DAD
Desenvolvimento de Aplicações Distribuídas

Coordination:
Mutual Exclusion, Leader Election

Summary

- **Coordination/Consensus**
  - Problems in distributed systems
  - Failure detection

- **Algorithms:**
  - Mutual Exclusion
    - centralized server, ring, Ricart and Agrawala, Maekawa
  - Election
    - ring, bully

Need for Coordination Algorithms

- **For a set of processes:**
  - To coordinate their actions or to agree on one or more values
- **Examples:**
  - Several computers in an airplane, spaceship, other complex equipment and distributed systems in general
- **Such coordination should be done even without a master-slave relation:**
  - Such solution has a single point of failure
- **Multicast is a useful communication paradigm:**
  - Is basically a problem of agreement between processes
- **These are hard problems:**
  - they are even more dificult when considering failures

Coordination Problems in Distributed Systems (1)

- **asynchronous distributed systems:**
  - no single process has a view of the current global system state
- **need to coordinate the actions of the independent processes to achieve common goals:**
  - **failure detection**: how do I know in an asynchronous network whether my peer is dead or alive?
  - **mutual exclusion**: no two process will ever get access to a shared resource in a critical section at the same time
  - **election**: in master-slave systems, how will the system elect a master (either at boot up time or when the master fails)?

Coordination Problems in Distributed Systems (2)

- **need to coordinate the actions of the independent processes to achieve common goals:**
  - **multicast**: sending to a group of recipients
    - reliability of multicast (correct delivery, only once, etc.)
    - order preservation
  - **consensus** in the presence of faults:
    - how to know whether acknowledgement was received over an unreliable communication medium?
    - how to agree on whether a transaction that is manipulating data on a set of distributed databases can be globally committed:
      - all databases agree that the transaction has accessed valid data (isolation)
      - no database crashes during the process (atomicity)?

Failure Detector

- **service that possesses the capability to decide whether a particular process has crashed or not**

- **local failure detector in each object, collaborating with peers in other processes to detect failure:**
  - distinguishes suspected and unsuspected peer processes
  - **reliable** failure detector: → always accurate in detecting a process's failure
    - unsuspected:
      - may have already crashed…
      - but eventually all faulty process have to be reported as faulty (completeness)
    - failed:
      - accurate determination that peer process has failed
        - no false positives:
        - no slow processes are ever reported as faulty

Unreliable Failure Detector

- **unreliable failure detector:**
  - unsuspected:
    - may be incomplete
      - not suspect an already failed process
  - suspected: only a hint on that peer process may have failed
    - e.g., because no message received in quite some time
    - may be inaccurate
      - e.g., peer process hasn't failed, but the communication link is down, or peer process is much slower than expected

Implementation of Unreliable Failure Detector

- **periodically, every T seconds each p sends "I'm alive" message to every other process**
- **if local failure detector at q does not receive "I'm alive" from p within T+D (D = est. max. transmission delay), then p is suspected**
  - local failure detector at q will revise verdict if message is subsequently received
- **problem:**
  - how to calibrate D
  - either, for small D, intermittent network performance downgrades will lead to suspected nodes, or
  - for large D crashes will remain unobserved (crashed nodes will be fixed before timeout expires)
- **solution approaches:**
  - variable D, based on observed network latencies
- **conclusion:**
  - implementation of reliable failure detectors only possible in synchronous networks

Distributed Mutual Exclusion

- **Algorithms:**
  - Centralized server,
  - Ring,
  - Ricart and Agrawala,
  - Maekawa

Distributed Mutual Exclusion Problems

- **prominent problem in multitasking operating systems**
  - access to shared memory
  - access to shared resources
  - access to shared data
  - various centralized algorithms to ensure mutual exclusion, e.g.
    - Dijkstra's Semaphores
    - Monitors
- **mutual exclusion in distributed systems**
  - no shared memory
  - usually, no centralized instance like operating system kernel that would coordinate access
  - based on a synchronous or asynchronous, usually failure-prone network infrastructure
- **examples**
  - consistent access to shared files (e.g., Network File Systems)
  - coordination of access to an access point in an IEEE 802.11 WLAN

