DISTRIBUTED COMPUTING
Module VI

COORDINATION AND AGREEMENT

DISTRIBUTED MUTUAL EXCLUSION

- Distributed processes often need to coordinate their activities
- Mutual exclusion is required to prevent interference and ensure consistency when accessing the shared resources
- This is the ***critical section problem*** *(familiar in operating systems)*
- A solution to *distributed mutual exclusion is required: based solely on message passing*

Algorithms for mutual exclusion

- Consider a system of *N processes pi, i = 1, 2,. ., N, that do not share variables*
- The processes access common resources, but in a critical section
- Assume that there is only one critical section(can extend to more than one critical section)
- Assumption:
  - system is asynchronous
  - processes do not fail and message delivery is reliable
  - any message sent is eventually delivered intact, exactly once
- 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*

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

- **Condition ME2 implies freedom from both deadlock and starvation**

- Even without a deadlock, a poor algorithm might lead to *starvation: the indefinite* postponement of entry for a process that has requested it.

- The absence of starvation is a ***fairness condition****.*

- Another fairness issue: order in which processes enter the critical section

- Cannot **order entry** to the critical section by time due to **absence of global clocks**

- **Happened-before ordering** may be used

  ME3: (-> ordering) If one request to enter the CS happened-before another, then entry to the CS is granted in that order

- Hence it is not possible for a process to enter the critical section more than once while another waits to enter

- We evaluate the performance of algorithms for mutual exclusion according to the following criteria:

✔ **the *bandwidth consumed***

(number of messages sent in* each *entry and exit operation)

✔ **the *client delay incurred by a process at each entry and exit operation***

✔ **the algorithm's effect upon the *throughput of the system***

***Rate at which* the collection of processes as a whole can access the critical section**

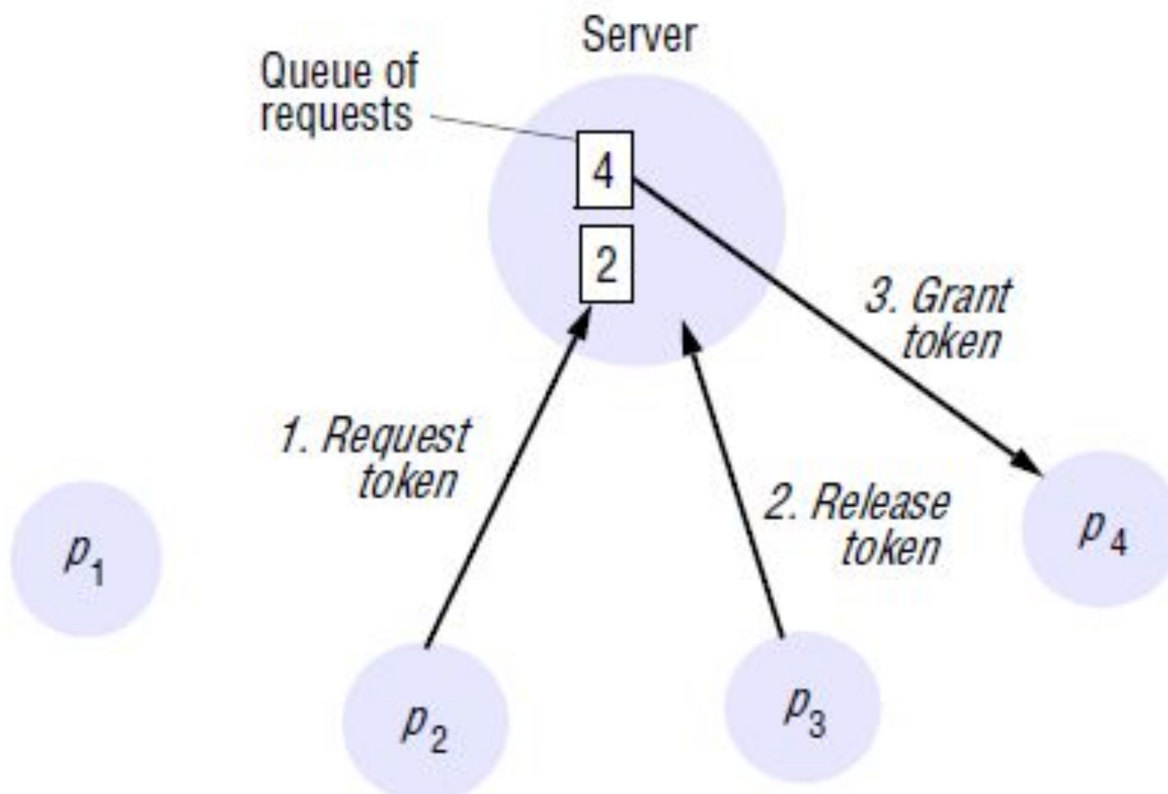
- The effect is measured using the *synchronization delay between one process exiting the critical* section and the next process entering it

