Self-Stabilizing Mutual Exclusion Under Arbitrary Scheduler[*]

Ajoy K. Datta[1] Maria Gradinariu[2] Sébastien Tixeuil[3†]

[1] School of Computer Science, University of Nevada Las Vegas
[2] IRISA, Campus de Beaulieu, France
[3] LRI-CNRS UMR 8623, Université Paris Sud, France

Abstract

A self-stabilizing algorithm, regardless of the initial system state, converges in finite time to a set of states that satisfy a legitimacy predicate. The mutual exclusion problem is fundamental in distributed computing, since it permits processors that compete to access a shared resource to be able to synchronize and get exclusive access to the resource (*i.e.* execute their critical section).

It is well known that providing self-stabilization in general uniform networks (*e.g.* anonymous rings of arbitrary size) can only be probabilistic. However, all existing uniform probabilistic self-stabilizing mutual exclusion algorithms designed to work under an unfair distributed scheduler (that may choose processors to execute their code in an arbitrary maneer) suffer from the following common drawback: Once stabilized, there exists no upper bound on time between two successive executions of the critical section at a given processor. In this paper, we present the first self-stabilizing algorithm that guarantees such a bound ($O(n^3)$, where n is the network size) while working using an unfair distributed scheduler. Our algorithm works in an anonymous unidirectional ring of any size and has a polynomial expected stabilization time.

Keywords: Distributed algorithm, self-stabilization, mutual exclusion, unfair scheduler, service time.

1 Introduction

Mutual Exclusion. The mutual exclusion is a fundamental problem in the area of distributed computing. Consider a distributed system of n processors. Every processor, from time to time, may need to execute a critical section in which exactly one processor is allowed to use some shared resource. A distributed system solving the mutual exclusion problem must guarantee the following two properties:

1. *Mutual Exclusion*: Exactly one processor is allowed to execute its critical section at any time.

2. *Fairness*: Every processor must be able to execute its critical section infinitely often.

[*]An extended abstract of this paper appeared in [4]

[†]This author was supported in part by the french project MobiCoop.

Self-stabilization. The concept of self-stabilization was first introduced by Edsger W. Dijkstra in 1974 [5]. It is now considered to be the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system guarantees that starting from an arbitrary state, the system converges to a legal configuration in a finite number of steps, and remains in a legal state until another fault occurs (see also [6]).

In the context of computer networks, resuming correct behavior after a fault occurs can be very costly [15] — the whole network may have to be shut down and globally reset in a good initial state. While this approach is feasible for small networks, it is far from practical in large networks such as the Internet. Self-stabilization provides a way to recover from faults without the cost and inconvenience of a generalized human intervention: after a fault is diagnosed, one simply has to remove, repair, or reinitialize the faulty components, and the system, by itself, will return to a good global state within a relatively short amount of time.

Scheduler. All components (processors and communication links) of distributed systems may not share the same speed assumptions (*i.e.*, one processor may execute its code fast, while many others are very slow). The scheduler is a way to model such different behaviors. A scheduler chooses processors to execute their code at a given time. If the scheduler is given more freedom (power) to make its selections, then the task of designing an algorithm to cope with this scheduler becomes more challenging. In other words, the scheduler works as an adversary against the algorithm ([2, 7]). The *synchronous scheduler* is one of the simplest (or weakest) schedulers — in every computation step, all processors are allowed to execute their code in lock-step. This scheduler models systems where all processors run (almost) at the same speed. The *k-bounded scheduler* may choose enabled processors in such a way that the ratio of speeds between any two processors is at most k. This scheduler models a situation where one processor is at most k times faster than another. So, the k-bounded scheduler is a stronger adversary than the synchronous one. The *arbitrary scheduler* represents the strongest possible adversary. It simply can arbitrarily choose any enabled processors.

Related Work. Dijkstra's three self-stabilizing mutual exclusion algorithms [5] are deterministic and *non-uniform* (in such an algorithm, some processors are distinguished in the sense that they are allowed to execute a program that is different from that of the other processors). In [3], Burns and Pachl presented a deterministic algorithm for uniform unidirectional rings of prime size, and proved that no deterministic solution exists for rings of composite size.