Requirements for Distributed Mutual Exclusion Algorithms in Message-Passing Based Systems

- **Application level protocol to enter a critical section:**
  - enter() – enter critical section, block if necessary
  - resourceAccesses() – access shared resources in critical section
  - exit() – leave critical section, other processes may now enter
- **ME1:**
  - at most one process may execute in the critical section at any given point in time (**safety**)
- **ME2:**
  - requests to enter or exit the critical section will eventually succeed (**liveness**)
- **ME3:**
  - if one request to enter the critical section *happened-before* another, then the entry to the critical section is granted in that order (**fairness, ordering**)

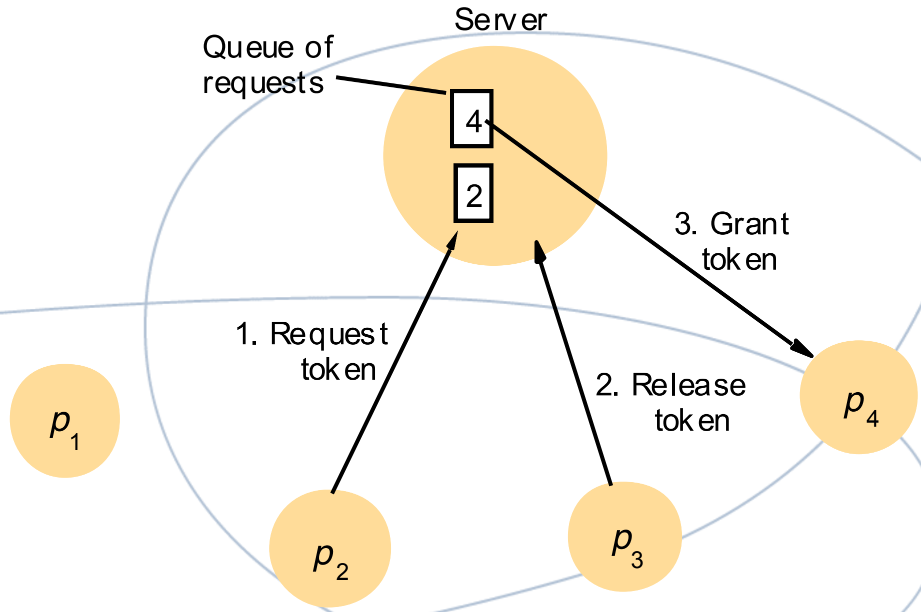
Performance Criteria for Distributed Mutual Exclusion Algorithms

- ***Bandwidth* consumed:**
  - proportional to the number of messages sent in each entry and exit operation
- ***Client delay:***
  - incurred by a process at each entry and exit operation
- **The algorithm's effect upon system *throughput* :**
  - rate at which the collection of processes as a whole can access the critical region, given that some communication is necessary between successive processes:
  - measured in terms of the **synchronization delay** between one process exiting the critical section and the next process entering it

Distributed Mutual Exclusion: Central Server-based Algorithm (1)

