Brief Announcement: Optimal Self-stabilizing Multi-token Ring: A Randomized Solution

Andrew Berns
The University of Iowa
Iowa City, Iowa, USA
adberns@cs.uiowa.edu

Anurag Dasgupta
The University of Iowa
Iowa City, Iowa, USA
adasgupt@cs.uiowa.edu

Sukumar Ghosh
The University of Iowa
Iowa City, Iowa, USA
ghosh@cs.uiowa.edu

ABSTRACT

The token ring is a seminal topic in self-stabilization research. It has been expanded to include multiple tokens, and improved upon using randomization to lower the state space requirements. In this brief announcement, we discuss how the two ideas can be brought together for an optimal solution to the multi-token ring problem.

Categories and Subject Descriptors

C.2.4 [**Computer System Organization**]: Computer Communication Networks—*Distributed Systems*

General Terms

Algorithms

Keywords

Multi-token ring, self-stabilization

1. INTRODUCTION

The token ring has been a staple of self-stabilization research since it was used by Edsger Dijkstra to introduce the notion of self-stabilizing algorithms [1].

One nontrivial solution to the original self-stabilizing token ring problem is using randomization to reduce the state space requirements of a single-token ring from $O(n)$ to $O(1)$ [4]. Gouda and Haddix proposed a deterministic solution to the same problem with three bits per process [3]. An extension of the original token ring problem is a multi-token ring: a ring with multiple tokens circulating at the same time, and [2] presents a solution for an l-token ring with $O(ln)$ space per process. In this paper, we synthesize these two algorithms. Specifically, we use a randomization to build an *optimal* solution for a self-stabilizing multi-token ring.

2. MODEL

Consider n processes with identification numbers from 0 to $n-1$ arranged in a unidirectional ring. Each process i is capable of reading the state of its only neighbor. For processes $0 < i < n$, this neighbor is $(i-1)$, while process 0's neighbor is process $n-1$. Process 0 is a special process and executes a special program, while the remaining $n-1$ processes execute the same program which differs from the program of process 0. A *configuration* is a set of all processes' local states at a particular point in time. Each step taken by a process i is a guarded action $g \rightarrow A$, where g is a predicate (involving the state of processes i and i's neighbor), and A is an action (that updates the state s_i of process i). Action A may be executed when g is true for some process in the system. In case multiple processes have enabled guards, a weakly fair scheduler arbitrarily chooses any one of them to execute an action. A *computation* is a sequence (finite or infinite) of configurations – each configuration is reachable from the preceding configuration by executing some enabled action of a single process.

A system is self-stabilizing, if, starting from an arbitrary initial system configuration, every computation leads the system to a *valid* configuration (convergence property), and maintains the configuration thereafter (closure property).

3. PROBLEM

In the original problem, a process with an enabled guard is said to hold a *token*. For a self-stabilizing single token ring, two conditions must hold: (1) Exactly one process must have a token at any time, and (2) in an infinite computation, every process must have a token infinitely often. For a self-stabilizing l-token ring, only the first condition has to be modified: in a valid configuration, there should be a total number of l tokens in the ring ($l \geq 1$).

With this generalization, the relationship between an enabled guard and the number of tokens held by a process has to be redefined, which is somewhat different from that in [2]. We share the fairness requirement that each process must execute infinitely often in an infinite computation.

4. A FIRST ATTEMPT

Perhaps the most intuitive solution is to begin with Herman's randomized protocol for a single-token ring [4] that uses a three-valued state variable for each process, and "glue" together l such single token rings. Each process maintains l ternary variables instead of one, and stores these variables in the array s_i, with the jth element represented as $s_i[j]$. For each process $i > 0$, a token is present in position j if $s_i[j] \neq s_{i-1}[j]$ (for process 0, this condition is modified to $s_0[j] = s_{n-1}[j]$). The total number of tokens in the system is simply the sum of the number of tokens for each of the l positions. The algorithm for this l-token ring is given in algorithm 1.

PODC'09, August 10–12, 2009, Calgary, Alberta, Canada.
ACM 978-1-60558-396-9/09/08.

Algorithm 1 A simple self-stabilizing l-token ring

Let γ be a random number $\in \{0, 1, 2\}$
$0 < i < n \wedge \exists j \mid s_i[j] \neq s_{i-1}[j] \rightarrow s_i[j] := s_{i-1}[j]$
$i = 0 \wedge \exists j \mid s_0[j] \neq s_{n-1}[j] \rightarrow s_0[j] := \gamma$

The proof of correctness for this algorithm follows from that of the original randomized single token protocol [4]: since each component ring is independently self-stabilizing, the system will stabilize to a configuration with exactly l tokens in it.

This simple algorithm requires, for each token, 3 states per process, resulting in a per-process space requirement of 3^l. As we show next, this solution is not optimal.