Several papers investigated the mutual exclusion problem in the probabilistic (or randomized) setting. Randomization was used to reduce the space in [9, 12], and to deal with anonymous networks in [1, 11]. However, a common problem in all these probabilistic algorithms is that once stabilized, there is no upper bound on the time between two entries into the critical section at a particular processor. In other words, although the expected time between two critical section executions is bounded, there exist computations in which a particular processor may not get the token infinitely often. We refer to this kind of algorithms as *weak probabilistic stabilizing* algorithms. Kakugawa and Yamashita [14] presented a probabilistic uniform self-stabilizing algorithm on uniform rings that does guarantee an upper bound between two critical section entries. We call this class of algorithms *strong probabilistic stabilizing* algorithms. However, the algorithm of [14] works *only* under the *central scheduler* (which allows exactly one enabled processor at any time). All previously known algorithms solving the mutual exclusion problem ensure fairness using one of the two well-known methods: *(i)* by choosing an *ad hoc* scheduler (e.g., the fair scheduler in [9] or randomized central scheduler in [12]) and *(ii)* by requiring that the correctness of the system is probabilistic (as in [1] and [11]). The open question in [14] was to design a strong probabilistic stabilizing algorithm that solves the mutual exclusion problem under an unfair distributed scheduler.

Our Contributions. We answer the open question of [14] and provide a *strong probabilistic stabilizing* algorithm for the mutual exclusion problem in an anonymous unidirectional ring of any size running under an *unfair distributed scheduler*. (The distributed scheduler selects an arbitrary non-empty subset of enabled processors in a computation step at any time.) We describe the probabilistic self-stabilizing systems in Section 3. We start with a *strong probabilistic algorithm* that works under a synchronous scheduler—all processors are activated simultaneously. This first algorithm is derived from the space-optimal *weak probabilistic* algorithm of [1]. Then we transform it to a strong probabilistic stabilizing algorithm to work under a k-bounded scheduler (that bounds the ratio of relative speeds of executions of any two processors to k). Finally, we use the composition technique described in [2] to stabilize the algorithm under an unfair distributed scheduler (Section 4). We show that the maximum expected stabilization time is $O(n^3)$ under the unfair and k-bounded scheduler, and $O(n^2)$ under the synchronous scheduler. After stabilization, the upper bound between two occurrences of the privilege at a given processor is $O(n^3)$ under the unfair scheduler, $O(kn)$ under the k-bounded scheduler, and $O(n)$ for the synchronous scheduler.

Outline. In Section 2, we present the underlying model for our algorithms. We also define properties related to self-stabilization in the context of probabilistic systems. We present three algorithms and their correctness proofs in Section 4. The complexity results of all algorithms are provided in Section 5. Concluding remarks are made in Section 6.

2 Model

Distributed Systems We model a distributed system $\mathcal{S} = (C, T, I)$ as a *transition system* where C is the set of system configurations, T is a transition function from C to C, and I is the set of initial configurations. A *probabilistic distributed system* is a distributed system where a probabilistic distribution is defined on the transition function of the system.

We consider unidirectional ring networks where the processors maintain two types of variables: *local variables* and *field variables*. Each processor, P_i, has two neighbors denoted by $left_i$ (the counter-clockwise neighbor of P_i) and $right_i$ (the clockwise neighbor of P_i). The local variables of P_i cannot be accessed by any of its neighbors, whereas the field variables of P_i are part of the shared register which is used to communicate with P_i's right neighbor. A processor can write only into its own shared register and can read only from the shared registers owned by its left neighbor or itself. The *state* of a processor is defined by the values of its local and field variables. A processor may change its state by executing its local *algorithm* (defined below). A *configuration* of a distributed system is an instance of the state of its processors.

The *algorithm* executed by each processor is described by a finite set of guarded actions of the form $\langle\text{guard}\rangle \longrightarrow \langle\text{statement}\rangle$. Each guard of processor P_i is a boolean expression involving P_i's variables and $left_i$'s field variables. A processor P_i is *enabled* in configuration c if at least one of the guards of the program of P_i is *true* in c. Let c be a configuration and CH be a subset of enabled processors in c. We denote by $\{c : CH\}$ the set of configurations that are *reachable* from c if every processor in CH executes an action starting from c. A *computation step* is a tuple (c, CH, c'), where $c' \in \{c : CH\}$. Note that all configurations $\in \{c : CH\}$ are reachable from c by executing *exactly one* computation step. In a probabilistic distributed system, every computation step is associated with a probabilistic value (the sum of the probabilities of the computation steps determined by $\{c : CH\}$ is 1). A *computation* of a distributed system is a maximal sequence of computation steps. A *history* of a computation is a finite prefix of the computation. A history of length n (denoted by h_n) starting with the initial configuration c_0 can be defined recursively as follows:

$$h_n \equiv \begin{cases} (c_0, CH_0, c_1) & \text{if } n = 1 \\ [h_{n-1}(c_{n-1}, CH_{n-1}, c_n)] & \text{otherwise} \end{cases}$$