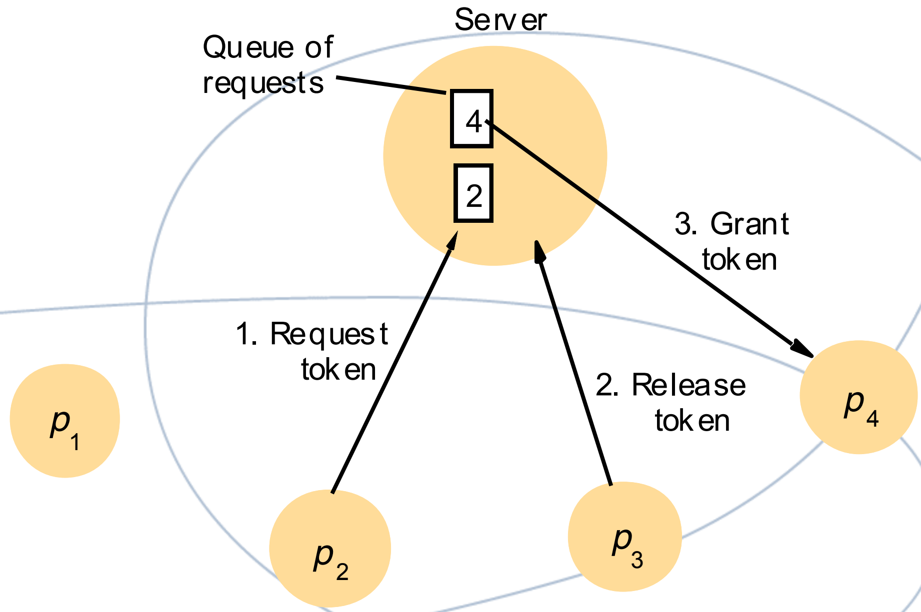
- **central server receives access requests**
  - if no process in critical section, request will be granted
  - if process in critical section, request will be queued

- **process leaving critical section**
  - grant access to next process in queue, or
  - wait for new requests if queue is empty



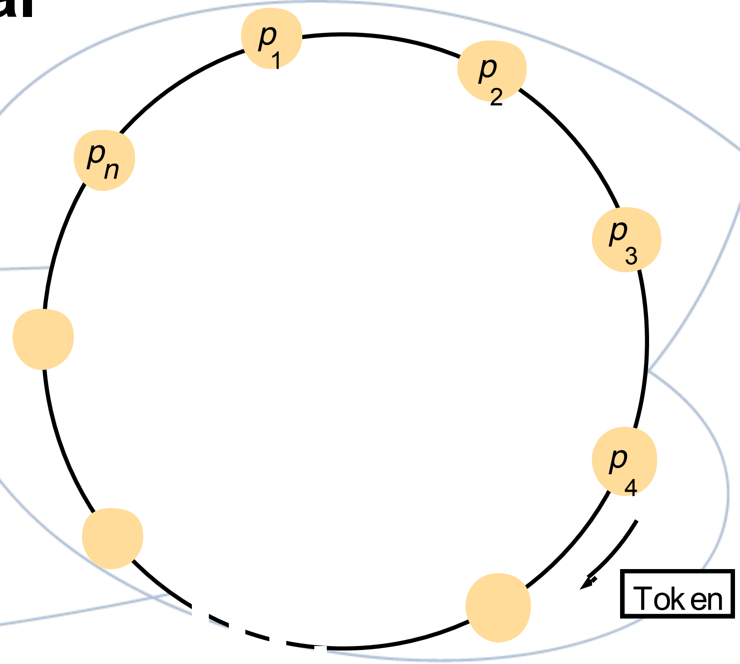
Distributed Mutual Exclusion: Central Server-based Algorithm (2)

- **Properties**
  - satisfies ME1 and ME2, but not ME3 (network delays may reorder requests)
  - entering the critical section takes two messages (delays the requesting process by a round-trip)
  - exiting the critical section takes one release message
  - performance and availability of server are the bottlenecks
  - synchronization delay is the time taken for a round-trip (release + grant)