- Throughput is greater when the synchronization delay is shorter

The central server algorithm

- Simplest way to achieve mutual exclusion
- Server grants permission to enter the critical section

Server managing a mutual exclusion token for a set of processes



- To enter a critical section, a process sends a request message to the server and awaits a reply from it.
- Conceptually, the reply constitutes a token signifying permission to enter the critical section.
- If no other process has the token at the time of the request, then the server replies immediately, granting the token.
- If the token is currently held by another process, then the server does not reply, but queues the request.
- When a process exits the critical section, it sends a message to the server, giving it back the token.
- If the queue of waiting processes is not empty, then the server chooses the oldest entry in the queue, removes it and replies to the corresponding process.
- The chosen process then holds the token.
- Safety and liveness conditions are met by this algorithm.

Performance of this algorithm:

- Entering the critical section takes two messages (a *request followed* by a *grant)*

- *Delays the requesting process by the time required for this round-trip*

- Exiting the critical section takes one *release message*

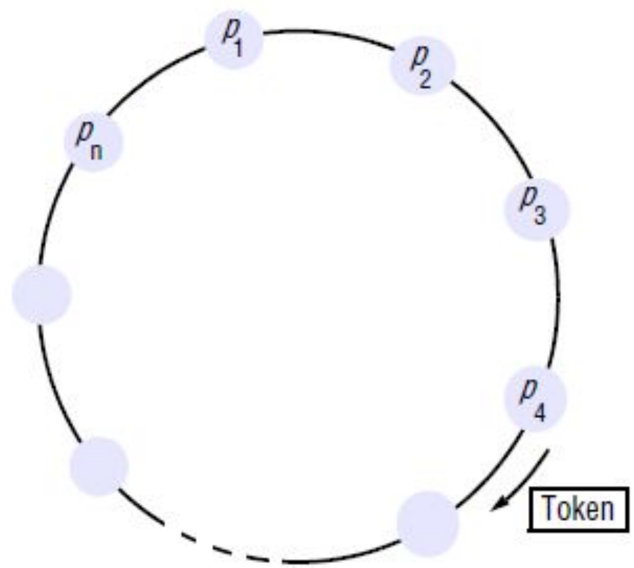
- *T*his does not delay the exiting process

- The server may become a performance bottleneck for the system

- The synchronization delay is the time taken for a round-trip:
  - a *release message to the server*
  - a *grant message to the next process to enter the critical section*

A ring-based algorithm

- Simple way to arrange mutual exclusion between the *N processes*
- *No additional process required to arrange them in a logical* ring
- Requirement: *a communication channel from* each process *Pi has to the next* process in the ring, *P(i + 1) mod N*
- *Exclusion is conferred by obtaining a* token in the form of a message passed from process to process in a single direction
- The ring topology may be unrelated to the physical interconnections between the underlying computers



- If a process does not require to enter the critical section when it receives the token, then it immediately forwards the token to its neighbour.
- A process that requires the token waits until it receives it, but retains it.
- To exit the critical section, the process sends the token on to its neighbour.
- Conditions **ME1 and ME2 are met** by this algorithm
- But, the token is **not necessarily obtained in happened-before order**
- Processes may exchange messages independently of the rotation of the token

