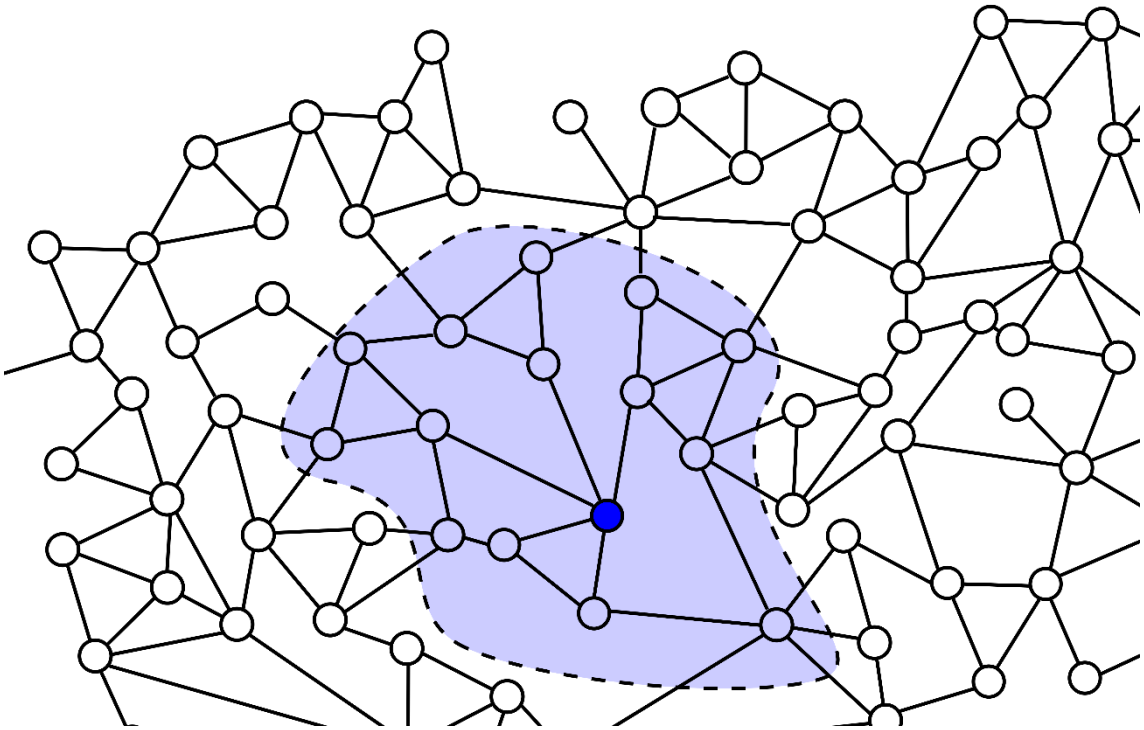
Homework

Robust Distributed Algorithms (Mas ter QDCS M1)

22d October 2025



Teacher: Janna Burman

Herrá Alpuente, Luis Miguel: luis-miguel.herra-alpuente@universite-paris-saclay.fr

1.Consider the basic LCR solution to leader election seen in class (see L eader Election slides 5 - 7). Prove that this algorithm terminates and th at it is also correct at termination (the definitions of these terms can b e found in the Model slides of the course).

Algorithm summary

Let the system be a directed ring of n processes P={p1,...,pn}, where each process pi holds a unique identifier idi∈N.
Initially, each process sends a message containing its identifier to its right neighbour (direction of the ring).
Upon receiving a message containing identifier m:

- if m>idi: forward m to the right;
- if m<idi: discard it;
- if m=idi: declare itself the leader.

Let M=max{idi|pi∈P}and pm its corresponding process.

Termination

At any time t, each message in transit corresponds to an identifier that has not yet been discarded; as the discarded ones would never be transmited.
If a message with identifier m<M meets the message with M, it is discarded upon comparison, since every process that receives both will only forward the greater identifier.
Hence, the message with M cannot be removed from the ring: it is the unique maximal element under the forwarding rule.Because the communication graph is a finite directed ring, M will traverse at most n edges before returning to pM. Upon receiving its own identifier, pM declares itself leader, and no further messages are sent.
Therefore, the algorithm terminates in a finite number of steps, bounded by n. In other words, idM is transmited from pM to all processes, if I have received my id back, that means no id is greater than mine, I am the leader. As the id of each process is unique, and the operations will not modify this values, we can affirm that there will never be multiple/no decision leader elections.