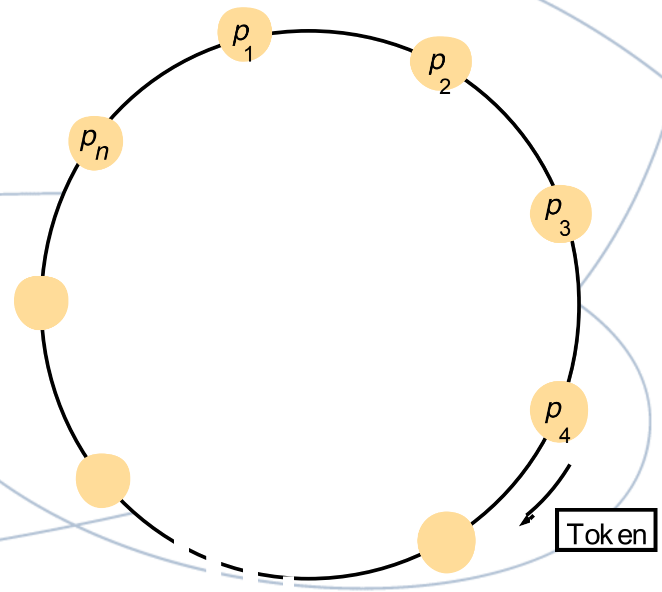
Distributed Mutual Exclusion: Ring Algorithm (1)

- **logical, not necessarily physical link:**
  - every process p_i has connection to process $p_{(i+1)}$ mod N
- **token passes in one direction through the ring**
- **token arrival**
  - only process in possession of token may access critical region
  - if no request upon arrival of token, or when exiting critical region,
    - pass token on to neighbour



Distributed Mutual Exclusion: Ring Algorithm (2)

- **Satisfies:**
  - **ME1 and ME2, but not ME3**
- **performance:**
  - constant bandwidth consumption!!!!
  - entry delay between 0 and N message transmission times (average N/2)
  - synchronization delay (between one process' s exit and the next process' s entry) is anywhere from 1 to N message transmissions
  - No ME3: token order <> request order.



processes may exchange application messages independently of the rotation of the token

Distributed Mutual Exclusion: Fairness and Decentralization

- **Questions:**
  - How to ensure ME3:
    - fairness, ordering, comply with happened-before relation
  - How to avoid single point of failure

- **Approach:**
  - use multicast
  - use logical clocks

Distributed Mutual Exclusion: Ricart and Agrawala Algorithm (1)

```
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;
On receipt of a request <T_i, p_i> at p_j (i ≠ 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;
```

processing of incoming requests deferred just here

- **based on multicast**
  - process requesting access multicasts request to **all** other processes
  - process may only enter critical section if **all** other processes return positive acknowledgement messages
- **assumptions:**
  - all processes have communication channels to all other processes
  - all processes have distinct numeric ID and maintain logical (Lamport) clocks with process IDs

Distributed Mutual Exclusion: Ricart and Agrawala Algorithm (2)