The probabilistic value of a history is the product of the probabilities of all the computation steps in the history. If h_n is a history such that

$$h_n = [(c_0, CH_0, c_1) \ldots (c_{n-1}, CH_{n-1}, c_n)]$$

then we use the following notations: the length of the history h_n (equal to n) is denoted as $length(h_n)$, the last configuration in h_n (which is c_n) is represented by $last(h_n)$, and the first configuration in h_n (which is c_0) is referred to as $first(h_n)$ ($first$ can also be used for an infinite computation). A *computation fragment* is a finite sequence of computation steps. Let h be a history, x be a computation fragment such that $first(x) = last(h)$, and e be a computation such that $first(e) = last(h)$. Then $[hx]$ denotes a history corresponding to the computation steps in h and x, and (he) denotes a computation containing the steps in h and e.

3 Probabilistic Systems

In this section, we give an outline of the probabilistic model used in the rest of the paper. A detailed description of this model is available in [2].

Scheduler. A *scheduler* is a *predicate* over the system computations. In a computation, a transition (c_i, c_{i+1}) occurs due to the execution of a nonempty subset of the enabled processors in configuration c_i. In every computation step, this subset is chosen by the scheduler. The interaction between a scheduler and the distributed system generates some special structures, called *strategies*. The scheduler strategy definition is based on the tree of computations (all the computations having the same initial configuration). Let c be a system configuration and S a distributed system. The tree representing all computations in S starting from the configuration c is the tree rooted at c and is denoted as $\mathcal{T}ree(S, c)$. Let n_1 be a configuration in $\mathcal{T}ree(S, c)$. A *branch* originating from n_1 represents the set of all $\mathcal{T}ree(S, c)$ computations starting in n_1 with the same first transition. The degree of n_1 is the number of branches rooted in n_1.

Definition 3.1 (Strategy) *Let S be a distributed system, D a scheduler, and c a configuration in S. We define a strategy w.r.t. D as the set of computations represented by the tree obtained by pruning*

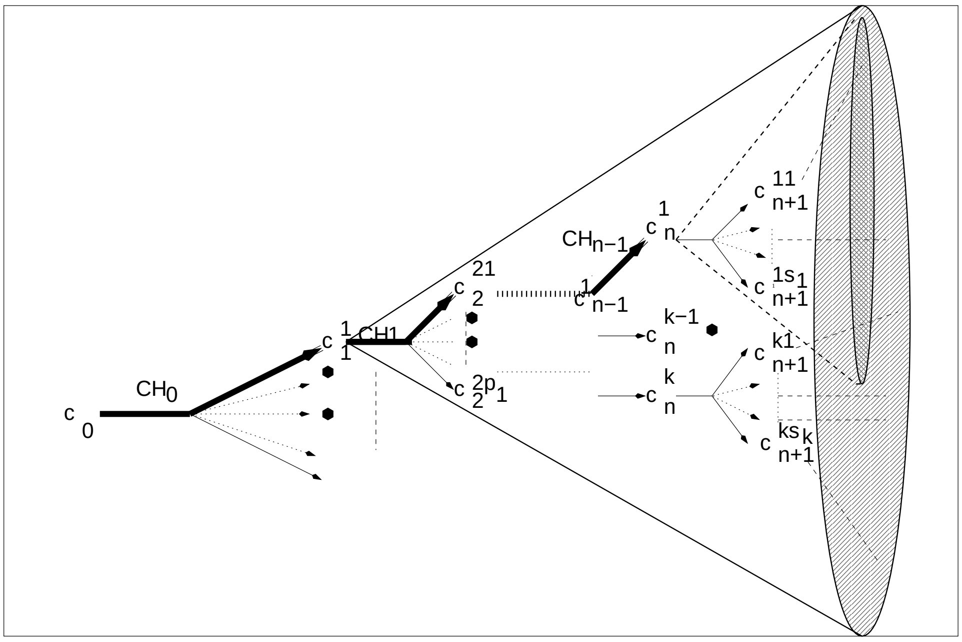

Figure 1: **Cone and subcone in a strategy.**

$\mathcal{T}ree(S, c)$ such that the degree of any processor is at most 1, and any execution in the pruned tree satisfies D.

