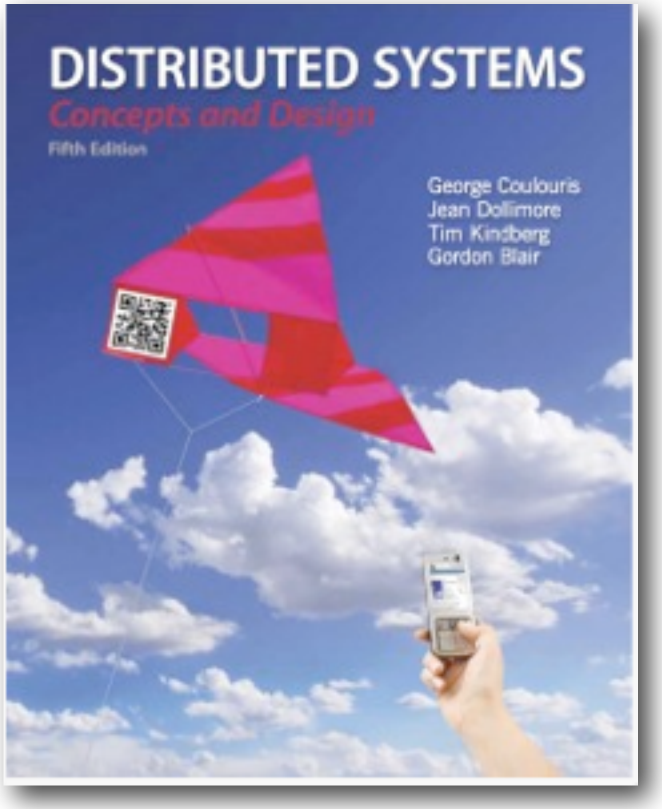
Coordination and Agreement



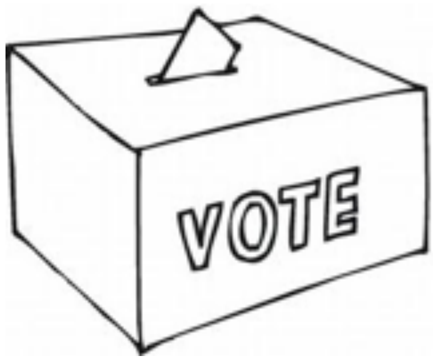
AIM: Coordination and/or Agreement

- Collection of algorithms whose goals vary

  but which share an aim that is fundamental in distributed systems

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

Election Algorithm



- An algorithm for choosing a unique process to play a particular role

- Example:
  - In a variant of the “central-server” algorithm for ME, the server is chosen from among the processes p_i, i = 1, 2, ..., N that need to use the CS
  - An election algorithm is needed to choose which of the processes will play the role of server
  - It is essential that all the processes agree on the choice
  - Afterwards, if the process that plays the role of server wishes to retire, then another election is required to choose a replacement

Roles and Election Calls

- At any point in time, a process p_i is either
  - a participant (meaning that it is engaged in some run of the algorithm)
  - or a non-participant (meaning that it is not currently engaged in any election)
- 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
- In principle, N processes could call N *concurrent* elections

Uniqueness of the Elected Process

- The choice of elected process must be **unique**, even if several processes call elections concurrently

- Without loss of generality, we require that the elected process be chosen as the one with the largest identifier

- The identifier may be any useful value, as long as the identifiers are **unique** and **totally ordered**

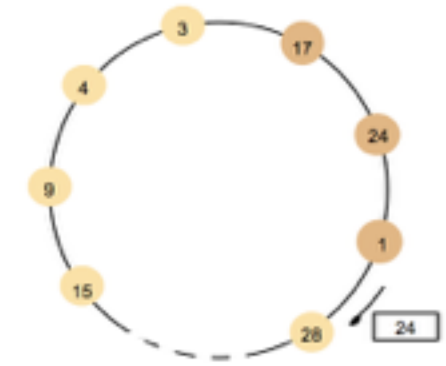
Example:
we could elect the process with the lowest computational load, by having each process use <1/load, i> as its identifier
where load > 0 and
the process index i is used to order identifiers with the same load

Election Algorithm Requirements

- Each process p_i has a variable $elected_i$, which will contain the identifier of the elected process
- Initially set to "$\perp$" (null, not defined)
- Requirements are that, during *any particular run* of the algorithm:
  - E1 (safety): A *participant* process p_i has $elected_i = \perp$ or $elected_i = P$ where P is chosen as the non-crashed process at the end of the run with the largest identifier
  - E2 (liveness): All processes p_i participate and eventually set $elected_i \neq \perp$ - or crash
- N.B.: there may be processes p_j that are not yet participants, which record in $elected_j$ the identifier of the previous elected process