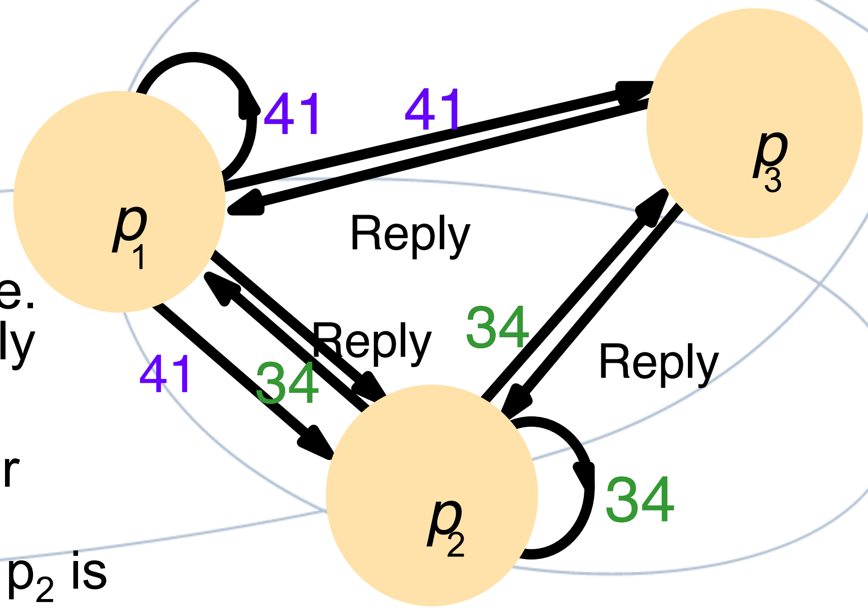
```
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast request to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (N – 1));
    state := HELD;
On receipt of a request <Ti, pi> at pj (i ≠ j)
    if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
        queue request from pi without replying;
    else
        reply immediately to pi;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

processing of incoming requests deferred just here

- if request is broadcast and state of all other processes is RELEASED, then all processes will reply immediately and requester will obtain entry
- if at least one process is in state HELD, that process will not reply until it left critical section
- if two or more processes request at the same time, whichever process' request bears lower timestamp will be the first to get N-1 replies (respects *happens-before* order – logical clock)
- in case of equal timestamps, process with lower ID wins

Distributed Mutual Exclusion: Ricart and Agrawala Algorithm (3)

- p_3 not attempting to enter, p_1 and p_2 request entry simultaneously
- p_3 replies immediately
- p_2 receives request from p_1, timestamp(p_2) < timestamp(p_1), i.e. 34<41, therefore p_2 does not reply
- p_1 sees its timestamp to be larger than that of the request from p_2, hence it replies immediately and p_2 is granted access
- p_2 will reply to p_1's request after exiting the critical section



Distributed Mutual Exclusion: Ricart and Agrawala Algorithm (4)

- **algorithm satisfies ME1**
  - two processes p_i and p_j can only access critical section at the same time in case they would have replied to each other
  - since pairs $<T_i, p_i>$ are totally ordered, this cannot happen
- **algorithm also satisfies ME2 and ME3**
- **Performance:**
  - getting access requires 2(N-1) messages per request: N-1 for multicast (can be optimized as single multicast), and N-1 for replies
  - synchronization delay: just one message transmission time (previous algorithms: from round-trip up to N)
- **protocol improvements:**
  - repeated entry of same process without executing protocol

Distributed Mutual Exclusion: Maekawa's Algorithm (1)

- **Observation:**
  - to get access, **not all processes have to agree**
  - suffices to split set of processes up into subsets ("voting sets ") that overlap
  - suffices that there is consensus within every subset
- **Model:**
  - processes p_1, .., p_N
  - associate a voting set V_i with each process
  - voting sets V_1, .., V_N chosen s.t. ∀ i,k and for some integer M:
    - $p_i \in V_i$
    - $V_i \cap V_k \neq \emptyset$ (there is at least one common member of any two voting sets)
    - $| V_i | = K$ (fairness: each process has a voting set of the same size)
    - each process p_j, is contained in M of the voting sets V_i

Distributed Mutual Exclusion: Maekawa's Algorithm (2)

- **to obtain entry to critical section:**
  - p_i sends request messages to all K members of voting set V_i including itself
- **p_i cannot enter until it has received all K replies**
- **when receiving request**
  - if state = HELD or already replied (voted) since last request
    - then queue request (in the order of its arrival)
  - else immediately send reply
- **when leaving critical section:**
  - send release to all members of V_i
- **when receiving release**
  - remove request at head of queue and sends a reply message (a vote) in response to it (*ordering but not necessarily HB*)

Distributed Mutual Exclusion: Maekawa's Algorithm (3)

