Contents

1. Reminder of Lectures So Far - Time & Mutual Exclusion

2. Co-ordination of Processes in Distributed Systems

3. Leader Election
Time & Mutual Exclusion

- Distributed Clocks & Importance of Accurate Time in Distributed Systems - Clock skew and Clock drift

- Physical Clock Synchronisation - External and Internal Clock Synch. Algorithms

- Causal Ordering in Distributed Processes, Logical Clocks, Vector Clocks

- Distributed Mutual Exclusion - Principled Ordering of Distributed Processes Wanting to enter CS
• Distributed Mutual Exclusion - Safety, Liveness, Performance, Correctness

• Distributed Mutual Exclusion - Central Server, Ring-Based, Ricart & Agrawala, Maekawa
Co-ordination of Processes in Distributed Systems

- Synch. & Asynch. Distributed Systems (speed & time guarantees in one, none in the other)

- Failure Types & Failure Models

- Problem - Provide Protocols for Principled Co-ordination of Processes in Distributed Systems

- Solutions include: Causal Ordering, Distributed Mutual Exclusion, Leader Election
Leader Election - Motivations

- Client-Server model of resource management

- Capability is a token given to a client by a server to perform a service

- Server is the leader, Clients send requests to Server, Requests granted or denied based on the token or the client

- Some examples: Capability-based Addressing, Capability-based Security, Capability-Based Computer Systems*, Access Control List, etc.

*https://homes.cs.washington.edu/~levy/capabook/
Leader Election - Motivations

- A centralised database manager - shared data updated by clients

- Manager maintains queue of pending reads and writes
  - Manager is the leader

- Token Ring Networks - Token circulates round the network

- Current holder (current leader) of token can do some work e.g. initiate communication, etc.
• Sometimes token may be lost or current token holder may fail

• Regeneration of Token - Leader Election
Leader Election - Informal Specification

Assumption: Our system is synchronous, using broadcast comm. and it’s reliable.

**Election Problem.** A set of processes $P_1, P_2, \ldots, P_n$ must select a leader. Each process $P_i$ has a unique identifier $uid(i)$. Devise a protocol s.t. all the processes learn the identity of the leader.
Leader Election - Formal Specification

Let $G = (V, E)$ represent our system topology, with $V$ being the vertices (or nodes/processes) of our system, and $E$ is the set of edges between the nodes. Each process $i \in V$ has a unique identifier, and a variable $L(i)$ identifies the leader. Also, let $ok(i)$ denote the predicate that $i$ is non-faulty. Then, the ffg. 3 conditions must hold:

- $\forall i, j \in V : ok(i) \land ok(j) \Rightarrow L(i) = L(j)$
- $L(i) \in V$
- $ok(L(i)) = true$
Leader Election vs Mutual Exclusion

Similarities btw L.E. & M.E. ?
Leader Election vs Mutual Exclusion

Similarities btw L.E. & M.E.? 

One answer: whichever process enters the critical section becomes the leader
Leader Election vs Mutual Exclusion

Differences btw L.E. & M.E. ?
Leader Election vs Mutual Exclusion

Differences btw L.E. & M.E.?

• Failure is not inherent in Mutual Exclusion - Failure in CS is ruled out

• Starvation is irrelevant in Leader Election. Processes need not take turns to be leader. System can function happily if current leader doesn't fail

• If you view L.E. as a mutual exclusion problem, then exit from CS is un-necessary
• But, leader needs to inform active processes of its identity. A non-issue in M.E.
Leader Election - Example Token Ring Network
Leader Election - Correctness

- **Safety**: A participant process $p_i$ has $elected_i = \bot$ or $elected_i = P$ here $P$ is a non-faulty process with the largest identifier.

- **Liveness**: All processes participate and eventually set $elected_i \neq \bot$, or fail (i.e. crash).
Leader Election - Performance

- bandwidth consumption: total number of messages sent

- turnaround time: number of serialised message transmissions in a single run
Leader Election Mechanism - Ring-Based Algorithm * - Informal Description

Each process, in the network, has a unique identifier (UID), chosen from some totally ordered space of identifiers, and each process's UID differs from the others in the network. Each process sends its UID around the ring, When a process receives an incoming UID, it compares that UID to its own. If the incoming UID is greater than its own, it keeps passing the identifier; if it is less than its own, it discards the incoming UID; if it is equal to its own, the process declares itself the leader.