Performance Parameters

- We measure the performance of an election algorithm by
  - its total network bandwidth utilization (proportional to the total number of messages sent)
  - the turnaround time: the number of serialized message transmission times between the initiation and termination of a single round



Ring-Based Election Algorithm

- Algorithm of Chang and Roberts [1979]

- 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 $p_{(i+1)modN}$

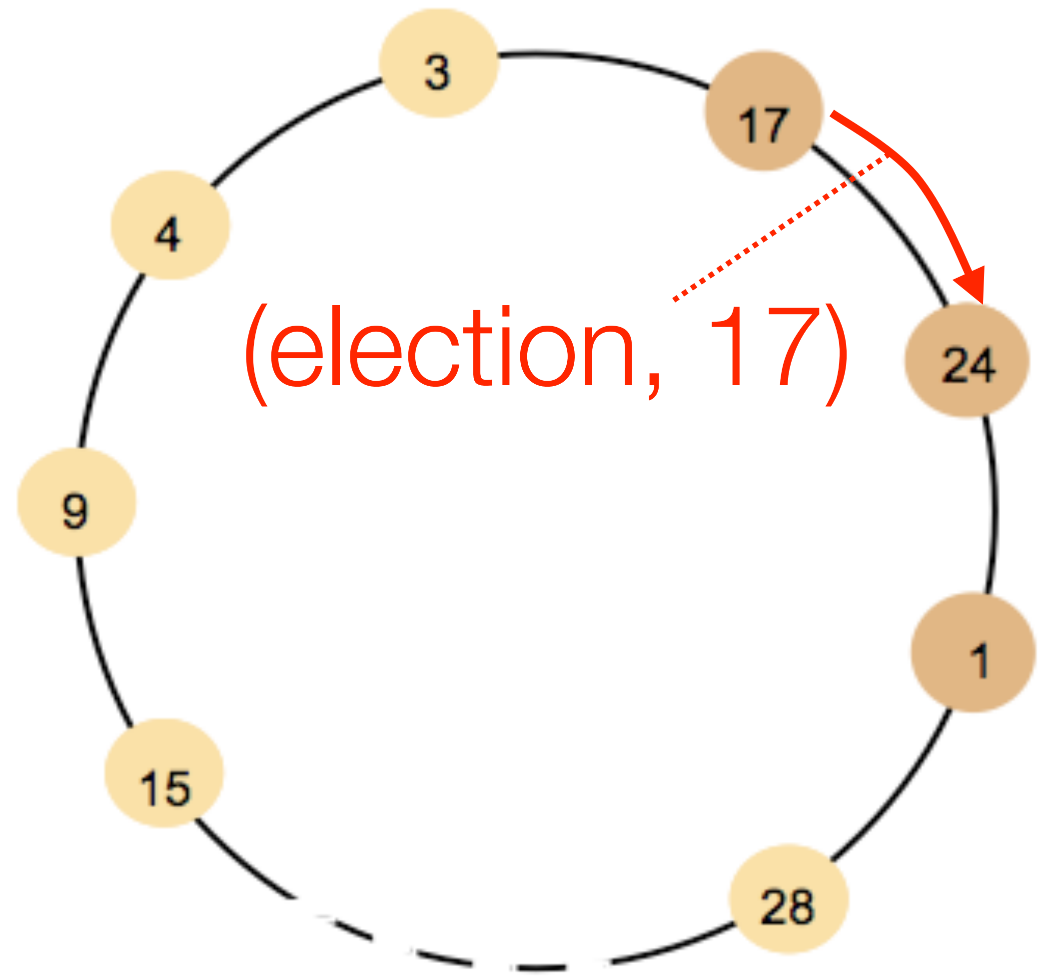
  - messages are sent clockwise around the ring

- No failures occur and the system is asynchronous

- Goal: elect a single process, called the **coordinator**, which is the process with the largest identifier

