Home assignment for Computer Network EDA387

Group 24

October 3, 2025

Problem Consider an asynchronous computer network with a distributed but fair scheduler, i.e., not a central daemon. The network 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 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 own 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., $\text{right}_i = \perp$).

Proof. Because of each processor has its own register, and in the register, it can be divided into fields, we will use this to store the important information as in the pseudo-code below.

```
00 do forever
01   if not hasLeft:
02       $left_i$ := 0
03   else
04       $left_i$ := min($left_j$ + 1, N)
05   if not hasRight:
06       $right_i$ := $left_i$ + 1
07   else
08       $right_i$ := $right_k$
```

Let consider the left field in the register is for the relative distance from the processor to the leftmost, while the right field is for the number of the processors. In this problem, this code uses 2 floods to update the correct value for a processors. One from the leftmost, carries the relative distance from leftmost processor to P_i. The other comes from rightmost, carries the information about the total number of processors in this set.

- **Legal execution and legal configuration** The legal configuration for this system is, in the register s_i, the left field is for relative distance to the leftmost and the right field is for the total number of processors in this system. Within the $doforever$, these processors will keep the same values for these 2 fields, hence, guaranted the stabilization. From this state, only legal executions can be started.

- **Correctness** this algorithm will flood the value from the leftmost to the rightmost. Because it takes exactly n steps from leftmost to rightmost, at the end of block $01 - 04$ (the left flood), each processor will know its relative distance to the leftmost, and this will remain unchanged (because the value is the same after the stabilization). As the same with block $05 - 08$, the rightmost will assign the value of n and will flood it to the leftmost. Because these 2 processes begin from different directions and write to different fields, it is guarantee that the values are correct.

- **Lemma for left convergence** if a processor P_i has a value $left_i$ after several asynchronuous rounds. Because the leftmost always set its $left$ to 0, it will be stabilized after at most n rounds. Hence, the complexity of this algorithm is $O(n)$ for the left convergence.

- **Lemma for right convergence** because the rightmost keeps setting its *right* to the value of its $left$, if a register $right_i$ has transient fault, it will eventually adopt the value n after at most n rounds. Same as left convergence, complexity is $O(n)$.

- **Time** at most n cycles, all these processors will be in the safe configuration state.

For the question: *Can a naming self-stabilization algorithm exist on such a system*
The answer is no without a distinguished node. As stated in "Self stabilization" (Angluin 1980, Dolev): "Naming in anonymous, symmetric networks without unique inputs is impossible. Because in this system, processor can not know the total number of nodes in this system, therefore, the total number of unique labels for naming is unknown.
For a system that is similar to question 1, we can see that:

- Every processors run the same deterministic program.

- Processors are initially in arbitrary state (require self-stabilizing).

- The topology is symmetric, all processors have identical view (only knows left of right neighbor).

But, we still can find an algorithm under some conditions. We can first use the self-stabilizing algorithm for all processors, once the system is in legal configuration, a leader elective algorithm can be run to determine the distinguished processor. With these conditions, the symmetry can be broken and a self-stabilizing naming algorithm can exist.

$\square$