By contradiction: Let **L** be the identifier of the process that declares itself leader. By definition, a process only declares leadership when it receives a message carrying its own identifier. Suppose **L≠M**.
Since M>L, the message carrying M must also circulate in the ring and, by the forwarding rule, eliminate any smaller identifier it encounters, including L.
Hence, the message with L cannot complete a full traversal, contradicting the fact that its owner received it back. Because all identifiers are unique, there exists exactly one process with identifier M; thus, only one process can declare leadership. Therefore L=M.

2.Assume an arbitrary communication topology and synchronous model. Propose a solution to the leader election problem; satisfying also the three variants appeared in slide 3 (Leader Election slides). Give a short textual explanation of the algorithm and provide also the complete pseudocode (states, starts, msgs(), trans()). Asume that an upper bound on the network diameter Diam is given to the processes (diameter is the longest among all the shortest paths in the given network). Explain why the latter assumption is needed.

Requested Algorithm summary

We must design a leader election algorithm for arbitrary network topologies in a synchronous model, satisfying three requirements: non-leaders learn their status, all processes learn the leader's ID, and all processes detect termination. We are given an upper bound on the network diameter, Diam.

The algorithms presented in the leader election section assume a ring-oriented topology, whereas we have an arbitrary network. We must adapt one of them for this new context. The best candidates are an LCR extension or Flood-Set.

We choose the LCR extension approach because it has less bit complexity. While Flood-Set sends sets of IDs (requiring O(n × log n) bits per message), an LCR extension sends only single IDs per message (O(log n) bits). Both approaches require similar message counts: O(Diam × |E|), but the LCR approach transmits significantly fewer total bits: O(Diam × |E| × log n) versus Flood-Set's O(Diam × |E| × n × log n).

The adaptation is easy: instead of sending IDs only to the next neighbor in a ring, each process broadcasts to all its neighbors. The main LCR principle remains, forward only IDs larger than your own, and the maximum ID eventually reaches all nodes. After Diam rounds, complete information propagation is guaranteed: any ID reaches any node via its shortest path of at most Diam hops.

Motivation for Leader election

- Simplifies coordination (of distributed nodes)
- Important symmetry breaking problem
  - useful for naming, counting, etc.
  - can help break symmetry in the case of deadlock (by removing the elected entity from the deadlock cycle)
- Helps to tolerate failures (e.g., Byz. consensus with leader)
- Can be useful for saving resources (e.g. help building a tree, etc.).

Slide3

Pseudocode

For process p_i:

M = UID ∪ {"elected"}

States + Initial states:

my_id: UID; my_id := the id of i

max_id: UID; max_id := my_id

round: ℕ+; round := 1

status: {leader, non_leader, unknown}; status := unknown

leader_id: UID ∪ {unknown}; leader_id := unknown

send_msg: M ∪ {null}; send_msg := my_id

terminated: boolean; terminated := false

msgs_i (message generation function):

```
If (terminated) then
        return null
Else
        return send_msg to ALL neighbors
EndIf
```

Why is diam upper bound is important?

Diameter is the **longest shortest path** between any two nodes in the network, measured in hops. For example, a complete graph has Diam = 1 regardless of size, while a "line" has Diam = n-1. We need Diam because it provides the **minimum guaranteed time** for complete information propagation: after Diam rounds, any message from any node has definitively reached all other nodes via shortest paths. Without knowing Diam, processes cannot determine when to safely make decisions; some might decide prematurely before seeing all IDs, leading to incorrect elections. Diam provides the essential synchronization where all processes can simultaneously decide, knowing they've seen the global maximum ID. Using n-1 instead would work but be inefficient for well-connected networks where Diam < n, making our O(Diam) algorithm unnecessarily slow.

```
trans i (transition function):
        old_max := max_id
        send_msg := null
// Phase 1: Flooding (rounds 1 to Diam)
        If (round ≤ Diam) then
                For each message m received from neighbors do
                        If (m ∈ UID and m > max_id) then
                                max_id := m
                        EndIf
                EndFor
  // If we learned a larger ID, propagate it
        If (max_id > old_max) then
                send_msg := max_id
        EndIf
 // Phase 2: Decision (round Diam + 1)
 Else If (round = Diam + 1) then
        For each message m received from neighbors do
                If (m ∈ UID and m > max_id) then
                        max_id := m
                EndIf
         EndFor
   leader_id := max_id
   If (max_id = my_id) then
     status := leader
     send_msg := "elected"
   Else
     status := non_leader
   EndIf
 // Phase 3: Announcement propagation (rounds Diam + 2 to 2×Diam + 1)
        Else If (round ≤ 2×Diam + 1) then
                For each message m received from neighbors do
                        If (m = "elected") then
                                send_msg := "elected"  // Relay announcement
                        EndIf
                EndFor
 // Phase 4: Termination
        Else
                terminated := true
        EndIf
round := round + 1
```