5. AN OPTIMAL SOLUTION

It is possible to maintain an l-token ring with $l+1$ states per process, which we show below to be optimal. As before, let each process i store a single variable, s_i, although now the value of s_i ranges from 0 to l. For processes 1 through $n-1$, we define the number of tokens present at each process as simply $(s_{i-1} - s_i)$ if $s_{i-1} \geq s_i$, and $(l - s_i + s_{i-1} + 1)$ otherwise. For process 0, the number of tokens present is equal to $(l - (s_0 - s_{n-1}))$ if $s_0 \geq s_{n-1}$, and $(s_{n-1} - s_0 - 1)$ otherwise. The program for a self-stabilizing l-token ring using these $l+1$ states is given in algorithm 2.

Algorithm 2 Optimal Self-stabilizing l-token Algorithm

Let γ be a random number $\in \{0, 1, \ldots, l\}$
$0 < i < n \wedge s_i \neq s_{i-1} \rightarrow s_i := s_{i-1}$
$i = 0 \wedge s_0 = s_{n-1} \rightarrow s_0 := \gamma$

With s_i ranging from 0 to l, it is easy to see the space complexity of algorithm 2 is $l+1$ per process, independent of the number of processes.

LEMMA 1. *A space complexity of $l+1$ states per process is optimal for an l-token ring.*

Each process must be in a unique state to reflect its token content. Since a process can hold any number of tokens between 0 and l, at least $l+1$ states are required for each process. Therefore, the given algorithm provides the optimal space complexity for the l-token ring problem.

LEMMA 2. *The l-token algorithm presented in algorithm 2 is deadlock-free.*

For the program to be deadlocked, all guards must be false - that is, $s_0 \neq s_{n-1} \wedge \forall i \neq 0 : s_i = s_{i-1}$. This is a contradiction, as $s_1 = s_0 \wedge s_2 = s_1 \wedge ... \wedge s_{n-1} = s_{n-2} \wedge s_0 \neq s_{n-1}$ is false.

5.1 Proofs of Closure and Convergence

LEMMA 3. *A valid configuration with l tokens is closed under the actions of algorithm 2.*

A proof sketch for lemma 3 follows. Begin with a valid configuration containing exactly l tokens. By lemma 2, we know that in all configurations, at least one process has an enabled guard. There are then two cases to consider: one for process 0, and one for the remaining processes.

In the first case, process 0 has an enabled guard (i.e. $s_{n-1} = s_0$), meaning process 0 currently has all l tokens, and $\forall i : s_i = q$, where q is an integer ranging from 0 to l. Process 0's action will set s_0 from q to a new value p, which can either be greater than or less than q.

Case 1a: $p > q$: number of tokens at process 0 becomes $(l - (p - q))$, and the number of tokens at process 1 becomes $p - q$. All other processes' token counts remain the same. The total number of tokens in the system can thus be written as $(l - (p - q)) + (p - q) = l$.

Case 1b: $p < q$: The number of tokens at process 0 becomes $q - p - 1$, while the number of tokens at process 1 becomes $(l - q + p + 1)$. The total number of tokens in the system is then $(q - p - 1) + (l - q + p + 1) = l$.

Now consider the second case where a process $i \neq 0$ has a guard enabled and executes its action, changing its value from s_i to s'_i, where $s'_i = s_{i-1}$. It is easy to verify that regardless of the relative values of s_i and s_{i+1}, the number of tokens lost by process i is equal to the number of tokens gained by process $i+1$.

LEMMA 4. *Beginning from an arbitrary initial state, algorithm 2 will eventually reach a valid configuration, with exactly l tokens being present in the system.*

Consider a computation σ, with a suffix where each of $n+1$ consecutive actions by process 0 results in a value of $s_0 \neq k$, where $0 \leq k \leq l$. In any computation where the value of s_0 is randomly chosen from the set $\{0, 1, 2, \cdots, l\}$, such a suffix must exist. Since each action by a process $i \neq 0$ copies the state from process $(i-1)$, after the suffix of the above computation, no process will have its state equal to k. Eventually thereafter, a random step by process 0 will lead to $s_0 = k$. Each process moves infinitely often, and the next move by process 0 is possible only after $s_{n-1} = k$. However, by then, $\forall i : 0 \leq i \leq n-1$, $s_i = k$, which implies a legal configuration, with process 0 holding all l tokens, and no other process having a token.

Lemmas 3 and 4 lead to theorem 1.

THEOREM 1. *Algorithm 2 is self-stabilizing.*

6. CONCLUSION

The proposed algorithm, in its present form, does not allow a process $i \neq 0$ to partially release its tokens. Also, the analysis currently considers only a central scheduler. Whether the algorithm retains the self-stabilization property in spite of the partial release of tokens by processes $i \neq 0$ and whether the algorithm is correct under a distributed scheduler are currently under investigation.

7. REFERENCES

[1] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[2] M. Flatebo, A. K. Datta, and A. A. Schoone. Self-stabilizing multi-token rings. *Distrib. Comput.*, 8(3):133–142, 1995.

[3] M. G. Gouda and F. F. Haddix. The stabilizing token ring in three bits. *J. Parallel Distrib. Comput.*, 35(1):43–48, 1996.

[4] T. Herman. Self-stabilization: randomness to reduce space. *Distrib. Comput.*, 6(2):95–98, 1992.