Home Assignment for Computer Networks (EDA387)

Elad Michael Schiller

October 17, 2024

Please submit your typed solutions in pairs via Canvas. To submit the assignment, join a lab group with your partner. For each deadline, two questions are available. Ensure you answer at least one of them.

1 Value discovery in complete graphs (due: September 19, 2024)

Consider a set, $\mathcal{P} = \{p_0, \ldots, p_{n-1}\}$, of n processors, such that each processor p_i is associated with two registers r_i and s_i (both of constant size that is independent of n). Processor p_i cannot read the value in s_i, however, any other processor can. Processor p_i has both read and write access rights to r_i and any other processor has only read-only access rights to r_i. The value in s_i is unknown to p_i. The other processors can help p_i to discover this unknown value. For example, suppose that $n = 2$. Processor $p_{(i+1) \bmod 2}$, can write the value of s_i to $r_{(i+1) \bmod 2}$ and then p_i can discover s_i's value by reading $r_{(i+1) \bmod 2}$'s value. Please solve the problem, which is to let p_i discover the secret s_i, for the case in which $n > 2$ is any finite known value.

Question 1. Assume that all processors have access to a globally unique identifier within the range $\{0, \ldots, n\text{-}1\}$, *i.e.*, the identifier is defined by a one-to-one function ID from $\mathcal{P}$ to $\{0, \ldots, n\text{-}1\}$. For instance, $\forall p_i \in \mathcal{P} : ID(p_i) = i$, *i.e.*, the identifier of processor p_i is equal to its index i. In practice, MAC addresses are generally assumed to be globally unique. Is there a solution when processors have globally unique identifiers? Prove your claims.

The answer idea

Algorithm 1 uses the order of the identifiers to construct a virtual ring, generalizing the example provided in the question. The solution is correct because, due to the properties of globally unique identifiers and lines 2 to 4, all processors will eventually store the same values in $ids[]$, $prev$, and $next$. Therefore, all processors will eventually use an identical virtual ring when reading the secret from the previous processor in the ring (line 5) and writing it to their own register. As a result, all processors will eventually

output their own secrets when reading from the next processor in the virtual ring (line 6). Note that since we are only interested in the value eventually output by the processors, no synchrony assumptions are needed, as all processors will eventually write the same values to their registers.

Algorithm 1: Using globally unique identifiers, code for processor p_i

```
1 do forever begin
2     foreach p_k ∈ P do lr[k] := read(ID(k));
3     let ids[] := sort({lr[k] | 0 ≤ k ≤ n − 1}) /* ids is a sorted identifier array;
4     let (prev, next) := (k − 1 mod n, k + 1 mod n), where ids[k] = ID(i);
5     write (read(s_ids[prev])) to r_i;
6     output(read(r_ids[next]));
```

Question 2. Now, suppose processors only have access to locally unique identifiers. Specifically, let $N(p_i) \subseteq \mathcal{P} \setminus \{p_i\}$ denote the neighborhood of processor $p_i \in \mathcal{P}$, which is the set of all processors directly connected to p_i. For each $p_i \in \mathcal{P}$, the local identifier is defined by a one-to-one function ID_i from $\mathcal{P}$ to $\{0, \ldots, n\text{-}1\}$. Note that $ID_i()$ depends on p_i, whereas $ID()$ is identical for all processors. Furthermore, for processors $p_i, p_j, p_k \in \mathcal{P}$, it may be the case that $ID_i(k) \neq ID_j(k)$. For example, port numbers are unique to the host but not across the Internet. Is there a solution when processors have only locally unique identifiers? Prove your claims.

The answer idea

Note that for the case of $n = 3$, and $r_0 := s_1 \oplus s_2$, $r_1 := s_0 \oplus s_2$, $r_2 := s_0 \oplus s_1$, we can say that p_0 can read r_1 and r_2 as well as s_1 and s_2. Therefore, $r_1 \oplus s_2 = s_0 \oplus s_2 \oplus s_2 = s_0$. The idea is to study the general case, and $r_0 := s_1 \oplus \ldots \oplus s_{n-1}$, $r_1 := s_0 \oplus s_2 \oplus \ldots \oplus s_{n-1}$, $\ldots$, $r_{n-1} := s_0 \oplus \ldots \oplus s_{n-2}$, we have that $r_1 \oplus s_2 \oplus \ldots \oplus s_{n-1} = s_0$. In general, $s_i = r_j \oplus \left(\bigoplus_{k \notin \{i,j\}} s_k \right)$, where the processor ordering is local to p_i. Algorithm 2's correctness proof follows similar arguments as in Algorithm 1's correctness proof.

