SafeNet for Computer Network EDA387

Huy Hoang Phung - huyhoang@chalmers.student.se

November 5, 2025

Algorithm 1 SNA for directed anonymous rings; code for processor p_i, where $M = n+1$

```
1: upon a pulse begin
2: let lr ← read r_{i−1 mod n};
3: let x ← min(r_i, lr);
4: if x ≠ n then
5:     x ← (x − 1) mod M;
6: end if
7: if x = 0 then
8:     signal();
9: end if
10: write x to r_i;
11: upon trigger() begin
12: if r_i = M − 1 then
13:     write M − 2 to r_i;
14: end if
```

Question 2: Self-Stabilizing Simultaneous Activation (20 points)

We consider a synchronous anonymous computer network in which processors communicate through shared memory. In each global pulse, all processors perform one atomic step simultaneously: at the beginning of the step, every processor reads the registers of its directly connected neighbors, performs local computation, and finally writes to its own register. The processors have no globally unique identifiers and execute the same deterministic algorithm. The communication topology is modeled as a connected graph $G = (P, E)$, where each vertex corresponds to a processor and each edge represents a bidirectional communication channel.

This question studies the problem of Simultaneous Network-wide Activation (SNA), in which all network processors must invoke signal() in the same synchronous round after one of them receives an earlier trigger() event. The goal of SNA is to design an algorithm satisfying the following properties (during their legal executions):

- *Simultaneity:* All processors invoke signal() in the same synchronous round.

- *No early signal:* No processor invokes signal() before the earliest possible common round.
- *Optimality:* The number of synchronous rounds until the activation round is asymptotically minimal with respect to the network size $n = |P|$, and the memory requirements are within $O(\Delta \log N)$, where N is an upper bound on n and Δ is an upper bound on the node degree in $G = (P, E)$.

Part I: Simultaneous Network-wide Activation (SNA) on a Directed Ring.

Algorithm 1 presents a solution for the case in which $G = (P, E)$ has the topology of a directed ring.

(2.a (1 point)) Define the set of legal executions for the SNA problem.

(2.b (1 point)) Define the set of safe configurations with respect to the SNA problem and Algorithm 1.

(2.c (6 points)) Algorithm 1's Consider an execution R of Algorithm 1 that starts from a configuration in which $\forall i \in P : r_i = n$, and is immediately followed by a step in which some processor p_j invokes trigger(). Prove that the ensuing activation satisfies requirements R-1, R-2, and R-3. (Each requirement is 2 pt)

For the sake of simplicity of this question, there is no need to prove the convergence of Algorithm 1. From this point on in this question, please assume that Algorithm 1 is self-stabilizing and that its stabilization time is in $O(n)$.

Part II: Simultaneous Network-wide Activation (SNA) on a Non-Rooted Tree

In this part of the question, we study self-stabilizing solutions for SNA on non-rooted trees.

(2.d (6 points)) **Reuse Algorithm.** Please design a solution to the SNA problem on non-rooted trees. You are welcome to reuse Algorithm 1 via a composition between the SNA for directed rings and an algorithm of your choice. The algorithm can be either from the Appendix or not. However, if the algorithm does not appear in the Appendix, please provide the complete pseudo-code and explain all the details needed to understand the composition.

Hint: Where in the course have we designed algorithms for anonymous unrooted trees? What techniques were used there?

(2.e (2 point)) Define the set of safe configurations with respect to your solution in item 2.d of this part.

(2.f (6 points)) **Correctness ideas.** Describe the key ideas for proving that your solution is self-stabilizing. Specifically, explain why the composition is correct, and if you have used or proposed any algorithm that does not appear in the Appendix, you must also outline the key ideas for proving its correctness. (Composition correctness: 2 pt; correctness of the algorithms not appearing in the Appendix, or stating that an algorithm from the Appendix that you use is correct and under which assumptions: 4 pt in total for all algorithms.)