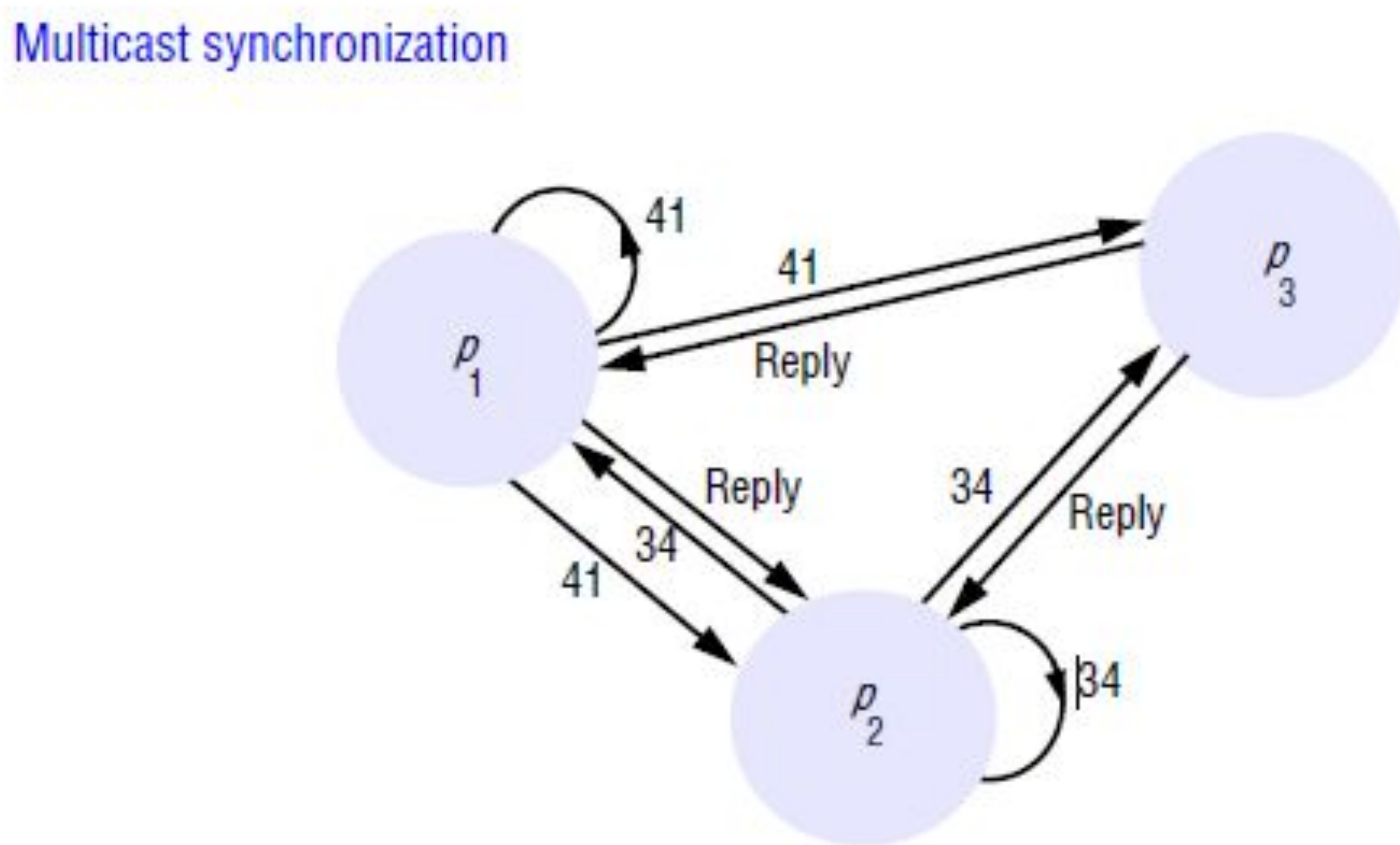
- This algorithm continuously consumes network bandwidth
  (except when a process is inside the critical section)

- Delay experienced by a process requesting entry to the critical section:
  between 0 messages (just received the token) and
  N messages (just passed on the token)

- *To exit the* critical section requires only one message

- Synchronization delay between one process's exit from the critical section and the next process's entry = From 1 to *N message transmissions*

An algorithm using multicast and logical clocks (Ricart and Agrawala's)