The following definition introduces the notion of cone of execution which can be seen intuitivelly as a branch in a strategy (see Figure 1).

Definition 3.2 (Cone) *Let s be a strategy of a scheduler D. A* cone *$\mathcal{C}_h(s)$ corresponding to a history h is defined as the set of all possible computations under D which create the same history h.*

For example, Figure 1 presents a cone $\mathcal{C}_h(s)$ whose history h is equal to $[(c_0, CH_0, c_1^1)]$.

The probabilistic value of a cone $\mathcal{C}_h(s)$ is the probabilistic value of the history h (*i.e.*, the product of the probabilities of all computation steps in h).

Definition 3.3 (Subcone) *A cone $\mathcal{C}_{h'}(s)$ is called a* subcone *of $\mathcal{C}_h(s)$ if and only if $h' = [hx]$, where x is a computation fragment.*

For example, Figure 1 presents a cone $\mathcal{C}'_{h'}(s)$ whose history h' is equal to

$$[(c_0, CH_0, c_1^1) \dots (c_{n-1}^1, CH_{n-1}, c_n^1)]$$

Since $h' = [hx]$ (with x equal to $[\dots(c_{n-1}^1, CH_{n-1}, c_n^1)]$), $\mathcal{C}'_{h'}(s)$ is a subcone of $\mathcal{C}_h(s)$.

Let S be a system, D a scheduler, and s a strategy of D. The set of computations under D that reach a configuration c' satisfying predicate P (denoted as $c' \vdash P$) is denoted as $\mathcal{EP}_s$, and its associated probabilistic value is represented by $Pr(\mathcal{EP}_s)$. We call a predicate P a *closed predicate* if the following is true: If P holds in configuration c, then P also holds in any configuration reachable from c.

Probabilistic Self-Stabilizing Systems. A probabilistic self-stabilizing system is a probabilistic distributed system satisfying two important properties: *probabilistic convergence* (the probability of the system to converge to a configuration satisfying a *legitimacy predicate* is 1) and *correctness* (once the system is in a configuration satisfying a legitimacy predicate, it satisfies the system specification). In this context, the correctness comes in two variants: *weak correctness* (the system correctness is only probabilistic) and *strong correctness* (the system correctness is certain).

Definition 3.4 (Strong Probabilistic Stabilization) *A system S is* strong self-stabilizing *under scheduler D for a specification SP if and only if there exists a closed legitimacy predicate L such that in any strategy s of S under D, the two following conditions hold:*
(i) The probability of the set of computations under D, starting from c, reaching a configuration c', such that c' satisfies L is 1 (probabilistic convergence). (Formally, $\forall s, Pr(\mathcal{EL}_s) = 1$).
(ii) All computations, starting from a configuration c' such that c' satisfies L, satisfy SP (strong correctness).(Formally, $\forall s, \forall e, e' \in s, e = (he') ::\ last(h) \vdash L \Rightarrow e' \vdash SP$).

Convergence of Probabilistic Stabilizing Systems. We borrow a result of [2] to prove the probabilistic convergence of the algorithms presented in this paper. This result is built upon some previous work on probabilistic automata ([16, 17, 18, 19]) and provides a complete framework for the verification of self-stabilizing probabilistic algorithms. We need to introduce a few terms before we are ready to present this result. First, we explain a key property, called *local convergence* and denoted by LC. Informally, the LC property characterizes a probabilistic self-stabilizing system in the following way: The system reaches a configuration which satisfies a particular predicate, in a bounded number of computation steps with a positive probability.