```
On initialization
    state := RELEASED;  voted := FALSE;
For pi to enter the critical section
    state := WANTED;
    Multicast request to all processes in Vi          ;
    Wait until (number of replies received = (K     ));
    state := HELD;
On receipt of a request from pi at pj
    if (state = HELD or voted = TRUE)
    then
        queue request from pi without replying;
    else
        send reply to pi;
        voted := TRUE;
    end if
For pi to exit the critical section
    state := RELEASED;
    Multicast release to all processes in Vi          ;
On receipt of a release from pi at pj
    if (queue of requests is non-empty)
    then
        remove head of queue - from pk, say;
        send reply to pk;
        voted := TRUE;
    else
        voted := FALSE;
    end if
```

Distributed Mutual Exclusion: Maekawa's Algorithm (4)

- **The algorithm respects ME1:**
  - If it were possible for two processes p_i and p_j to enter the critical section at the same time, then the processes in $V_i \cap V_j \neq \emptyset$ would have to have voted for both
  - But the algorithm allows a process to make at most one vote between successive receipts of a release message
  - So, such situation is impossible
- **However, the algorithm is deadlock-prone (ME2 not ensured):**
  - Consider p_1, p_2, p_3
  - $V_1 = \{p_1, p_2\}$; $V_2 = \{p_2,p_3\}$; $V_3 = \{p_3,p_1\}$
  - If the three processes requests entry to the critical section, it is possible that:
    - p_1 replies to itself and holds off p2
    - p_2 replies to itself and holds off p3
    - p_3 replies to itself and holds off p1
  - Each process has received one out of two replies, and none can proceed
- **algorithm can be modified to ensure absence of deadlocks using logical clocks – ensures ME2 and ME3 (HB)**

Distributed Mutual Exclusion: Maekawa's Algorithm (5)

- **Performance:**
  - Bandwidth consumption:
    - Entry: 2 * SQRT (N), plus
      - ~quorum size: 1 request and 1 reply per each member
    - Exit: SQRT (N)
      - 1 release per each quorum member
  - Client delay: ~ 1 round-trip
  - Synchronization delay:
    - 1 round-trip
      - instead of single message in Ricart and Agrawala
    - Why?:
      - 1 release message to (all) quorum of exiting node
      - that triggers on reception
      - 1 reply message from (at least one) node of quorum of waiting node.

Notes on Fault Tolerance

- **none of these algorithms tolerates message loss**
- **ring-algorithms cannot tolerate single crash failure**
- **Maekawa's algorithm can tolerate some crash failures:**
  - if a crashed process is not in a voting set that is required, its failure does not affect the rest of the system
- **Central-Server:**
  - tolerates crash failure of node that has neither requested access nor is currently in the critical section
- **Ricart and Agrawala algorithm can be modified to tolerate crash failures by the assumption that a failed process grants all requests immediately:**
  - requires reliable failure detector

Election Algorithms

- **Algorithms**
  - Ring
  - Bully

Election Algorithms

- **Algorithms designed to:**
  - *designate one unique process out of a set of processes with similar capabilities to take over certain functions in a distributed system*
  - **central server** for mutual exclusion
  - **ring master** in token ring networks
  - **bus master**
- **necessary when**
  - system is booted