Ricart and Agrawala's Algorithm

On initialization
    state := RELEASED;

To enter the section
    state := WANTED;
    Multicast *request* to all processes;   *Request processing deferred her*
    T := request's timestamp;
    Wait until (number of replies received = $(N - 1)$);
    state := HELD;

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;
    end if

To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Maekawa’s voting algorithm

- Not necessary for all of its peers to grant it access
- Needonly obtain permission to enter from *subsets of their peers*
- *The subsets* used by any two processes should overlap
- Processes vote for one another to enter the critical section
- A ‘candidate’ process must collect sufficient votes to enter
- Processes in the intersection of two sets of voters ensure the safety property ME1

- Let there be 7 processes- 0,1,2,3,4,5,6

$$S_0 = \{0, 1, 2\}$$
$$S_1 = \{1, 3, 5\}$$
$$S_2 = \{2, 4, 5\}$$
$$S_3 = \{0, 3, 4\}$$
$$S_4 = \{1, 4, 6\}$$
$$S_5 = \{0, 5, 6\}$$
$$S_6 = \{2, 3, 6\}$$

- Voting set V_i with each process p_i ($i = 1, 2, \ldots N$), where V_i is a subset of $\{p_1, p_2, \ldots, p_N\}$

- The sets V_i are chosen so that, for all $i, j = 1, 2, \ldots, N$:
  - **$p_i \in V_i$**
  - **$V_i \cap V_j \neq \phi$ – 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**

- Optimal solution:
  - minimizes *K and allows the* processes to achieve mutual exclusion
  - *$K \sim \sqrt{N}$ and $M = K$ (so that each process is* in as many of the voting sets as there are elements in each one of those sets).

- It is nontrivial to calculate the optimal sets *R_i*

```
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

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
```

- To obtain entry to the critical section, a process *pi sends request messages to all K members of Vi (including itself).*
- *pi cannot enter the critical section until it has received all K reply messages.*
- *When a* process *pj in Vi receives pi 's request message, it sends a reply message immediately,* unless either its state is *HELD or it has already replied ('voted') since it last received a release message.*
- *Otherwise, it queues the request message (in the order of its arrival)* but does not yet reply.
- When a process receives a *release message, it removes the head* of its queue of outstanding requests (if the queue is nonempty) and sends a *reply* message (a 'vote') in response to it.
- To leave the critical section, *pi sends release* messages to all *K members of Vi (including itself).*

- This algorithm achieves the safety property, ME1

Unfortunately, the algorithm is deadlock-prone:

- Consider three processes, *p1 , p2* and *p3 , with*
  - *V1= { p1, p2}*
  - *V2 = { p2, p3}*
  - *V3 = {p3, p1}*
- *If the three* processes concurrently request entry to the critical section
- *Each process has received one out of two replies, and* none can proceed

- The algorithm can be adapted so that it becomes deadlock-free
- Processes may queue outstanding requests in happened-before order
- ME3 is also satisfied
- The algorithm's bandwidth utilization is
  - 2 *N messages per entry to the critical* section
  - *N messages per exit (assuming no hardware multicast facilities)*

ELECTIONS

- Algorithm for choosing a unique process to play a particular role is called an *election algorithm*
  - *For example, in a variant of central-server algorithm for mutual* exclusion, the 'server' is chosen from among the processes *pi,(i = 1, 2,... N) that* need to use the critical section
- An election algorithm is needed for this choice
- It is essential that all the processes agree on the choice
- Afterwards, if the server wishes to retire then another election is required to choose a replacement

- Process **calls the election-** takes an action that initiates a run of the election algorithm.
- An individual process does not call more than one election at a time
- **N processes could call N concurrent elections**
- At any point in time, a process pi is either a **participant** or a **non-participant**
- The **elected process** must be **unique**
  - For example, two processes could decide independently that a coordinator process has failed, and both call elections
- The elected process be chosen as **the one with the largest identifier**
- The ‘identifier’ may be any useful value, unique and totally ordered

- Each process *has a variable electedi , which will contain the* identifier of the elected process

- Variable electedi set to the special value '$\perp$' initially to denote that it is not yet defined.