3. Consider the Time Slice algorithm (see Leader Election slides 21-24). Propose a slight change to the algorithm to improve its bit complexity (in terms of O). Explain why the changed algorithm is correct and analyze the new bit complexity.

Initially, I considered replacing the transmitted identifiers with a single Boolean flag. The idea was that each process would simply send "true" to indicate that a leader exists, reducing the size of each message to one bit. However, I quickly noticed that this approach would violate the agreement condition, since the other processes would not know which process was the leader; they would only know that a leader had been elected.

Looking back at the original pseudocode, I realized that even after the leader's identifier completes a full circulation, the message generation function continues to send the same identifier indefinitely. To fix this, **the leader should stop sending messages once it receives back its own identifier**. This can be implemented either by modifying the msgs_i() function, which controls message output, or equivalently within the transition function by adding a condition that sets the outgoing message to null when the leader receives its own ID.

msgs_i():

if status = leader and received_msg = my_id then

msg := null # stop sending

else => msg := leader_id or null as before

It could also be approached/seen as:

trans_i (transition function): (same as before but adding one condition)

msg := null

If (status = unknown) then

If (the received message m is non-null, in UID) then => status := non_leader; leader_id := msg := m

Else If (round = (my_id - 1)·n) then => status := leader; leader_id := msg := my_id ; EndIf

Else If (status = leader) then // Leader received its own ID back - stop sending

If (the received message m = my_id) then => msg := null // Stop circulation

EndIf; EndIf; round := round + 1

With this change, the algorithm preserves all correctness properties: validity, agreement, and termination, while improving the communication efficiency. The number of messages remains the same in big O terms, but their total bit count decreases, since the redundant retransmission of the leader's identifier is eliminated. Consequently, the bit complexity becomes closer to O(n) rather than O(nlogn), reducing the overall cost without affecting the algorithm's behavior.

4. Consider the Flood-Set algorithm and assumptions in slide 3 of Consensus slides. Assume that n=4 and f=2. Present and execution scenario of the algorithm where every two (still) alive (not yet crashed) processes have different values in variable W at least in round 1 and 2. Whether these values become equal in round 3? Explain why.

These assumptions are required for the standard Flood-Set reasoning used in the notes:

- **Synchronous rounds.** Communication proceeds in global rounds: every message sent in round r is delivered before round r+1 processing.
- **Reliable point-to-point links.** If a process sends a message to a neighbor in a given round, that neighbor receives it (unless the sender has crashed before sending).
- **Crash-stop failures.** A crashed process stops sending forever (unless a recovery model is explicitly assumed). A crashed process cannot forge messages.
- **Known upper bound f on crashes.** The algorithm and its termination guarantee use the given f.
- **No message loss except due to crash.** (If links can drop messages arbitrarily, Flood-Set guarantees disappear.)

These are the minimal, non-negotiable hypotheses under which Flood-Set and the f+1 rounds correctness statements hold.

Specific Scenario

If both failed processes remain crashed for all three rounds, then the only possible way for the two still-alive processes to have the same window W in round 3 is that they are **direct neighbours** in the communication graph.
When two alive processes share an active link, their windows are exchanged and merged in that round, so by round 3 they both contain the same set of values. If instead they are **not neighbours**, for example, they are located at opposite corners of the square and the two intermediate nodes are crashed: then there is no path between them, and their windows cannot be updated with each other's values. In that case, even in round 3 their W sets remain different.

Flood-Set guarantees convergence after f+r (where r is the number of rounds per node recovery, 1 in our slides) rounds **only when there exists a communication path among all non-faulty nodes**. If the crashes permanently disconnect the surviving nodes, equality of W is impossible regardless of the number of rounds. Now let's analyze different scenarios.

Non crashed process are neighbours

Even if the other two processes remained crashed for the 3 rounds, both processes will have the same window, as they were able to communicate.