Algorithm 2: Using local identifiers, code for processor p_i

```
1 do forever begin
2     foreach p_k ∈ P \ {p_i} do lr[k] := read(s_k);
3     write ( ⊕_{p_k ∈ P\{p_i}} lr[k]) to r_i;
4     let rJ := read(r_j) where p_j ∈ P \ {p_i};
5     output r_j ⊕ ( ⊕_{p_k ∈ P\{p_i,p_j}} lr[k] );
```

2 Self-stabilizing vertex-coloring and maximum matching (due: September 26, 2024)

Let $\mathcal{P} = \{p_0, \ldots, p_{n-1}\}$ be n processors on a directed ring that Dijkstra considered in his self-stabilizing token circulation algorithm (in which processors are semi-uniform, i.e., there is one distinguished processor, p_0, that runs a different program than all other processors but processors have no unique identifiers). Recall that in Dijkstra's directed ring, each processor p_i can only read from $p_{(i-1) \bmod n}$'s shared variables and each processor can use shared variables of constant size. For each of the following queations, please give a clearly written pseudo-code, define the set of legal executions, provide a correctness proof with all the arguments needed to convince the reader that the algorithm is correct and self-stabilizing. Prove your claims.

Question 1. The *vertex-coloring* problem is crucial for resource allocation, scheduling, and network topology control tasks. It involves assigning colors to the processors (aka vertices of a graph) so that no two neighboring processors share the same color. Specifcially, each $p_i \in \mathcal{P}$ must be assigned a color x_i, such that $x_i \neq x_{(i-1) \bmod n}$. The main complexity measure is the total number of colors k used to color the network. Design a deterministic self-stabilizing vertex-coloring algorithm that always provides an optimal solution (after stabilization). Assume that n is known.

The answer idea

The solution is presented in Algorithm 3, and the correctness proof is combined with that of Question 3. The chromatic number of a graph is defined as the smallest integer k such that the graph can be correctly colored with k colors. In the case of a ring topology, we clarify that the chromatic number is two when the number of nodes is even. For example, the coloring $(x_0, x_1, x_2, x_3, x_4, \ldots) = (0, 1, 0, 1, 0, \ldots)$ applies. When the number of nodes is odd, the chromatic number becomes three, as illustrated by $(x_0, x_1, x_2, x_3, x_4, \ldots) = (2, 1, 0, 1, 0, \ldots)$.

Algorithm 3: Self-stabilizing vertex-coloring for networks with the topology of semi-anonymous directed rings

1 **shared variables:** $x \in \{0, 1, 2\}$ **stores a color**

2 **Code for processor** $p_i : i = 0$
3 $x_i \leftarrow x_{n-1} + 2 \bmod 3$

4 **Code for processor** $p_i : i \neq 0$
5 $x_i \leftarrow x_{i-1} + 1 \bmod 2$

Question 2. The *maximum matching* problem is essential in optimizing resource allocation, communication, and network design. It involves finding the largest possible set

of pairs of neighboring processors, aka edges, in the network, such that no two pairs share a processor. Specifically, a matching $M \subseteq \{(p_i, p_{(i-1) \bmod n}) : p_i \in \mathcal{P}\}$ is a subset of neighboring pairs, such that no two neighboring pairs in M share a processor, *i.e.*, for any two neighboring pairs $(p_i, p_{(i-1) \bmod n})$ and $(p_j, p_{(j-1) \bmod n})$ in M, $p_{(i-1) \bmod n} \neq p_j$ and $p_{(j-1) \bmod n} \neq p_i$. The goal is to find a matching M to maximize the number of neighboring pairs in M. Design a deterministic self-stabilizing maximum matching algorithm that always provides an optimal solution (after stabilization). Assume that n is known.

