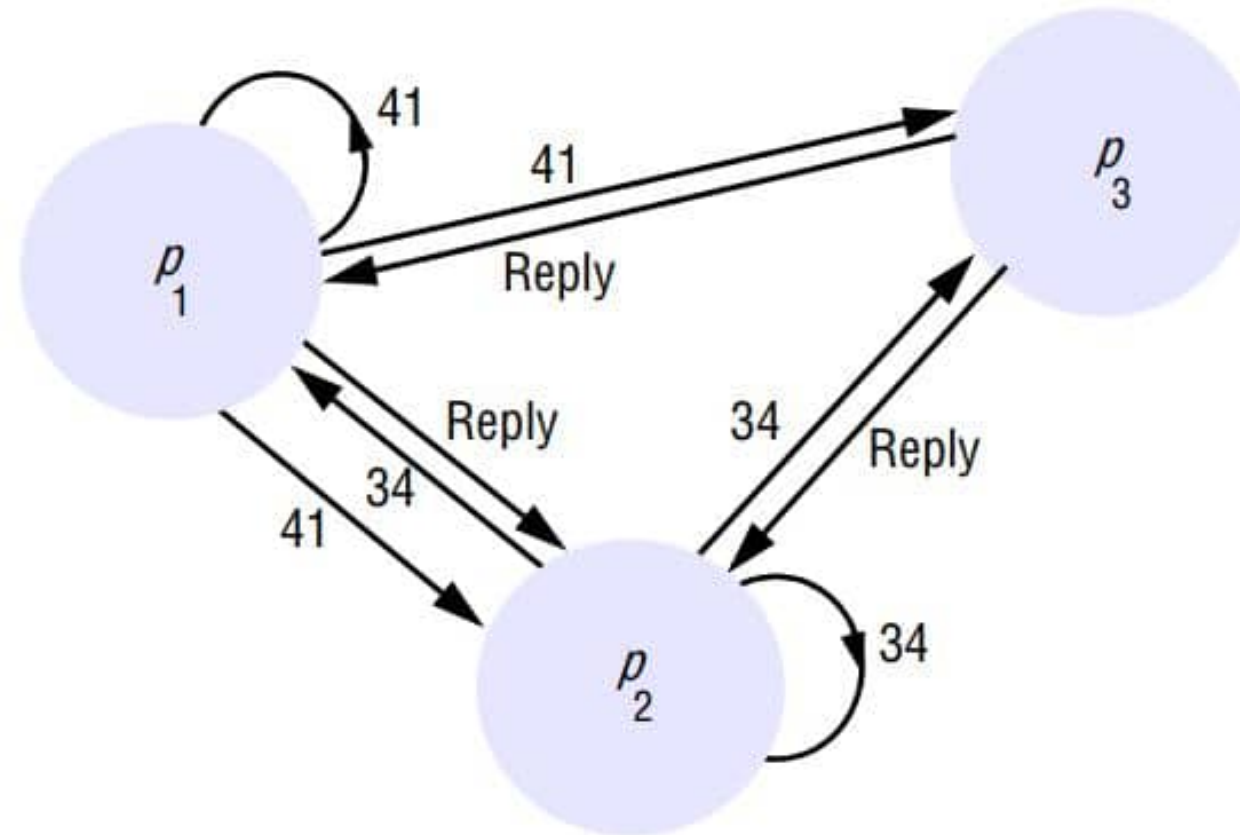
To exit the critical section

state := RELEASED;

reply to any queued requests;

An algorithm using multicast and logical clocks

Multicast synchronization



Maekawa's voting algorithm

- Maekawa observed that in order for a process to enter a critical section, it is not necessary for all of its peers to grant it access.
- Processes need only obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.
- A 'candidate' process must collect sufficient votes to enter.
- Processes in the intersection of two sets of voters ensure the safety property ME1, that at most one process can enter the critical section, by casting their votes for only one candidate.

Maekawa's voting algorithm

Maekawa associated a *voting set* V_i with each process p_i ($i = 1, 2, \dots, N$), where $V_i \subseteq \{p_1, p_1, \dots, p_N\}$. The sets V_i are chosen so that, for all $i, j = 1, 2, \dots, N$:

- $p_i \in V_i$
- $V_i \cap V_j \neq \emptyset$ – there is at least one common member of any two voting sets
- $|V_i| = K$ – to be fair, each process has a voting set of the same size
- Each process p_j is contained in M of the voting sets V_i .

Maekawa's voting 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

Maekawa's voting algorithm

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

Election

- An algorithm for choosing a unique process to play a particular role is called an **election algorithm**.
- A process calls the election if it takes an action that initiates a particular run of the election algorithm.
- An individual process does not call more than one election at a time, but in principle the N processes could call N concurrent elections.
- At any point in time, a process p_i is either a **participant** – meaning that it is engaged in some run of the election algorithm – or
- a **non-participant** – meaning that it is not currently engaged in any election.