  - server fails
  - server retires

Election Algorithms (2)

- **Properties: to be valid during any particular run of the system**
  - **E1**: a process p_i has $elected_i = \perp$ (undefined) or $elected_i = P$ for some non-crashed process P that will be chosen at the end of the run with the largest *identifier* **(safety)**
    - *e.g., identifiers: process IDs, CPU or memory availability, etc., any unique ordered value*
  - **E2**: all processes p_i will eventually set $elected_i \neq \perp$ **(liveness)**
- **Performance**
  - **network bandwidth** utilization (proportional to total number of messages sent)
  - **turnaround time**: number of serialized message transmission times between initiation and termination of a single run

Ring-based Algorithm (1)

- **Assumptions:**
  - all nodes communicate on uni-directional ring structure
  - all processes have unique integer id
  - asynchronous, reliable system
- **Initially:**
  - all processes marked "non-participant"
- **To begin election:**
  - process places election message with **own** identifier on ring and marks itself "participant"

Ring-based Algorithm (2)

- **Upon receipt of election message:**
  - **compare** received identifier with own
  - if received id **greater** than own id, **forward** message to neighbor
  - if received id **smaller** than own id,
    - if own status is “non-participant”, then **substitute own id** in election message and **forward** on ring
    - otherwise, **do not forward** message (already “participant”)
  - if received id is **identical** to own id
    - this process’s id must be **greatest and it becomes elected**
    - marks own status as **“non-participant”**
    - sends out **“elected”** message

Ring-based Algorithm (3)

- **Upon any forwarding:**
  - mark own state as "participant"
- **When receiving "elected" message**
  - mark own status as **"non-participant"**
  - set **elected**$_i$ appropriately and **forward elected** message

Ring-based Algorithm (4)

- **Properties:**
  - E1: a process p_i has $elected_i = \perp$ (undefined) or $elected_i = P$ for some non-crashed process P that will be chosen at the end of the run with the largest identifier (safety)
  - E2: all processes p_i will eventually set $elected_i \neq \perp$ (liveness)
  - E1 satisfied, since all identifiers are compared
  - E2 follows from reliable communication property
- **Failures:**
  - tolerates no failures
  - failed process causes broken ring and algorithm stops.
- **Performance:**
  - bandwidth: up to 3N-1 messages:
    - anti-clockwise neighbour has the highest identifier
  - turnaround time: up to 3N-1, sequential, messages

Bully Algorithm (1)

- **works for <u>synchronous</u> networks**
  - nodes can crash, and crashes will be detected reliably
- **assumptions**
  - each node **knows** identifiers of **all other nodes**
  - every node **can communicate** with **every other node**
- **message types**
  - **election**: announce an election
  - **answer:** reply to an election message
  - **coordinator**: announce identity of elected process

Bully Algorithm (2)

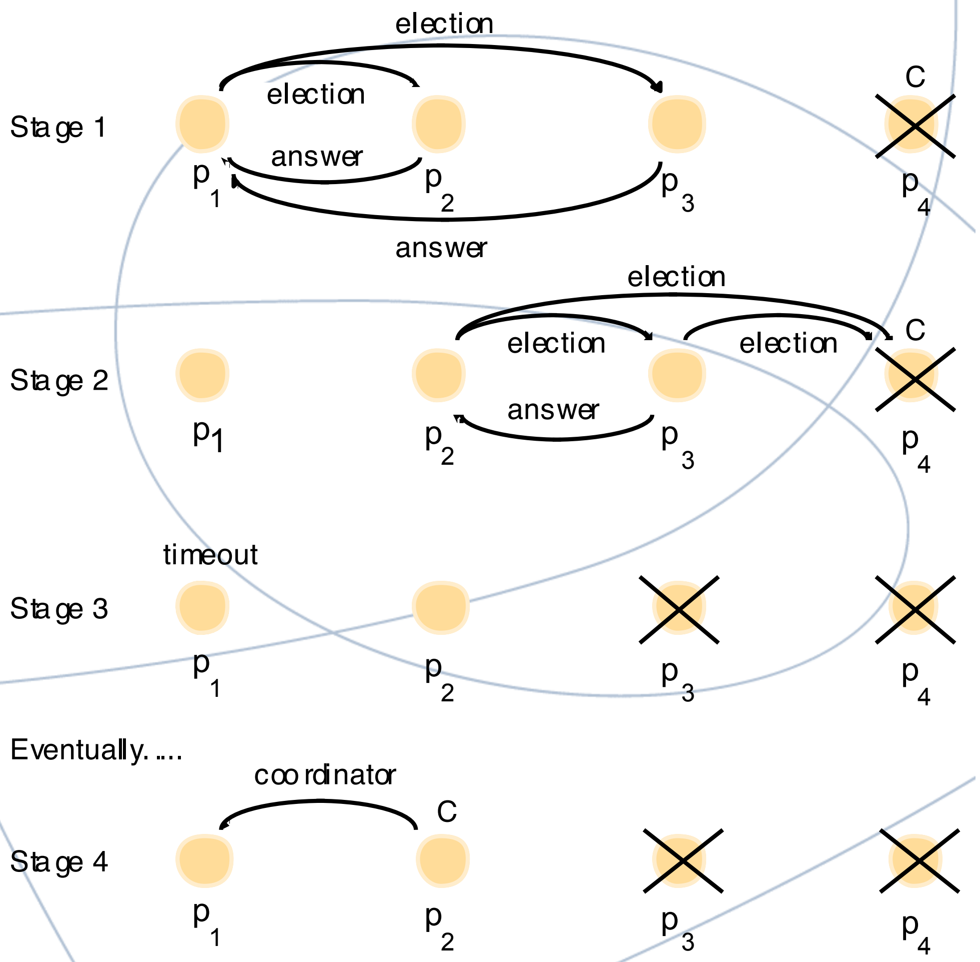
- **Initiation of algorithm: reliable failure detection**
  - a peer process failed if no answer to request within
    - $T = 2T_{trans} + T_{process}$
- **process can decide whether:**
  - to **<u>become coordinator</u>** by comparing own id with all other ids (highest wins)