Question 3 (bonus). What if n is unknown, except for the fact that it is finite? Please solve Questions 1 and 2 under the assumption that n is unknown.

The answer idea

The correctness proof of Algorithm 4 shows by induction on the value of n that the value of x_0 is 0 for the case of n is even and 2 otherwise. The rest of the proof is implied but the definition of the function *output*().

Algorithm 4: Self-stabilizing maximum matching for networks with the topology of semi-anonymous directed rings

```
1  shared variables: x ∈ {0, 1, 2} stores a color

2  Code for processor p_i : i = 0
3  x_i ← x_{n−1} + 2 mod 3

4  Code for processor p_i : i ≠ 0
5  x_i ← x_{i−1} + 1 mod 2

6  function output() begin
7      switch x_i do
8          case x_i = 0 do print 'p_i and p_{i−1 mod n} are matched';
9          case x_i = 1 do print 'p_i and p_{i+1 mod n} are matched';
10         case x_i = 2 do print 'p_i is single';
```

Theorem 2.1. *Algorithm 4 is a self-stabilizing solution to the problem of maximum matching that stabilizes within $\mathcal{O}(n)$ asynchronous cycles.*

Proof of Theorem 2.1 The proof is implied by Lemma 2.4, which uses Claims 2.2 and 2.3. The use of the proof technique of convergence stairs in order to deal with the challenges imposed by the model of distributed fair scheduler. Specifically, the proof of Lemma 2.4 uses the following stairs:

- (a) Invariant (i) of Claim 2.2,

- (b) Claim 2.3, and
- (c) Invariant (ii) of Claim 2.2.

Claim 2.2. *(i) Starting in any configuration and within one asynchronous cycle, the network reaches configuration, c_1, in which $x_0 \in \{0, 2\}$. (ii) Moreover, suppose that $x_{n-1} = 0$ and $x_{n-1} = 1$ throughout the execution of Algorithm 4. Then, $x_0 = 2$ and $x_0 = 0$, respectively.*

Proof of Claim 2.2 By line 5, the shared variable x_{n-1} can only encode the values 0 or 1. Starting in any configuration and within one asynchronous cycle, the claim holds due to line 3. $\square_{Claim\ 2.2}$

Claim 2.3. *Let* $\mathsf{altOneZero}(k) \equiv (1, 0, 1, 0, \ldots)$ *an (k)-length alternative sequence of 1's and 0's that starts at 1. Let $k \in \{1, \ldots, n-1\}$ and* $\mathsf{seqWithoutRoot}(k) \equiv (x_1, x_2, \ldots, x_k)$ *be a sequence that lists the values that the ring's registers (except the root) encode. Starting in c_1 and within $n-1$ asynchronous cycle, the network reaches configuration, c_n, in which* $\mathsf{seqWithoutRoot}(n-1) = \mathsf{altOneZero}(n-1)$.

Proof of Claim 2.3 The proof shows that, for any $k \in \{1, \ldots, n-1\}$, it holds that $\mathsf{seqWithoutRoot}(k) = \mathsf{altOneZero}(k)$ by induction on $k \in \{1, \ldots, n-1\}$. The proof for the case of $k = 1$ is implied by Claim 2.2 and line 5. Suppose the claim is true for every $k \in \{1, \ldots, n-2\}$. The proof for the case of $k+1$ is implied by the induction hypothesis and line 5. $\square_{Claim\ 2.3}$

Lemma 2.4. *Let* $\mathsf{evenOdd}(n)$ *be the function that returns the sequence* (0) *when n is even and* (2) *otherwise. Also, $seq \equiv (x_0, x_1, x_2, \ldots, x_{n-1})$ is a sequence that lists the values that the ring's registers encode. (i) Starting in any configuration and within $n+1$ asynchronous cycles, the network reaches configuration c_{n+1} in which $seq = \mathsf{evenOdd}(n) \circ \mathsf{altOneZero}(n-1)$, where $\circ$ is the concatenation operator, which returns the joint sequence of its operands. (ii) Moreover, c_{n+1} is safe with respect to Algorithm 4 and the problem of maximum matching.*