Proof. **Part I**

- **Set of legal execution** Any execution that statifies these following can be considered as a legal execution:
  - All nodes must hold the same value (equals to n, where n is the total number of nodes in this system).
  - At round r_t, one or more nodes receive a $trigger()$.
  - No early fire, all processors wait for others to invoke the $signal()$ in the same round.
  - Signal round r_s and trigger round r_t must be different, and statifies the condition that $r_s > r_t$
- **Safe configuration** When all the processors hold the same value (and equals to n)
  - With value n, the value of x is not changing, hence the line 4-9 will be skipped.
  - Line 10 keeps the value of r_i unchanging, hence, the system is safe.
- **Prove**
  - When p_j is triggered, lines 12-14 will be actived, changing the value of r_j
  - Following this step, when a *pulse* wakes another processor k, neighbor of j. Because j has already changed its value, line 2-5 will be executed.
  - **R1** The minfunction will ensure each neighbor looks into their right node in this ring and set the clock to "lower" value. Then subtracts 1 (for countdown), writes new value in shared register (line 2-6 and line 10). A node only fires when line 7 is met. With the characteristics of ring topology, the "trigger wave" will be spread to all other nodes.
  - **R2** Because each node reads its directly connected neighbor first, the performs computation, it guarantees that at most 1 node can change the value of register at the round following $trigger()$ (the directed neighbor of node received trigger). Furthermore, the line 7 ensures only when $x = 0$, node can invoke $signal()$. Combining with $R1$, there is no early firing in this system.
  - **R3** As stated, each time, there is only 1 additional node changes its "clock" by performing line 2-6, and all these nodes will followed the "clock" of fastest node. Therefore, in n round, the activation will take place.

Part II

- **Reusable** The problem can be solved by using a combination from the SNA algorithm ($AL1$) and the Algorithm 20 ($AL20$) from the Appendix (Couting problem in the non-rooted anonymous tree).
  - **Correctness** $AL20$ will give each node two information: one about the total number of nodes in the system and for each node p_j, neighbor of p_i, information about how many nodes in the subtree toward to j, excluding the branch contains j itself.
  - **Idea** $AL1$ provides control mechanism for the synchronous firing, while $AL20$ is responsible for advertising the path for "trigger" signal to propagate through the tree.

Algorithm 2 SNA for Non-Rooted Trees (General Graph); code for processor p_i

```
1: Variables:
2: n ← (total nodes or processors in this tree)
3: M ← n + 1
4: r_i ← n {SNA state register, initialized to the safe state}
5: N(i) ← Set of neighbors of p_i
6: upon trigger() begin
7: if r_i = n then
8:    write r_i ← n − 1 {Start the countdown}
9: end if
10: end
11: upon synchronous pulse begin
12: do forever begin
13: min_neighbor_state ← n
14: for all p_j ∈ N(i) do
15:    l_j ← read(r_j)
16:    if l_j < min_neighbor_state then
17:       min_neighbor_state ← l_j
18:    end if
19: end for
20: z ← min(r_i, min_neighbor_state)
21: if z ≠ n then
22:    if z = 0 then
23:       signal()
24:       write r_i ← n
25:    else
26:       write r_i ← (z − 1) mod M
27:    end if
28: else
29:    write r_i ← n
30: end if
31: end
```

- **Safe configuration** every node must hold two values, the first one is total number of nodes in this system in register $count[i]$, the second one is the value n in the

register r_i for the firing. With this configuration, line 13-20 will repeatly write the same value n. Then line 21 to 28 will be skipped.

- **Correctness**
  - **Counting tree** This has been proved before, from any chaostic state, the network will be stabilized and all node will know how many nodes are there in this system.
  - **SNA** Has been proved to be correct and convergence is $O(n)$

  In the adapted algorithm, from line 13-19, a node p_i only takes the faster neighbors j. Moreover, it also enusre (line 20) this signal only propagated 1 hop per round. When the firing is done, line 24 will reset the value to n, achieving safe configuration.