Ring-based election algorithm

- The algorithm of Chang and Roberts [1979] is suitable for a collection of processes arranged in a logical ring.
- Each process p_i has a communication channel to the next process in the ring, and all messages are sent clockwise around the ring.
- We assume that no failures occur, and that the system is asynchronous.
- The goal of this algorithm is to elect a single process called the coordinator, which is the process with the largest identifier.

Ring-based election algorithm

- Initially, every process is marked as a *non-participant* in an election.
- Any process can begin an election. It proceeds by marking itself as a participant, placing its identifier in an election message and sending it to its clockwise neighbor.

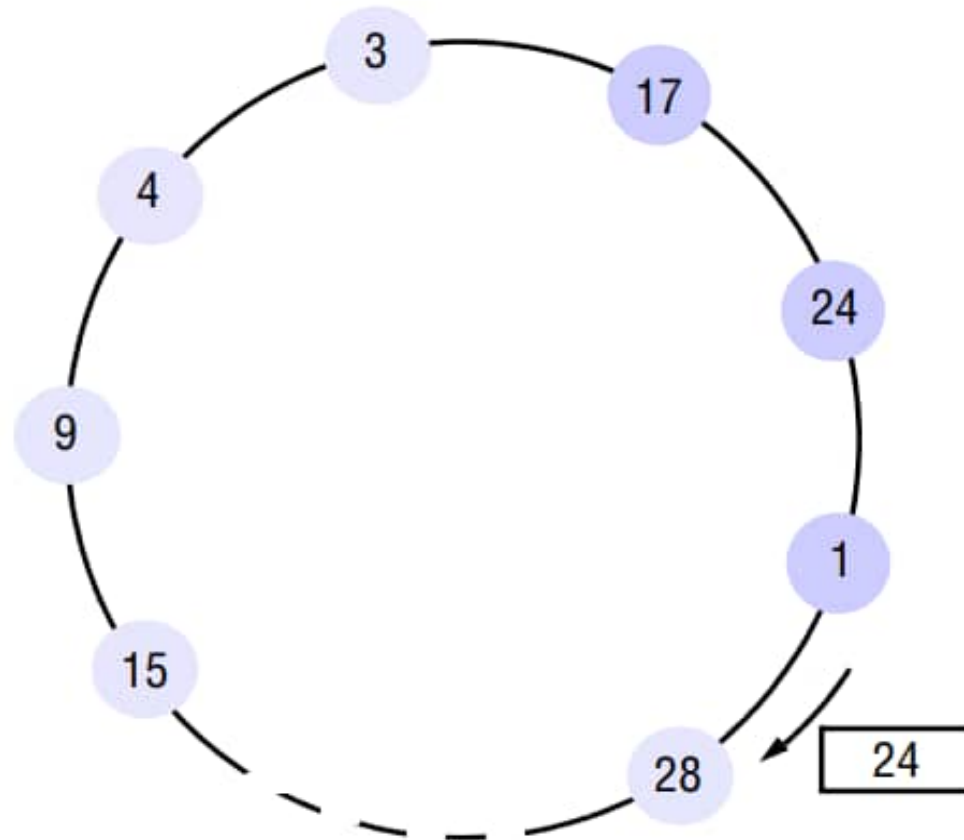
Ring-based election algorithm

- When a process receives an election message, it compares the identifier in the message with its own.
 - ❖ If the arrived identifier is greater, then it forwards the message to its neighbor.
 - ❖ If the arrived identifier is smaller and the receiver is not a participant, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a participant.

Ring-based election algorithm

- On forwarding an election message in any case, the process marks itself as a participant.
- If, the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the coordinator.
- The coordinator marks itself as a non-participant once more and sends an elected message to its neighbor, announcing its election and enclosing its identity.

A ring-based election in progress



Note: The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown in a darker tint.

The bully algorithm

- The bully algorithm [Garcia-Molina 1982] allows processes to crash during an election, although it assumes that message delivery between processes is reliable.
- Unlike the ring-based algorithm, this algorithm assumes that the system is synchronous: it uses timeouts to detect a process failure.
- Another difference is that the ring-based algorithm assumed that processes have minimal a priori knowledge of one another: each knows only how to communicate with its neighbor, and none knows the identifiers of the other processes.

The bully algorithm

- There are three types of message in this algorithm:
- an **election message** is sent to announce an election
- an **answer message** is sent in response to an election message
- a **coordinator message** is sent to announce the identity of the elected process— the new ‘coordinator’.