Proof of Lemma 2.4 Inviarient (i) of the lemma is demonstrated by using Invariant (i) of Claim 2.2 and then Claim 2.3 before using Invariant (ii) of Claim 2.2. The proof of Invariant (ii) of the lemma is implied by lines 8 to 10 and the fact that when n is even, every processor can have a match. Furthermore, when n is odd, there is one processor, e.g., the root, is a single processor that cannot have a match. Thus, Algorithm 4 is a self-stabilizing solution to the studied problem. $\square_{Lemma\ 2.4}$ $\square_{Theorem\ 2.1}$

3 Self-stabilizing Node Counting Algorithm on an Anonymous and Oriented Path Graph and Trees (due: October 8, 2024)

Consider an asynchronous computer network with a distributed but fair scheduler, *i.e.,* not a central daemon. The network that has n nodes, where $n < N$ is a finite number

that is unknown to the algorithm, and N is an upper bound on n, such that only the value of $\lceil \log_2 N \rceil$ is known. The network is based on shared memory where each processor can write to a single register with up to $O(\lceil \log_2 N \rceil)$ bits. Recall that each register can be divided into multiple fields, say, one field per neighbor. The processors do not have globally unique identifiers, and all nodes run the same program without the presence of a distinguished processor.

Your task is to design a self-stabilizing algorithm that operates within an asynchronous network with a graph topology, which we specify below. The algorithm is required to achieve an accurate node count. After a period of recovery from the occurrence of the last transient fault, each processor should output, using the operation `print(x)`, the total number x of nodes in the system.

Please present a comprehensive explanation of your solution along with your exact assumptions.

1. Provide a well-structured pseudo-code for your algorithm.
2. Define the set of legal executions.
3. Present a proof of correctness, complete with all necessary arguments to convincingly demonstrate the algorithm's accuracy and self-stabilization. If needed, separate between the convergence and the closure proofs.
4. What is the stabilization time of the proposed algorithm?
5. Does a self-stabilizing naming algorithm exist for the system described? If such an algorithm exists, provide a detailed description and proof of its correctness. If not, provide proof of the impossibility of the result.

Question 1. Suppose the network topology is of an *oriented bidirectional path graph* P_n. Specifically, each register is divided into two fields. That is, processor p_i can read its neighbors' registers, left_j or right_k while writing/reading only to its left_i and right_i fields in its register, where p_j and p_k are p_i's neighbors. The path is oriented from left to right in the sense that each node has a right and/or left neighbor. That is, starting from the leftmost node (which has no left neighbor, *i.e.*, $\text{left}_i = \perp$) and taking exactly n hops to the right brings us to the rightmost node (which has no right neighbor, *i.e.*, $\text{right}_i = \perp$).

The answer idea

The solution consists of two self-stabilizing algorithms, enabled by the theorem on fair composition of self-stabilizing algorithms (see Chapter 2.7).

First algorithm: Define one of the extreme nodes, say the leftmost one, as the root node, $p_{root} \in \mathcal{P}$, where $\text{left}_{root} = \perp$. Using the algorithm by Dolev, Israeli, and Moran [Distributed Computing, 1993] (see Chapter 2.5), span a BFS tree rooted at p_{root}. This algorithm stabilizes within $\mathcal{O}(n)$ asynchronous cycles.

Second algorithm: Define the other extreme node, $p_{source} \in \mathcal{P}$, where $\text{right}_{source} = \perp$, as the information source node. Using the output of the first algorithm, p_{source} stores its distance to p_{root} in a shared register, $count_{source}$. Each node $p_i \in \mathcal{P} \setminus \{p_{source}\}$ writes the value of $count_j$, where p_j is its child, into $count_i$. By induction on the distance from p_{source}, the information in $count_{source}$ propagates to $count_i$ until it reaches $count_{root}$ within n asynchronous cycles after the stabilization of the first algorithm. Thus, the second algorithm can output the value in $count_k$ for any $p_k \in \mathcal{P}$, as the composition of the two algorithms stabilizes within $\mathcal{O}(n)$ asynchronous cycles. Also, the ranking problem is solved by using the distance field from the first algorithm.