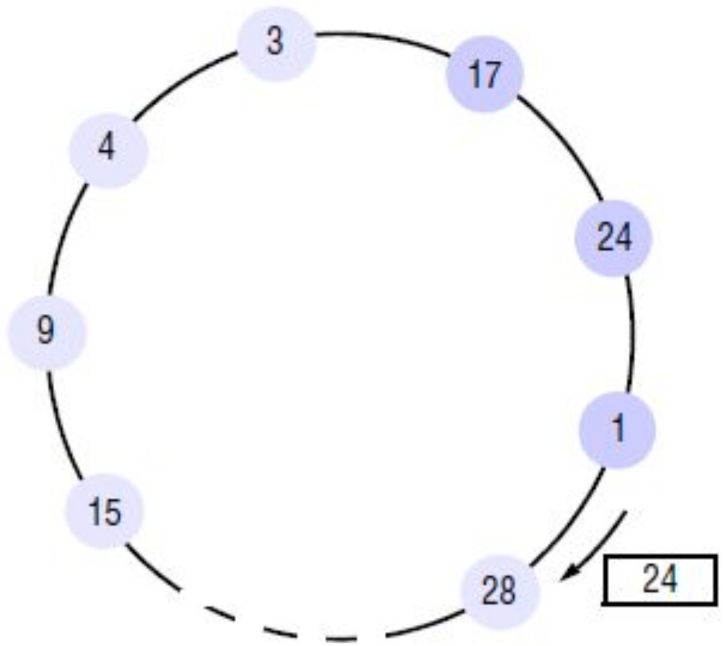
Requirements during any particular run of the algorithm:

- **E1: (safety)** A participant process *pi has electedi =* $\perp$ *or electedi = P,* where *P is non-crashed process chosen at the end of* the run with the largest identifier
- **E2: (liveness)** All processes *pi participate and eventually either set electedi ≠* $\perp$ *– or crash*

- There may be processes *pj that are not yet participants, which record in electedj the identifier of the previous elected process*

A RING-BASED ELECTION ALGORITHM

- Each process *pi has a* communication channel to the next (*pi + 1 mod N)*
- Assumption: no failures occur, and the system is asynchronous
- Goal: To elect a single process called the *coordinator, which is the process with the largest identifier*



- Initially, every process is marked as a *non-participant*

- *Any process* can begin an election
  - marks itself as a *participant*
  - *placing its identifier* in an *election message*
  - *sending it to its clockwise neighbour*

- When a process receives an *election message*
  - *it compares the identifier in the* message with its own
  - If the arrived identifier is greater- forwards the message to its neighbour
  - 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
  - Does not forward the message if it is already a *participant*

- *On forwarding an election message* the process marks itself as a *participant*

- If the received identifier is that of the receiver itself
  - Its identifier must be the greatest, and it becomes the coordinator

- The coordinator
  - marks itself as a *non-participant once more*
  - *sends an elected message to its neighbour,* announcing its election and enclosing its identity

- When a process *pi receives an elected message:*
  - *it marks itself as a nonparticipant*
  - sets its variable *electedi to the identifier in the message*
  - *unless it is the* new coordinator, forwards the message to its neighbour

Condition E1 is met:

- All identifiers are compared, since a process must receive its own identifier back before sending an *elected message.*
- *For any* two processes the one with the larger identifier will not pass on the other's identifier
- It is therefore impossible that both should receive their own identifier back.

Condition E2 is met:

- Follows immediately from the guaranteed traversals of the ring (if there are no failures)
- The *non-participant and participant states*
  - *Eliminate duplicate messages* as soon as possible, and always before the 'winning' election result has been announced

Performance if only a single process starts an election:

- Worst-performing case - its anti-clockwise neighbour has the highest identifier
  - A total of *N – 1 messages are* then required to reach this neighbour
  - Its election announced completed with another *N messages*
  - *The elected* message is then sent *N times*
  - *Total = 3N – 1 messages in all*
  - *The turnaround time is* also 3*N – 1*

- Ring-based algorithm is useful for understanding the properties of election algorithms

- Not practical as it tolerates no failures

- A reliable failure detector may help to reconstitute the ring when a process crashes