Ring-Based Algorithm

Assumptions

• processes \( p_1 \ldots p_n \) arranged in a logical ring

• \( p_i \) sends messages to \( p(i \mod n) + 1 \)

• no failures

• asynchronous system
Ring-Based Algorithm

- Initially, every process is a non-participant

- Any process can call an election:
  1. Marks itself as participant
  2. Places its id in an election message
  3. Sends the message to its neighbour

- Upon receiving an election message:
  1. If \( id > myid \), forward the msg, mark participant
2. if $id < myid$
   
   - non-participant: replace $id$ with $myid$ in msg, forward the msg, mark participant

   - participant: stop forwarding (multiple elections)

3. if $id = myid$, coordinator found, $elected_i := myid$, send $elected$ message with $myid$, mark non-participant

   • upon receiving an elected message:

   1. if $id \neq myid$, $elected_i := id$, forward the msg, mark non-participant

   2. if $id = myid$, stop forwarding
Ring-Based Algorithm - Correctness

- **Safety**: yes - only the process with the largest id can send an *elected* message

- **Liveness**: yes - every process in the ring eventually participates in the election; extra elections are stopped
Ring-Based Algorithm - Performance

- one election, best case (initiating process has highest identifier)

  1. $N$ election messages, $N$ elected messages
  2. turnaround: $2N$ messages

- one election, worst case (anti-clockwise neighbour has highest identifier)

  1. $2N - 1$ election messages, $N$ elected messages
  2. turnaround: $3N - 1$ messages
• works if more than one process starts an election

• does not tolerate failures!
Ring-Based Algorithm - 2 Qs to Ponder

- Can we solve the Election problem if the UIDs are identical?

- Can we solve the Election problem if processes do not have UIDs or UIDs are not used
Ring-Based Algorithm - Possible Answers

- Election in Anonymous Networks
- Breaking Symmetry - Randomization
- Another Question - Any problem with randomization?
Ring-Based Algorithm - Possible Answers

- Breaking Symmetry - Randomization

- Another Question - Any problem with randomization?

- Yes! Correctness or Performance is only probabilistic (game of chances - No certainty!)

- Side note: (java.Random is pseudo-random not totally random). Why?
Bully Algorithm

Assumptions

- processes can crash
- message delivery reliable
- synchronous system, use time-outs to detect process crashes ($T_{timeout} = 2T_{transmitting} + T_{processing}$)
- each process knows all other processes and can communicate with them
• each process knows which processes have higher identifiers

• the process with largest identifier out of the non-failing ones must be elected

• messages: election, answer, coordinator
Bully Algorithm - Informal Spec.

- **Step 1**: Any process, after detecting the failure of the leader, bids for being the new leader by sending an *election* message to every process with a higher identifier.

- **Step 2**: If any process with a higher id responds with a *reply* message, then the requesting process gives up its bid for becoming the leader. Subsequently, it waits to receive a *leader* message (*I am the leader*) from some process with a higher identifier.
• **Step 3**: If no higher-numbered process responds to the *election* message sent by node *i* within a time-out period, then node *i* elects itself as the leader and sends a leader message to every process in the system.

• **Step 4**: If no *leader* message is received by process *i* within a time-out period after receiving a reply to its *election* message, then process *i* suspects that the winner of the election failed in the mean time and re-initiates the election.
Bully Algorithm

Program bully // program for process i

failed: Boolean // set if the failure of the leader is detected

L: process // identifies the leader

m: message // election | leader | reply

state : idle | wait for reply | wait for leader

initially forall i in V. state = idle, failed = false
do
IF
failure of L(i) detected && failed:= false
THEN failed := true
ENDIF

IF failed
THEN forall j > i
send election to j;
state := wait for reply;
failed := false
ENDIF

IF
(state = idle) && (m = election)
THEN
send reply to sender;
failed := true
ENDIF

IF
(state = wait for reply) && (m = reply)
THEN state := wait for leader
ENDIF

IF (state = wait for reply) && timeout
THEN
L(i) := i;
state := idle;
send leader to all
ENDIF

IF (m = leader)
THEN
L(i) := sender;
state := idle
ENDIF

ENDIF

IF (state = wait for leader) && timeout
THEN
failed := true;
state := idle
ENDIF

od
Bully Algorithm - Correctness

- **Safety:**

  1. a lower-id process always yields to a higher-id one

  2. however, during an election, if a failed process is replaced

      – the low-id processes might have two different coordinators: the newly elected coordinator and the new process (see 2 slides down)!

  3. failure detection might be unreliable!
• **Liveness**: all processes participate and know the coordinator at the end
Bully Algorithm - Safety Violation Example

- assume $p_3$ failed, then replaced (or $p_3$ running very slowly and incorrect time-out values are used)

- while $p_2$ sends coordinator message, so does $p_3$

- $p_1$ may conclude $p_2$ is the coordinator, while $p_2$ and $p_3$ conclude $p_3$ is the coordinator
Bully Algorithm - Performance

- **best case** (#2 process detects failure of coordinator and elects itself):
  
  1. overhead: \( N - 2 \) coordinator messages
  
  2. turnaround delay: one message (no election/answer messages)
Bully Algorithm - Performance

- **worst case** (process with least identifier detects coordinator failure):

  1. overhead:
    - $1 + 2 + \ldots + (N - 2) + (N - 2) = (N - 1)(N - 2)/2 + (N - 2)$ election messages
    - $1 + \ldots + (N - 2)$ answer messages
    - $N - 2$ coordinator messages total: $(N - 1)(N - 2) + 2(N - 2) = (N + 1)(N - 2) = O(N^2)$

  2. turnaround delay: delay of election and answer messages
Bully Algorithm - A Question

In the bully algorithm, suppose two processes detect the failure of the coordinator simultaneously and both decide to hold an election. What happens?