- **R1** line 20 acts as a shortest path countdown spreading. When a node p_j is triggered, its r_j is set to $n-1$, statifies line 21, allowing the signal to be propagated. i node with distance d_i will receive this signal after d_i round because in line 20, each time, a node j only perform computation after reading, hence, the $trigger()$ must travel at most 1 hop each round. After receiving the signal, node i will begin countdown from $n - d_i$. The total time for the firing will be $d_i + n - d_i = n$. This result does not depend on how far from i to j, so every nodes fire exact the same round (after n).

- **R2** the $trigger()$ sets the counter for node j from $n-1$, In line 26, r_j only decreases 1 by round. With the condition $R1$ above, no processor can fire before round n.

- **R3** activation time is n round, which is minimal with respect to the network size.

□

Question 3: Self-Stabilizing Leader Election (20 points)

Let $P = \{p_0, \ldots, p_{N-1}\}$ be a finite set of processors, such that any processor can communicate directly with any other processor in the network. Each $p \in P$ has a globally unique identifier, (ID). It may also, without explicit notification, fail-and-stop (i.e. stop taking steps for a finite or infinite period of time) but can later resume execution. We say that a processor is non-faulty if it is currently up and connected. We say that the network is stable if there exists at least one processor that never fails (i.e. never performs a fail-stop) throughout the entire network execution.
The Fault-Tolerant Leader Election (FTLE) problem requires that all non-faulty processors eventually agree on one processor, called the leader, such that the following requirements hold:

- **Uniqueness:** Eventually, at most one processor is considered leader (even if the network is unstable).

- **Agreement:** If the network is stable, all non-faulty processors quickly recognize the identifier of the same leader.

- **Validity:** If the network is stable, the elected leader is the non-faulty processor that has been continuously operational for the longest time. (Ties are broken by selecting the processor with the highest identifier).
- **Termination:** If the network is stable, every non-faulty processor quickly knows the leader's identity.

(3.a (1 point)) What is the model most suitable for solving the Fault-Tolerant Leader Election (FTLE) problem? Please choose from the models if you think there is more than one correct answer; select the one that can help you the most to simplify your answer to the rest of this question. Clearly justify your selection.

- An asynchronous shared-memory network with the topology of a fully connected graph.
- Synchronous message-passing network with the topology of a general graph and reliable communication channels.
- Synchronous message-passing network with the topology of a general graph and fair communication.
- An asynchronous message-passing network with the topology of a general graph and fair communication.
- Synchronous shared-memory network with the topology of a fully connected graph.

(3.b (1 point)) Define the set of legal executions for the Fault-Tolerant Leader Election (FTLE) problem.

(3.c (8 points)) Provide yourself a detailed pseudo-code for a self-stabilizing solution to the FTLE in this question. (you are allowed to use a single unbounded shared variable per processor (and showing the method for bounding this variable is not required for the sake of simplicity)). Your pseudocode must be self-contained and specify: local state variables, message types (or shared variables), and event handlers (e.g., message arrivals, do-forever loops, upon pulses), and it must follow the model selected in item 3.a of this question. You may draw inspiration from Algorithm 16 in the Appendix, but your solution must be adapted to the FTLE specification (Uniqueness, Agreement, Validity, Termination) and stand on its own without cross-referencing the Appendix.

Hint: Can an unbounded counter help to satisfy the Validity requirement regarding the longest continuous operation of the elected leader? As mentioned, for the sake of simplicity of this question, you may assume one unbounded counter per processor.

(3.d (3 points)) Define the set of safe configurations with respect to the FTLE problem and the algorithm you have proposed in item 3.c of this question.

(3.e (5 points)) Prove that, when the network execution starts in a safe configuration (from 3.d), the execution of the algorithm you have proposed in item 3.c satisfies the requirements of Uniqueness, Agreement, Validity, and Termination (1.25 pt per requirement).

(3.f (5 points)) Prove that, when the network execution starts in any configuration, the algorithm you have proposed in item 3.c eventually brings the network to a safe configuration. Note that providing the exact or asymptotic stabilization time is not required; a very long stabilization time is acceptable for your solution.