Non crashed process not neighbours but every process worked at least once, in previous rounds

Example: A and C alive; B sends its round-1 message then crashes; D behaves the same or at least one of them forwarded something.

Messages from B or D were delivered and possibly forwarded before those nodes crashed, creating a temporal path. It becomes possible that $W_A^3=W_C^3$. Flood-Set guarantees that after f+1 rounds all non-faulty processes have the union of all values that could be forwarded despite up to f crashes, if B/D's sent messages were delivered and forwarded, those values can reach A and C by round 3.

5. Consider 4 processes distributed within a complete network (known all processes), of which at most one (unknown) is Byzantine (n=4, f=1). We consider the following agreement protocol, composed of 2 synchronous rounds (the transmission of messages is synchronous) and the messages are the sets of pairs (id, initial value).

Round 1: each process i sends a pair (message) value v_i (message(i, v_i)), to all the other processes and receives messages from the others (a message (j, v_j) from a process p ¡= j is ignored).

Round 2: each process i relays to all other processes the (j, v_j) pairs received during round 1 (only one pair at most for each j), and receives the pairs from the others (a process ignores a (j, v_j) pair from process j). At the end of this round, process i assigns to process j the strict majority value of the values received for j (in pairs (j,v)) during these two rounds (if such a majority value exists, and otherwise it assigns the smallest majority value in case of tie).

The decided value of process i is the majority values among the values it has assigned (the smallest majority value in case of a tie). We asume that in case of a tie). We asume that a node can receive (and treat) at most one pair (j,v), for a given identifier j, on a link during a round (otherwise it ignores all duplicates). Also, remember that each process knows all its neighbours (in fact, the whole communication graph), and a correct process ignores any message with an unknown identifier in the given solution.

The protocol operates in two synchronous rounds to achieve Byzantine consensus among n = 4 processes with at most f = 1 Byzantine failure. In Round 1, each process broadcasts its identifier paired with its initial value to all others. In Round 2, each process relays all the pairs it received in Round 1 (except its own) to all other processes. After both rounds, each process assigns to every identifier j the majority value among all received pairs for j, then decides on the majority among these assigned values.

Like Reliable Broadcast, it uses message relay to prevent Byzantine processes from equivocating. However, instead of broadcasting a single value with INIT and ECHO phases, all processes simultaneously broadcast their values and relay complete information about all identifiers. This makes the protocol simpler in structure but significantly worse in message complexity. Each process must relay n-1 pairs in Round 2, resulting in $O(n*(n-1) = n^2)$ pairs transmitted per process.

1)Show that at the end of the first round, the majority values that could have been calculated/assigned (from the set of the received value) by correct processes may be different (so such a solution with the decision at the end of the 1st round, may not be correct).

We should demonstrate that correct processes may compute different majority values at the end of Round 1, we can do it by making a single-round solution insufficient.

Consider processes p_1, p_2, p_3 (correct) with initial values $v_1 = 5$, $v_2 = 5$, $v_3 = 7$, and p_4 (Byzantine). The Byzantine process sends (4, 5) to p_1, but (4, 7) to p_2 and p_3. After Round 1, process p_1 receives {5, 5, 7, 5} yielding a strict majority of 5, while processes p_2 and p_3 receive {5, 5, 7, 7} yielding a tie that is broken to 5 by the smallest-value rule. If instead the Byzantine process sends (4, 7) to p_1 but (4, 5) to p_2 and p_3, then p_1 receives {5, 5, 7, 7} (tie, decide 5) while p_2 and p_3 receive {5, 5, 7, 5} (majority 5). More critically, with different initial values such as $v_1 = 3$, $v_2 = 8$, $v_3 = 8$, if the Byzantine sends (4, 8) to p_1 but (4, 3) to p_2 and p_3, then p_1 sees {3, 8, 8, 8} (majority 8) while p_2 and p_3 see {3, 8, 8, 3} (tie, decide 3). This demonstrates that Round 1 alone cannot guarantee agreement.

2)Show that at the end of the second round, a correct process cannot re ceive, for the same j, 2 pairs (j, v_j) and 2 pairs (j, v'_j) with v_j ≠ v'_j.

We prove by contradiction that a correct process cannot receive two pairs (j, v_j) and two pairs (j, v'_j) with $v_j \neq v'_j$ for the same identifier j.

Assume for <u>contradiction</u> that some correct process p_i receives at least two pairs (j, v_j) and at least two pairs (j, v'_j) with $v_j \neq v'_j$ by the end of Round 2. Let n_v denote the number of pairs (j, v_j) received and n_v' denote the number of pairs (j, v'_j) received. By assumption, $n_v \geq 2$ and $n_v' \geq 2$.