Question 2. Consider the following definition of a tree graph, which we later extend:

- *Connectedness:* There is a path between any pair of vertices in the tree.
- *Acyclicity:* The tree contains no cycles.
- *Finite vertices:* The tree has a finite number of vertices and edges.
- *Edges:* Suppose there are n vertices in the tree. The number of edges is $n - 1$.

This question assumes that the network topology is of a *non-rooted tree*, which is a tree network with only local identifiers. This is unlike rooted tree networks, where one of the nodes is distinguished.

The answer idea

Algorithm 5: Self-stabilizing Counting for Non-rooted Anonymous Tree Networks, code for processor p_i

```
1  do forever begin
2      // read the neighbors' registers into the local array lr[]
3      foreach p_j ∈ N(i) do lr[j] := read(r_{j,i});
4      let sum[δ] = [0, ..., 0]; // an array with an entry for p_i's neighbors
5      foreach p_j ∈ N(i) do
6          // consider the tree rooted at r_{i,j}
7          // sum up the number of nodes of the root's children
8          foreach p_k ∈ N(i) : p_j ≠ p_k do sum[j] := sum[j] + lr[k].count;
9          // store in the local register the number of nodes in the
               tree rooted at r_{i,j}, which includes one for the root and the
               summation of the children counters
10         write count[j] := sum[j] + 1 to r_{i,j};
11     end
12     // output the number of nodes in the tree, which includes one
           for the root and the summation of the neighbors' counters
13     output(1 + Σ_{p_j ∈ N(i)} count[j]);
14 end
```

Lemma 3.1 considers the following definitions. The height of a rooted tree is the maximum number of edges from the root to any leaf. Let the height of a communication register $r_{i,j}$ be defined as the height of the tree rooted at p_i, after removing the edge between p_i and p_j.

Lemma 3.1. *Let h be the height of the tree rooted at $r_{i,j}$. Within $\mathcal{O}(h)$ asynchronous cycles, Algorithm 5 counts correctly the nodes in the tree root at $r_{i,j}$.*

Proof of Lemma 3.1 The correctness of the register assignment in the counting algorithm is guaranteed by induction on the height of the communication registers.

Base case: In the first cycle of every fair execution, the $r_{i,j}$ field of every register of height 0 (i.e., a register associated with a leaf node) is assigned the value 1 by processor p_i (line 10). Notice that any subsequent prospective assignments by the leaf processor p_i to $r_{i,j}$ do not alter the value of $r_{i,j}$.

Induction hypothesis: We assume that the lemma is correct for every register of height less than $h \geq 1$.

Induction step: For every register of height $h + 1$, its value is computed using the values of registers of height $h' \leq h$ (lines 8 and 10). $\square_{Lemma\ 3.1}$

Since the height of the registers is bounded by the network diameter d, a safe configuration is reached within $\mathsf{O}(d)$ cycles. This implies Corollary 3.2.

Corollary 3.2. *Within $\mathcal{O}(d)$ asynchronous cycles, Algorithm 5 outputs correctly the nodes in the tree.*

Consider a computer network with the following tree topology $G := (\{0, 1\}, \{\{0, 1\}\})$. By reducing leader election in anonymous networks to the ranking problem, we observe that if ranking is solvable for anonymous trees, leader election in such trees would also be solvable since one can elect the lowest identifier. However, since leader election in anonymous trees is impossible (as studied in class), it follows that ranking in anonymous trees is also impossible.

4 The Link State Algorithm and Self-stabilization (due: October 15, 2024)

The Link State algorithm involves sending information not only to neighbors but to all network nodes. This question asks whether the link state algorithm is self-stabilizing or can become one. Algorithm 6 sketches the protocol for reliable flooding, as we learned in class. We clarify that the Link State algorithm also uses a local calculation. That is, the flooding method described in Algorithm 6 supplies the link state algorithm with a set of LSPs for each SEQNO. Using Dijkstra's shortest path algorithm, this link state algorithm enables each node to calculate the shortest path within a given graph locally.

Question 1.