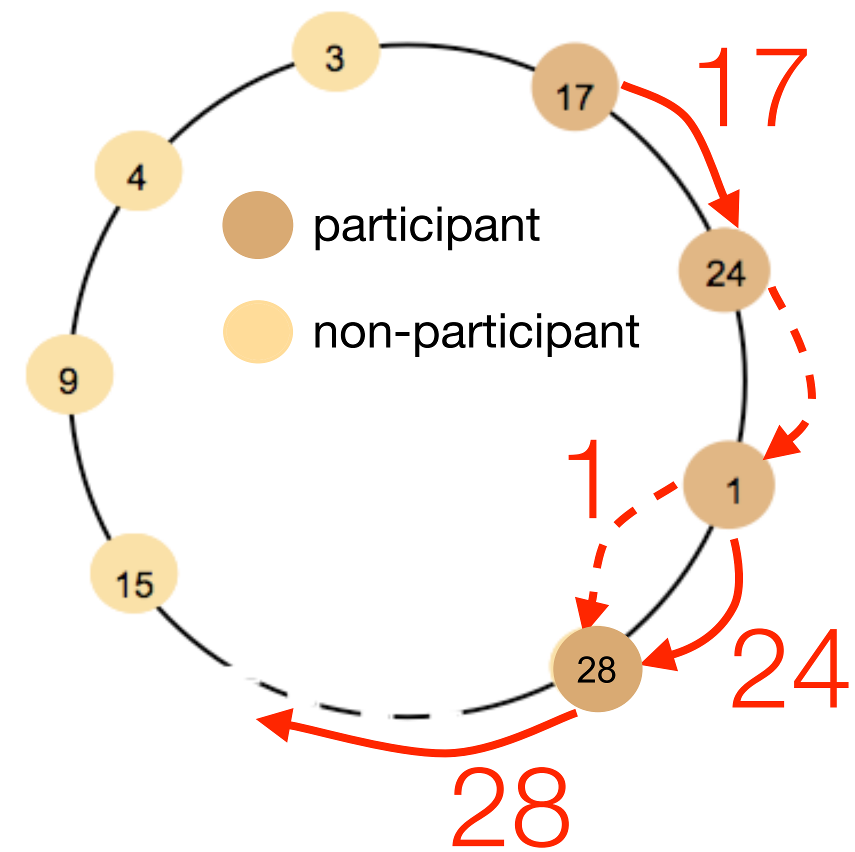
[Ring-Based Election Alg.] Starting an Election

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



[Ring-Based Election Alg.] Election

- 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 neighbour;
it marks itself as a participant

ELSIF the arrived identifier is smaller **AND**
the receiver is **not** a participant

THEN it substitutes its own identifier in the message;
it forwards the message to its neighbour;
it marks itself as a participant

ELSIF the arrived identifier is smaller **AND** the receiver is a participant

THEN it discards the message (i.e., it does not forward the message)

ELSE (the received identifier is that of the receiver itself)

this process's identifier must be the greatest: coordinator

[Ring-Based Election Alg.] Notification

- The coordinator marks itself as a non-participant once more
- Then it sends an *elected* message to its neighbour, announcing its election and enclosing its identity
- When a process p_i receives an *elected* message:
  - it marks itself as a non-participant
  - it sets its variable $elected_i$ to the identifier in the message
  - unless it is the new coordinator, forwards the message to its neighbour
- When the *elected* message reaches the newly elected process the election is over

Conditions: E1 (Safety)

- E1 (safety): A *participant* process p_i has $elected_i = \perp$ or $elected_i = P$ where P is chosen as the non-crashed process at the end of the run with the largest identifier.

- E1 is met (proof by contradiction). Idea:
  - 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 wins on the other's identifier
  - It is therefore impossible that both should receive their own identifier back

Conditions: E2 (Liveness)

- E2 (liveness): All processes p_i participate and eventually set $elected_i \neq \perp$ - or crash
- E2 is met. Idea:
  - It follows immediately from the guaranteed traversals of the ring (there are no failures)
  - Note how the non-participant and participant states are used so that messages arising when another starts an election at the same time are extinguished as soon as possible, and always before the "winning" election result has been announced

[Ring-Based Algorithm] Performance Analysis

- Total networks bandwidth utilization (proportional to the total number of messages sent):
  - If only a single process starts an election, then the worst-performing case is when its anti-clockwise neighbour has the highest identifier
  - A total of N-1 messages is then required to reach this neighbour
  - This neighbour will not announce its election until its identifier has completed another circuit, taking a further N messages
  - The elected message is then sent N times, making 3N-1 messages in all
- The turnaround time is also 3N-1, since these messages are sent sequentially