  - announce by **sending <u>coordinator</u>** message to all other nodes with **lower id**
- **process with lower id can:**
  - **<u>bid to become coordinator</u>** by sending **<u>election</u>** message to all processes with higher ID
  - if **no response** within T, **considers itself elected** coordinator, **sends <u>coordinator</u>** message to all processes with **lower id**
  - otherwise, **wait** for another T' time units **for a <u>coordinator</u> message** to arrive from new coordinator
    - if no response, then begin another <u>election</u> process

Bully Algorithm (3)

- **When process receives election message:**
  - sends back an **answer** message and
  - begins another election - unless one was already initiated
- **When process receives coordinator message:**
  - sets variable $election_i$ equal to:
    - the id of the coordinator received in the election message
- **New process replacing crashed process:**
  - if highest id, will immediately send coordinator message and "***bully***" current coordinator to resign

Bully Algorithm (4)

- Assumes that the system is synchronous
- Uses timeouts to detect process failures
- p_1 detects the failure of the coordinator p4 and announces an election **(stage 1)**
- On receiving an election message from p_1, p_2 and p_3 send answer messages to p_1 and begin their own elections
- p_3 sends an answer message to p_2, but p_3 receives no message from p_4 **(stage 2)**
- p_3 decides that it is the coordinator; but before sending out the coordinator message, it fails too **(stage 3)**
- When p_1 timeout T' expires (we assume before p_2 timeout expires), p_1 deduces the absence of a coordinator message and starts another election
- Eventually, p_2 is elected coordinator **(stage 4)**



Bully Algorithm (5)

- **Properties:**
  - E1: a process p_i has $elected_i = \perp$ (undefined) or $elected_i = P$ for some non-crashed process P that will be chosen at the end of the run with the largest identifier (safety)
  - E2: all processes p_i will eventually set $elected_i \neq \perp$ (liveness)
  - E1 satisfied (if no process replaced and timeout T estimate accurate)
  - E2 satisfied (synchronous network, reliable transmission)
  - E1 not satisfied if crashed process replaced at the same time while another process has announced that it is the new coordinator
    - or if timeout values are innacurate (unreliable failure detection)

Bully Algorithm (6)

- **Performance:**
  - bandwidth:
    - from N-2...
      - process with highest ID detects failure, triggering election
    - ...up to N^2
      - process with lowest ID detects failure, triggering elections
  - turnaround time:
    - from 1...
      - process with highest ID detects failure, triggering election
    - ...up to to 2N
      - process with lowest ID detects failure, triggering elections

Bully Algorithm (7)

- **Algorithm complexity**
  - due to using same algorithm to address:
    - coordination/election issues
    - fault detection of nodes
- **Assuming reliable failure detector available:**
  - simpler implementation and election
    - every process knows other correct (un-failed) processes
    - every process knows process with highest ID
  - **drawback**: fault detection implies sending messages to/from all nodes
  - **advantage**: bully checks only failures from processes with higher IDs
    - fewer messages required