THE BULLY ALGORITHM

- The bully algorithm allows processes to crash during an election

- It assumes that message delivery between processes is reliable

- Assumes that the **system is synchronous**:
  it uses timeouts to detect a process failure

- Another difference
  - The ring-based algorithm assumed that processes have minimal *a priori knowledge of one* another, communicates with only its neighbour

  - The bully algorithm assumes that **each process knows which processes have higher identifiers**, and can communicate with all such processes

There are three types of message in this algorithm:

- **election message** is sent to announce an election
- **answer message** is sent in response to an election message
- **coordinator message** is sent to announce the identity of the elected process – the new ‘coordinator’.

- A process begins an election when it notices, through timeouts, that the coordinator has failed

- Several processes may discover this concurrently

- Since the system is synchronous, we can construct a reliable failure detector

- Maximum message transmission delay = **Ttrans**

- Maximum delay for processing a message **Tprocess** .

- **Round trip time T = 2Ttrans + Tprocess**
  - Upper bound on the time that can elapse between sending a message to another process and receiving a response.

- If no response arrives within time T-
  - local **failure detector** can report that intended recipient has failed

- 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.

- On the other hand, 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' for a coordinator* message to arrive from the new coordinator.

- If none arrives, it begins another election.

- If a process *pi receives a coordinator message, it sets its variable electedi 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.

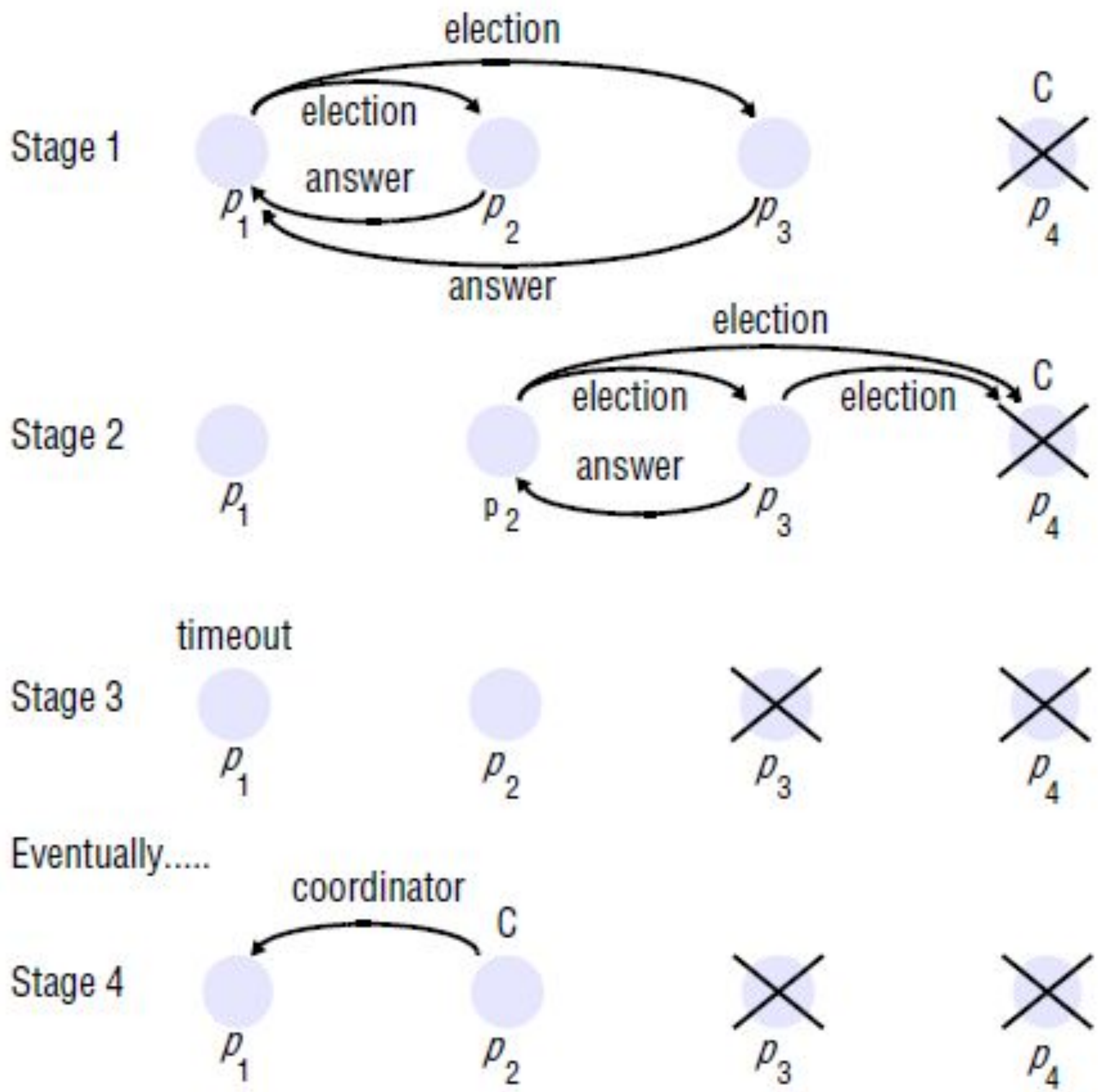
When a process is started, to replace a crashed process, it begins an election