A correct process can receive information about j from at most two sources: directly from j in Round 1 (at most one pair), and from relays by other processes in Round 2. Since process j does not relay information about itself, only the other n - 1 = 3 processes can provide information about j across both rounds. Therefore, p_i receives at most 1 (from j in Round 1) + 3 (from others in Round 2) = 4 pairs total for identifier j.

If j is correct, it sends the same pair (j, v_j) to all processes in Round 1. At least two other processes are correct (since f = 1) and will relay (j, v_j) in Round 2. Thus, p_i receives at least three pairs with v_j. The Byzantine process can contribute at most one pair with v'_j. This gives $n_v \geq 3$ and $n_v' \leq 1$, contradicting our assumption that $n_v' \geq 2$.

If j is Byzantine, all information about j comes from Round 1 reception by the four processes and subsequent Round 2 relays by three processes (excluding j). The total number of pairs p_i can receive is at most 4. For $n_v \geq 2$ and $n_v' \geq 2$ to hold simultaneously, we would need $n_v + n_v' \geq 4$. However, achieving exactly $n_v = 2$ and $n_v' = 2$ requires that exactly two processes send v_j and exactly two processes send v'_j. Since j is Byzantine and does not relay in Round 2, only three other processes relay information.

Therefore, our assumption leads to a contradiction. A correct process cannot receive two pairs with v_j and two pairs with v'_j for the same j.

3)Show that, at the end of the second round, given a process j, all correct processes assign to j the same value.

If j is correct, it sends (j, v_j) to all in Round 1. Each correct process receives this value directly, and receives it again from other correct processes in Round 2, accumulating three instances of v_j. The Byzantine process contributes at most one contradictory value, which is overwhelmed by the strict majority of three. All correct processes assign v_j to j.

If j is Byzantine, it may send different values in Round 1. However, in Round 2, all correct processes relay what they received. Each correct process learns what all other correct processes received from j, forming the same multiset of values. Since all correct processes apply the majority function to the same multiset, they compute the same result and assign the same value to j. By contradiction: If a Byzantine value were selected in Round 2, it must have been relayed by a majority of processes; with only one Byzantine process, that majority includes correct ones, so the accepted value is effectively validated by correct nodes and cannot be false.

4) Deduce that the executions of this protocol verify the agreement condition. Prove.

From Problem 3, all correct processes assign identical values (w_1, w_2, w_3, w_4) to all process identifiers. Each correct process then applies the majority function to this common vector. Since they have identical inputs and use the same deterministic function, all correct processes decide the same value. The Agreement property is satisfied. By contradiction:

Let *di* denote the decision value of process i, and let M() denote the majority-tiebreaker function used by all processes to compute their final decision from the set of assigned values. Let's asume, towards a contradiction, that the agreement property does not gol, that there exist two correct processes p and q, such that: dp ≠ dq.

By definition of the method, each process computes its decision deterministically as:

Dp = M(assignp(1), …, assignp(n)); Dq = M(assignq(1),…, assignq(n))

From 3), we know that at the end of round 2, all correct processes assign the same value to every identifier j. That is: ∀j, assignp(j)=assignq(j)

Therefore, the input vectors to M at processes p and q are identical:

(assignp(1),…, assignp(n)) = (assignq(1),…,assignq(n))

Since M is deterministic, idential inputs mean identical outputs: M(ap,) = M(aq,), in other words, dp and dq should be identical: dp = dq which contradicts our initial assumption. Agreement is confirmed.

5) Do they verify the other conditions of the Byzantine consensus? Prove

Termination. The protocol executes exactly two synchronous rounds with no conditional loops or blocking operations. After Round 2, each correct process performs deterministic computations on finite data. Every correct process reaches a decision in finite time, even if crashes our anormal conditions happen.

Validity. Assume all correct processes start with value v. In Round 1, each correct process broadcasts v. After Rounds 1 and 2, each correct process has at least three instances of v associated with each correct process identifier, versus at most one contradictory value from the Byzantine process. The majority mechanism assigns v to all correct processes. When computing the majority of assigned values, the vector contains at least three instances of v, forcing the decision to be v. As we have proved in 4), same inputs mean same output, and if every input is identical, then the only deterministic output posible is that input. Hence, validity is confirmed.