Limitations of the Ring-Based Algorithm

- Useful for understanding the *properties* of election algorithms in general

BUT

the fact it tolerates no failures makes it of limited practical value

- However, with a **reliable failure detector** it is in principle possible to reconstitute the ring when a process crashes

- The bully algorithm [Garcia-Molina, 1982] addresses the problem of process crashes by means of **reliable failure detectors**

Failure Detectors

- Failure detector: service that processes queries about whether a particular process has failed

- Often implemented by an object local to each process (on the same computer), called local failure detector, that runs a distributed failure detection algorithm (in conjunction with its counterparts at other processes)

- A failure detector is not necessarily accurate (asynchronous systems)

- Two classes of failure detectors: **unreliable** and **reliable**

- Most fall into the category of *unreliable failure detectors*

Unreliable Failure Detector

- May produce one of two values when given the identity of a process: Unsuspected or Suspected
- Both of these results are **hints**, *which may or may not accurately reflect whether the process has actually failed*
- Unsuspected: the detector has recently received evidence suggesting that the process has not failed (example: a msg was recently received from it)
  - But of course the process can have failed since then!
- Suspected: the failure detection has some indication that the process ***may*** have failed (example: message not received or received *late*)
  - **The suspicion may be misplaced!** (Example: the process could be functioning correctly, but on the other side of a network partition; or it could be running more slowly than expected)

[Unreliable Failure Detector] Possible Algorithm

- D secs: estimate of the maximum msgs transmission

- Every T secs, each process p sends a "p is here" msg to every other process

  IF the local failure detector at process q does not receive a "p is here" msg within T + D secs of the last one

  THEN it reports to q that p is Suspected

- However, IF it subsequently receives a "p is here" message, THEN it reports to q that p is Unsuspected

What About T and D?

- In a real distr. system, there are practical limits on msg transmission times
- If we choose small values for T and D (total 0.1 sec, say): failure detector may suspect non-crashed process (*inaccurate* failure detector)
- If we choose a large total timeout value (a week, say): crashed processes will be often reported as Unsuspected (*incomplete* failure detector)
- Solution: adaptive timeouts, reflecting the observed network delay conditions
  - Example: if a local failure detector receives a "p is here" in 20 secs instead of the expected maximum of 10 secs, then it could reset its timeout value for p accordingly
- The failure detector remains unreliable (only hints!), but the probability of its accuracy increases

Reliable Failure Detector

- Always accurate in detecting a process's failure
- It always processes' queries with either a response of Unsuspected (a *hint* as before) or Failed
- Failed: means that the detector has determined that the process has crashed
- Reliable failure detectors require that the system is synchronous!

The Bully Algorithm [Garcia-Molina, 1982]

- It allows processes to **crash** during an election
- It assumes that message delivery between processes is **reliable**
- Unlike the ring-based algorithm, it assumes that the system is **synchronous**
- It assumes that each process knows which processes have higher identifiers and that it can communicate with all such processes

N.B.: the ring-based algorithm assumed that processes have a minimal *a priori* knowledge of one another: each knows only how to communicate with its neighbour, and *none knows the identifiers of the other processes*

[Bully Algorithm] Types of Messages

- Three type of messages in the algorithm:
  - *election*: sent to announce an election
  - *answer*: sent in response to an election message
  - *coordinator*: sent to announce the identity of the elected process (new coordinator)
- A process begins an election when it notices, through timeouts, that the coordinator has failed
- *Several processes may discover this concurrently!*

[Bully Algorithm] Reliable Failure Detector

- Since the system is synchronous, we can construct a reliable failure detector
- Ttrans = maximum transmission delay
- Tprocess = maximum delay for processing a message
- T = 2Ttrans + Tprocess: upper bound on the total elapsed time from sending a message to another process to receiving a response
- If no response arrives within time T, then the local failure detector can report that the intended recipient of the request has failed

Bully Algorithm - Part 1

- The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a *coordinator* message to all processes
- A process with a lower identifier begins an election by sending en *election* message to those processes that have a higher identifier
- Then it awaits an *answer* message in response

IF none arrives within time T

THEN the process considers itself the coordinator and sends a *coordinator* message to all the processes with lower identifiers announcing this