- 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 bully algorithm



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

There are four processes, p1 –p4 .

- **Stage1**: Process p1 detects the failure of the coordinator p4 and announces an election.

  On receiving an election message from p1 , processes p2 and p3 send answer messages to p1 and begin their own elections;

- **Stage 2**: p3 sends an answer message to p2 , but p3 receives no answer message from the failed process p4.

  It therefore decides that it is the coordinator.

- **Stage 3**: But before it can send out the coordinator message, it too fails. When p1 ’s timeout period T’ expires (which we assume occurs before p2 ’s timeout expires), it deduces the absence of a coordinator message and begins another election.

- **Stage 4**: Eventually, p2 is elected coordinator.

- **Liveness condition E2 is met** by assuming reliable message delivery

- Condition E1 is met if **no process is replaced**

- ***Safety condition E1 not met if***
  - ***processes* that have crashed are replaced by processes with the same identifiers**
  - the assumed timeout values turn out to be inaccurate (suppose p3 was just slow)

Performance of the algorithm:

- Best case - the process with the second-highest identifier notices the coordinator's failure
  - Then it can immediately elect itself and send *$N-2$ coordinator messages.*
  - *The turnaround time is one message.*

- Worst case - process with lowest identifier first detects the coordinator's failure
  - The bully algorithm requires *$O(N^2)$ messages*
  - For then *$N-1$* processes altogether begin elections, each sending messages to processes with higher identifiers.

- This algorithm clearly meets the **liveness condition E2**, by the assumption of reliable message delivery.
- And if **no process is replaced**, then the algorithm **meets condition E1.**
- *It is impossible for two processes to decide that they are the coordinator,* since the process with the lower identifier will discover that the other exists and defer to it.
- But the algorithm is ***not guaranteed to meet the safety condition E1 if processes*** **that have crashed are replaced by processes with the same identifiers.**
- A process that replaces a crashed process *p may decide that it has the highest identifier just as another* process (which has detected *p's crash) decides that it has the highest identifier.*
- ***Two*** **processes will therefore announce themselves as the coordinator concurrently**.
- Unfortunately, there are no guarantees on message delivery order, and the recipients of these messages may reach different conclusions on which is the coordinator process.
- Furthermore, condition E1 may be broken if the assumed timeout values turn out to be inaccurate – that is, if the processes' failure detector is unreliable.

- Suppose that either *p3 had not failed but was just* running unusually slowly (that is, that the assumption that the system is synchronous is incorrect), or that *p3 had failed but was then replaced.*

- *Just as p2 sends its coordinator* message, *p3 (or its replacement) does the same.*

- *p2 receives p3 ‘s coordinator message* after it has sent its own and so sets *elected2 = p3 .*

- *Due to variable message* transmission delays, *p1 receives p2 ’s coordinator message after p3 ’s and so* eventually sets *elected1 = p2 .*

- *Condition E1 has been broken.*

THANK YOU!