Definition 3.5 (Local Convergence) *Let s be a strategy, and P_1 and P_2 be two predicates on configurations, where P_1 is a closed predicate. Let δ be a positive number $\in]0,1[$ and N a positive integer. Let $\mathcal{C}_h(s)$ be a cone with $last(h) \vdash P_1$ and let M denote the set of subcones $\mathcal{C}_{h'}(s)$ of $\mathcal{C}_h(s)$ such that $last(h') \vdash P_2$ and $length(h') - length(h) \leq N$. Then $\mathcal{C}_h(s)$ satisfies the local convergence property denoted as $LC(P_1, P_2, \delta, N)$ if and only if $Pr(\bigcup_{\mathcal{C}_{h'}(s) \in M} \mathcal{C}_{h'}(s)) \geq \delta$.*

Now, if in strategy s, there exist $\delta_s > 0$ and $N_s \geq 1$ such that any cone $\mathcal{C}_h(s)$ with $last(h) \vdash P_1$ satisfies $LC(P_1, P_2, \delta_s, N_s)$, then the result of [2] states that the probability of the set of computations under D reaching configurations satisfying $P_1 \wedge P_2$ is 1. Formally:

Theorem 3.1 ([2]) *Let s be a strategy. Let P_1 and P_2 be closed predicates on configurations such that $Pr(\mathcal{EP}1_s) = 1$. If $\exists \delta_s > 0$ and $\exists N_s \geq 1$ such that any cone $\mathcal{C}_h(s)$ with $last(h) \vdash P_1$ satisfies $LC(P_1, P_2, \delta_s, N_s)$, then $Pr(\mathcal{EP}12_s) = 1$, where $P_{12} = P_1 \wedge P_2$.*

4 Strong Probabilistic Stabilizing Mutual Exclusion

In this section, we present three solutions to the mutual exclusion problem for three different schedulers: synchronous scheduler (Section 4.1), k-bounded scheduler (Section 4.2) and arbitrary scheduler (Section 4.3).

Specification of the Mutual Exclusion Problem We specify the mutual exclusion problem ($\mathcal{SP}_{ME}$) as follows: There is exactly one privilege in the system at any time and every processor obtains the privilege infinitely often.

4.1 Synchronous Scheduler

The algorithm for mutual exclusion under the synchronous scheduler is presented as Algorithm 4.1.

Algorithm 4.1 Mutual exclusion under a synchronous scheduler (for p).

```
Field variables:
    $t_p \in [0, mnd(n) - 1]$ (the privilege.)
Variables:
    $go\_ahead_p \in \{pass, wait\}$.
    $rand\_bool_p$ holds a random value in $\{1, 0\}$. Each value has a probability of 1/2.
Predicate:
    $Privilege(p) \equiv t_p - t_{lp} \neq 1 \bmod mnd(n)$
Macro:
    $Pass\_privilege(p) : t_p := (t_{lp} + 1) \bmod mnd(n)$
Actions:
$\mathcal{A}_1$:: $Privilege(p) \wedge go\_ahead_p$=wait $\longrightarrow$
    if ($rand\_bool_p$ = 1) then $go\_ahead_p$=pass;
    else $Pass\_privilege(p)$;
$\mathcal{A}_2$:: $Privilege(p) \wedge go\_ahead_p$=pass $\longrightarrow$
    $Pass\_privilege(p)$;
    if ($rand\_bool_p$ = 0) then $go\_ahead_p$=wait;
```

In Algorithm 4.1, every processor p in the system has a field variable t_p. A processor is *privileged* if and only if the difference between t_p and t_{lp} (the t_p variable of its left neighbor) is not 1. It was proven in [1] that if operations on t_p variables are made always *modulo* $mnd(n)$[1], where n is the number of processors in the ring, then at least one privilege is always present in the ring.

A privileged processor tosses a coin to decide whether it wants to pass the privilege or not. If it decides to keep the privilege, it can do so for one more computation step. The purpose of the go_ahead_p variable is to prevent processor p from keeping the token too long. This allows the algorithm to achieve an upper bound on the service time (the time between two consecutive grants given to processor p to enter its critical section).

We define the legitimacy predicate $\mathcal{L}_{ME}$ as follows:

$$\mathcal{L}_{ME} \quad \equiv \quad \text{There exists exactly one privilege.}$$

Correctness proof. In this subsection, we will show that every processor enjoys the privilege infinitely often.

Lemma 4.1 *Starting from any legitimate configuration, each node is privileged once in at most every* $2 \times n$ *computation steps, where* n *is the size of the ring.*

[1] $mnd(n)$ denotes the minimum non-divisor of n. For example, $mnd(5) = 2$.