The bully algorithm

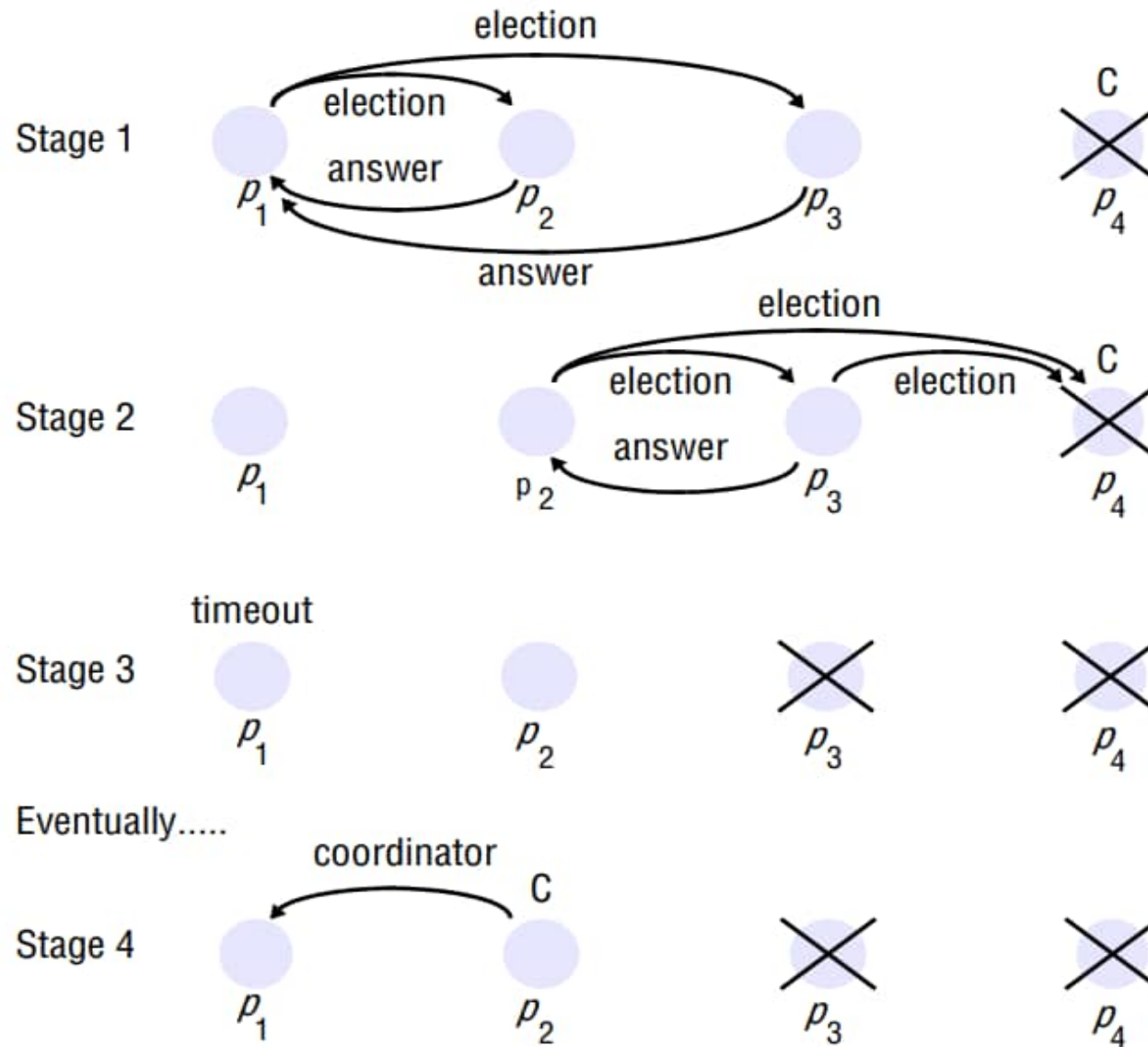
- The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a coordinator message to all processes with lower identifiers.
- a process with a lower identifier can begin an election by sending an election message to those processes that have a higher identifier and awaiting answer messages in response.
- If none arrives within time T , the process considers itself the coordinator and sends a coordinator message to all processes with lower identifiers announcing this.
- Otherwise, the process waits a further period T_c for a coordinator message to arrive from the new coordinator. If none arrives, it begins another election.

The bully algorithm

- If a process p_i receives a coordinator message, it sets its variable *elected_i* to the identifier of the coordinator contained within it and treats that process as the coordinator.
- If a process receives an election message, it sends back an answer message and begins another election – unless it has begun one already.

The bully algorithm

- When a process is started to replace a crashed process, it begins an election.
- If it has the highest process identifier, then it will decide that it is the coordinator and announce this to the other processes. Thus it will become the coordinator, even though the current coordinator is functioning.
- It is for this reason that the algorithm is called the 'bully' algorithm.



The election of coordinator p_2 , after the failure of p_4 and then p_3

Coordination and Agreement in group communication

System model:

- The system under consideration contains a collection of processes, which can communicate reliably over one-to-one channels.
- As before, processes may fail only by crashing.
- The processes are members of groups, which are the destinations of messages sent with the multicast operation.
- The operation $\text{multicast}(g, m)$ sends the message m to all members of the group g of processes.
- Correspondingly, there is an operation $\text{deliver}(m)$ that delivers a message sent by multicast to the calling process.