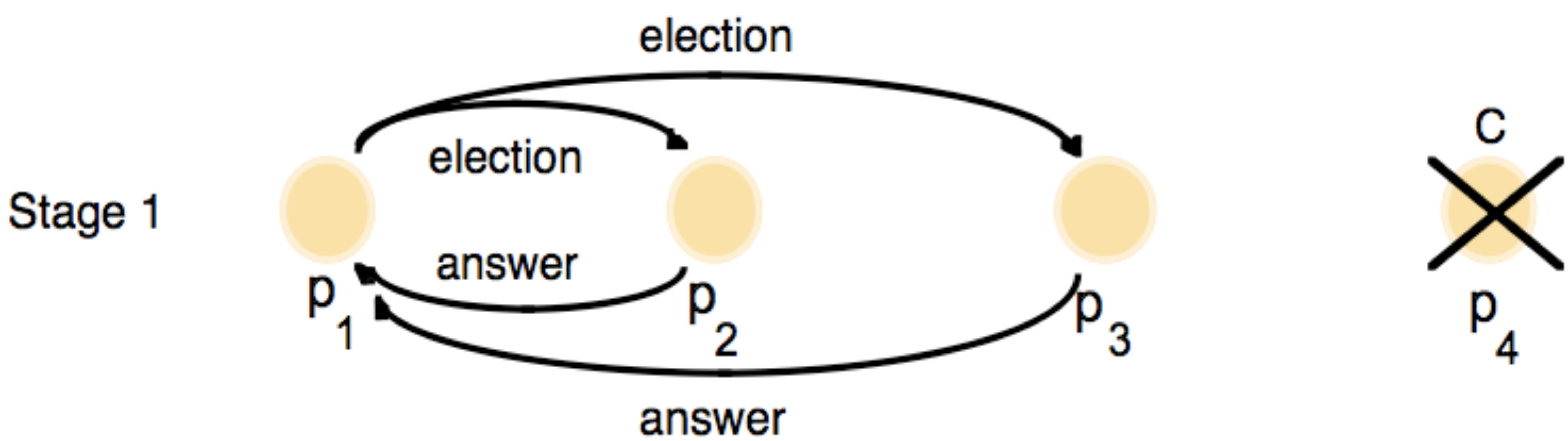
ELSE the process waits a further period T for a *coordinator* message to arrive from the new coordinator
If none arrives, it begins another election

Bully Algorithm - Part 2

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

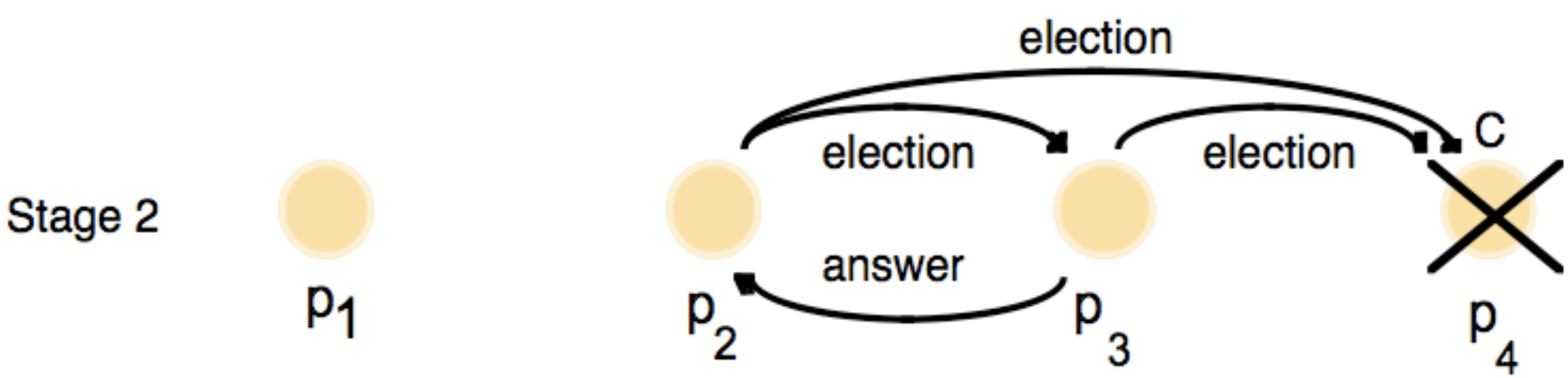
[Bully Algorithm] Example

- Four processes p_1, p_2, p_3 and p_4 (coordinator)
- Process p_1 detects the failure of the coordinator p_4, and starts an election
- On receiving an *election* message from p_1, processes, p_2 and p_3 send *answer* messages to p_1