(3.g (1 point)) What are the communication costs of the algorithm you have proposed in item 3.c? Specify the message (or shared-variable) size: list all message types (or registers), their payload fields, and their asymptotic sizes (e.g., O(log N) bits for identifiers, where N is an upper bound on the number of processors n). Briefly justify your bounds.

Algorithm 3 Self-Stabilizing FTLE Algorithm for processor p_i

```
1: Shared state in p_i:
2: R_i.epoch ← 0
3: R_i.leader ← ID(i) {Initially vote itself}
4: do forever begin
5: S ← snapshot of all R_j registers
6: L_id ← ID(i);    L_epoch ← S[i].epoch
7: for all p_j ∈ P do
8:    if S[j].epoch > L_epoch then
9:       L_id ← ID(j);    L_epoch ← S[j].epoch
10:   else if S[j].epoch = L_epoch and ID(j) > L_id then
11:      L_id ← ID(j);    L_epoch ← S[j].epoch
12:   end if
13: end for
14: if L_id = ID(i) then
15:   R_i.epoch ← R_i.epoch + 1
16:   R_i.leader ← ID(i)
17: else
18:   R_i.epoch ← 0
19:   R_i.leader ← L_id
20: end if
21: end
```

Proof. **Model choosing & idea**

- **3.a Model** An asynchronous shared-memory network with the topology of a fully connected. This model is suite for this problem because:
  - This problem states that processor can nap/fail then runs again, so rely on asynchronous round is not suitable (because this system must run without notification from failed ones).
  - Because the usage of unbounded register is allowed and in the $AL16$, node can read its neighbor's register to update its value, shared memory is a suite model.
  - Fully connected allows nodes to agree on electing a leader.
- **3.b Legal execution**

  - At any point of time, after t successfully operated rounds, there is one and only a processor i acts as a leader for this sytem. This also happens if the network is unstable.
  - If the network is stable, all processors must have the same leader.
  - A leader is a processor that operated for the longest time and also has the highest ID.
  - Every processors must quickly learn the identity of the leader.

- **3.c Proposed** $AL3$ above is the proposed solution for this problem. Each processor maintains a register (unbounded) to store its operated time (*epoch*). It also writes this information along with its "leader" ID to exchange with other processors in this system (register R).

- **3.d Safe configuration** when there is only 1 processor (leader) keeps increasing its *epoch*, other processors continuously reset *epoch* to 0 and also have the same value for L_id.

- **3.e Convergence in safe configuration**

  - **Uniqueness** line 7-13 enforces processor i to give up leadership for any j that operated for a longer time (tie break with ID).
  - **Agreement** line 16 and 19 ensure i always writes the result in the shared register R. And the $doforever$ allows processor to exchange these information with neighbors. Moreover, this is a fully connected graph, this information will reach all processors.
  - **Validity** if network is stable, all processors can execute the computation and write to shared register R, allowing all neighbors to read and agree in a common ledear.
  - **Termination** even if the network is unstable, line 17-19 ensures all processors should overwrite all garbage value in the *epoch*, only running processor can elect itself.

- **3.f Convergence in faulty condition**

  - **leader election** because all processors are running the same deterministic algorithm, they will decide to pick the same processor to be the leader (line 7-13). The conditions are: the longest operating with the highest ID.
  - **correction wave** when a leader is elected, every other processors will execute line 17-20, flush all faulty value. Meanwhile, leader keeps increasing its *epoch*, ensure its leadership. This is indicated from line 14 to 16.

    Even with an error that causes running processors to have multiple values of *epoch*, because ID is unique, a leader will be picked, starting the reset to 0 of all other *epoch*.

- **3.g Communication cost** for ID, it required O(log N) bit to represent N processors. Register L_id also takes O(log N). The *epoch* is unbounded, as stated in line 15, it keeps increasing if the leader runs forever. In conclusion, the cost for the register R(shared) is unbounded.

□