- What is the model used by the Link-State algorithm? Please choose the most appropriate, correct, complete, and accurate answer. If you think there is more

Algorithm 6: Reliable Flooding for disseminating Link State Packets (LSPs) across a network.

1 Each node stores the latest Link State Packet (LSP) from every other node.

2 LSPs are forwarded to all nodes except the sender, preventing unnecessary retransmissions.

3 Periodic generation of new LSPs occurs, with an incremented Sequence Number (SEQNO). After a reboot, SEQNO starts at 0.

4 Upon hearing an LSP with SEQNO $= x$, a node sets its next SEQNO to $x + 1$.

5 Stored LSPs' Time-to-Live (TTL) decreases; LSPs expire when TTL reaches 0.

than one correct answer, select the ones that can help you simplify your answer to the rest of this question. Clearly justify your selection.

1. Asynchronous shared memory network with the topology of the fully connected graph with fair communication.
2. Synchronous message-passing network with the topology of general graph and reliable communication channels.
3. Synchronous message-passing network with the topology of the general graph with fair execution.
4. Asynchronous message-passing network with the topology of the general graph with fair communication.
5. Synchronous shared memory network with topology of fully connected graph and fair communications.

- Please provide a detailed pseudo code for reliable flooding, which is used by the Link State Algorithm. Your pseudo-code should consider the model you selected above.
- Discuss the self-stabilization properties of the algorithm you have provided above. A simple 'yes' or 'no' response is insufficient for credit; your answer should demonstrate a comprehensive understanding of the topic. For example, you may analyze how the algorithm behaves when initiated from an arbitrary starting state (related to the link-state algorithm), highlighting potential disruptions in network operation without the ability to self-recover. Conversely, if you believe the algorithm is self-stabilizing, you can outline a correctness proof while emphasizing its key assumptions.

The answer idea

Algorithm 7 presents a self-stabilizing solution to the problem of LSP (Link State Packet) reliable broadcast for synchronous message-passing networks with a general graph topol-

Algorithm 7: Self-stabilizing Reliable Flooding for disseminating LSPs

```
1  D an upper bound on the network diameter;
2  timeoutCounter an integer that counts down for simulating a timeout;
3  timeoutBound > D a constant that sets the timeout value;
4  SEQbound > (N+1)^2 a constant that serves a bound on SEQNO;
5  LSP[N] the most recent to arrive LSP messages. LSP[i] is p_i's number;
6  upon reboot do (timeoutCounter, LSP[]) ← (0, [⊥, ..., ⊥]);
7  upon a pulse do timeoutCounter ← max{0, timeoutCounter − 1};
8  upon a pulse begin
9      if timeoutCounter = 0 then
10         timeoutCounter ← timeoutBound;
11         let seqno := 0;
12         if LSP[i] ≠ ⊥ then seqno ← LSP[i].SEQNO + 1 [mod SEQbound];
13         LSP[i].(ID, NBHD, SEQNO, TTL) ← ⟨i, N(i), seqno, D⟩;
           /* use LSP[] to construct the graph of the network topology
              when computing reachability in Line 14                    */
14         foreach ℓ ∈ {0, ..., N − 1} : p_ℓ is unreachable from p_i do LSP[ℓ] ← ⊥;
15         foreach p_k ∈ N(i) do
16             send ⟨{LSP[ℓ] : LSP[ℓ] ≠ ⊥ ∧ LSP[ℓ].TTL > 0}_{ℓ∈{0,...,N−1}}⟩ to p_k
17 upon message ⟨LSPset⟩ arrival from p_j begin
18     foreach lsp ∈ LSPset do
19         lsp.TTL ← max{0, lsp.TTL − 1};
20         if LSP[lsp.ID] = ⊥ ∨ LSP[lsp.ID].SEQNO ≤ lsp.SEQNO then
21             LSP[lsp.ID] ← lsp;
22             LSP[lsp.ID].SEQNO ← LSP[lsp.ID].SEQNO + 1 [mod SEQbound];
23         foreach p_k ∈ N(i) \ {p_j} do
24             send ⟨{LSP[ℓ] : LSP[ℓ] ≠ ⊥ ∧ LSP[ℓ].TTL > 0}_{ℓ∈{0,...,N−1}}⟩ to p_k
```

ogy and reliable communication channels. The algorithm comes in two variations: one that uses unbounded *SEQNO* counters and another that uses bounded *SEQNO* counters. The former does not include the boxed code ferments (in Lines 12 and 22), whereas the latter does include these fragments as well as a revision of the underlined code fragment in Line 20, which we present after the proof of Lemma 4.1.