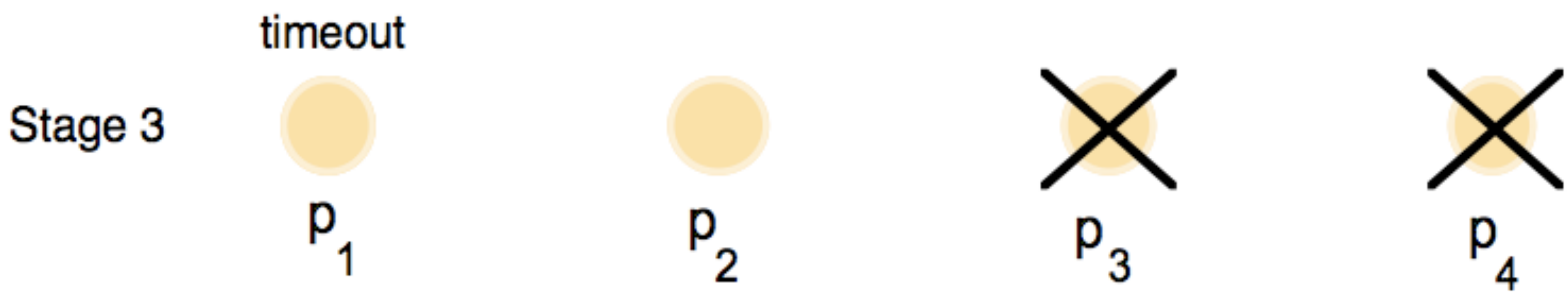
[Bully Algorithm] Example

- Consequently, p_2 and p_3 begin their own elections
- p_3 sends an *answer* message to p_2
- But p_3 receives no *answer* message from the failed process p_4

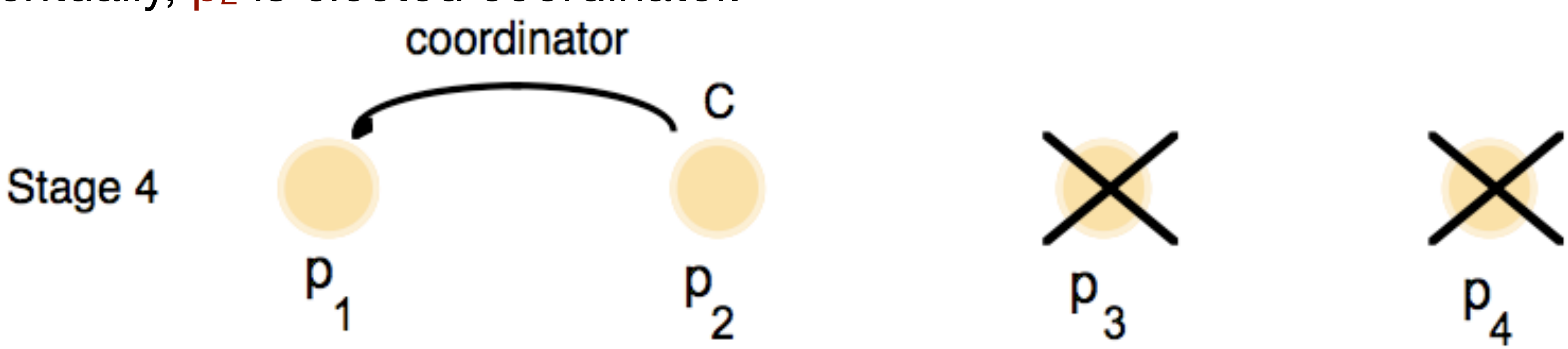


[Bully Algorithm] Example

- p_3 therefore decides that it is the coordinator
- But before it can send out the *coordinator* message, it too fails
- When p_1's timeout period T expires (which we assume occurs before p_2's timeout expires), it deduces the absence of a *coordinator* message and begins another election



- Eventually, p_2 is elected coordinator.



Why "Bully"?

- 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!

If a process receives a *coordinator* message from a process with a lower identifier, it immediately initiates a new election. This is how the algorithm gets its name: ***a process with a higher identifier will bully a lower identifier process out of the coordinator position as soon as it comes online***.

[Bully Algorithm] Performance Analysis

- In the 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
  - Turnaround time is 1 message transmission time: *coordinator*
- In the worst case, the algorithm requires $O(N^2)$ messages
  - The process with the least identifier first detects the coordinator's failure
  - for then N-1 processes altogether begin elections, each sending messages to processes with higher identifiers
  - Turnaround time is approx. 5 message transmission times if there are no failures during the run: *election*, *answer*, *election*, *answer*, *coordinator*