Lemma 4.1. *Let maxSEQNO be the maximum SEQNO value associated with p_i that is held by any node (other than p_i) or a message in the starting system state. Within* $\max\{timeoutBound, maxSEQNO\}$ *sysnchnous rounds, p_j stores $LSP_i[i]$'s value in $LSP_j[i]$, where $p_j \in \mathcal{P}$.*

Proof. Observations 1 to 3 imply the proof.

- **Observation 1:** Within $timeoutBound$ synchronous rounds, the if-statement condition in Line 9 holds. This is due to Lines 7 and 10.
- **Observation 2:** Within $(maxSEQNO + 2) \cdot timeoutBound$ sysnchnous rounds, $LSP_i[i].SEQNO > maxSEQNO$, where $LSP_i[]$ is $LSP[]$'s value in p_i. This is due to Observation 1 as well as Lines 12 and 22.
- **Observation 3:** Suppose $LSP_i[i].SEQNO > maxSEQNO$ holds in the starting system state. Within D synchnous rounds, every node p_j stores $LSP_i[i]$'s value in $LSP_j[i]$. This is by induction on the shortest path length between p_i and p_j as well as Lines 16 to 24.

□

For the self-stabilization variation with bounded $SEQNO$ counters, include the boxed code fregments and substitute $LSP[lsp.ID].SEQNO < lsp.SEQNO$ in Line 20 with $(0 \leq LSP[lsp.ID].SEQNO - N \wedge lsp.SEQNO \notin \{LSP[lsp.ID].SEQNO - N \bmod SEQbound, \ldots, LSP[lsp.ID].SEQNO\}) \vee (0 > LSP[lsp.ID].SEQNO - N \wedge lsp.SEQNO \notin \{0, \ldots, LSP[lsp.ID].SEQNO\} \cup \{LSP[lsp.ID].SEQNO - N \bmod SEQbound, \ldots, SEQbound - 1\})$. This allows every node to ignore stale messages that might arrive from intermediate nodes, and yet within $SEQbound$ synchronous rounds, p_j stores $LSP_i[i]$'s value in $LSP_j[i]$, where $p_j \in \mathcal{P}$.

Question 2. Please offer a self-stabilizing solution to the problem addressed by the Link State algorithm within the context of an asynchronous shared memory network, with a general graph topology and fair execution.

- Your solution should encompass its own pseudocode, which may leverage concepts and algorithms learned in class. In that case, please cite them clearly using the exact book chapter and section numbers.
- Additionally, provide a self-contained correctness proof, referencing relevant theorems and lemmas explicitly if utilized.

The answer idea

One can add the field of local topology to the tuples of the update algorithm. Please find the code in the lecture slides. There is a need to modify lines 5 to 7. Specifically, in line 7, there is a need to change the tuple from $\langle i, 0\rangle$ to $\langle i, 0, N(i)\rangle$. In line 5, modify from $\langle i, *\rangle$ to $\langle i, *, *\rangle$. In line 6, from $\langle *, 1\rangle$ to $\langle *, 1, *\rangle$. Create a layer above the update algorithm. That layer takes the value of Processorsi and constructs the global topology since it knows the neighborhood of every processor in the system. The correctness proof is based on the correctness of the update algorithm and the theorem of fair composition.

Question 3. Kindly conduct a comparative analysis of your solutions to Questions 1 and 2. Evaluate and contrast the respective merits and advantages/drawbacks of each solution.

The answer idea

Both approaches are self-stabilizing. However, the one in Question 2 uses an optimal number of LSP transmissions, i.e., $O(n)$, and stabilizes within $O(d)$ synchronous rounds, whereas the approach in Question 1 uses $O(Dn^2)$ transmissions and stabilizes within $O(N^2)$ synchronous rounds.