Proof: Let c be a legitimate configuration. Let p_1 be the processor holding the privilege in c. If the value of p_1's local variable go_ahead equals *wait*, then p_1 executes Action $\mathcal{A}_1$. A coin is tossed with two possible outcomes:

1. p_1 changes go_ahead to *pass*, and in the next computation step, Action $\mathcal{A}_2$ is executed, making p_1 to pass the privilege to its right neighbor.

2. p_1 passes the privilege immediately to its right neighbor.

Therefore, within at most two computation steps, the privilege is passed from p_1 to p_2. Since the size of the ring is n, p_1 gets privileged again within $2 \times n$ computational steps. $\Box$

Corollary 4.1 *The privilege circulates infinitely often in the ring.*

Proof of Convergence In order to prove the probabilistic convergence of Algorithm 4.1, we first show that for any strategy s, all cones of s satisfy the local convergence property. Then, by Theorem 3.1, the probabilistic convergence of the system will be established.

Notation 4.1 *Let* $\mathrm{Priv}(c)$ *denote the number of privileged processors in configuration c.*

Lemma 4.2 *Let s be a strategy of Algorithm 4.1 under a synchronous scheduler starting from configuration c. There exist* $\delta > 0$ *and* $N \geq 1$ *such that any cone of s satisfies*

$$LC(\text{true}, \mathcal{L}_{ME}, \delta, N)$$

Proof: The proof is done using the following two steps:

1. In strategy s, there exists a cone $\mathcal{C}_{h1}(s)$, such that configuration $last(h1)$ satisfies the property:
$$Priv(last(h1)) \leq Priv(c) - 1$$

2. Repeat the argument of (1) by considering $\mathcal{C}_{h1}(s)$ as the current strategy, until the number of privileges becomes equal to 1.

Proposition 4.1 *Let s be a strategy. There exist* $\delta_1 > 0$, $N_1 \geq 1$, *and a cone* $\mathcal{C}_{h1}(s)$ *such that*

$$Pr(\mathcal{C}_{h1}(s)) \geq \delta_1$$

$$length(h1) \leq N_1$$

and

$$\mathrm{Priv}(last(h1)) \leq \mathrm{Priv}(c_0) - 1$$

Proof: Assume that $Priv(c_0) = m$. Let us number the privileges $(t_i)_{i=1,m}$ clockwise such that

$$dist(t_m, t_1) = min_{1 \leq i \leq m-1} dist(t_i, t_{i+1})$$

where $dist(p, q)$ is the distance from p to q measured following the clockwise direction. Let d_i be the distance in c_0 between t_i and t_{i+1}. We want to prove that the probability that the distance between the privileges t_m and t_1 decreases, is strictly positive. Intuitively, this means that there is a positive probability that the "speed" of privilege t_m increases as it approaches t_1. We consider privileges t_m, t_1, and t_2, and calculate the probability of t_2 leaving a go_ahead variable equal to *wait*, and that of t_1 leaving a go_ahead variable equal to *pass*. The worst initial configuration for our scenario is when every processor p between t_m and t_1 has $go\_ahead_p = wait$, and every processor p' between t_1 and t_2 has $go\_ahead_{p'} = pass$. We will show that even then, there is a positive probability that t_m approaches t_1. The proof consists of the following two steps:

1. Let us calculate the probability δ_0 to obtain a cone $\mathcal{C}_{h0}(s)$ such that in $last(h0)$, the three following conditions hold: *(1)* t_m reached the processor which held t_1 in c_0, *(2)* t_1 reached the processor which held t_2 in c_0, and *(3)* all the processors visited by t_1 set their variable go_ahead to $pass$ and all processors visited by all other privileges set their variable go_ahead to $wait$. With the above three conditions satisfied, we establish that:
$$\delta_0 = \delta_0^1 \times \delta_0^2 \times \delta_0^3$$
where:
   (a) δ_0^1 is the probability for t_m to reach the processor which held t_1 in c_0.
   $$\delta_0^1 \geq \left(\frac{1}{4}\right)^{d_m}$$
   (b) δ_0^2 is the probability for t_1 to reach the processor which held t_2 in c_0 while all processors visited by t_1 set their go_ahead variable to $pass$.
   $$\delta_0^2 \geq \left(\frac{1}{4}\right)^{d_1}$$
   (c) δ_0^3 is the probability for all processors visited by all other privileges to set their go_ahead variable to $wait$.
   $$\delta_0^3 \geq \left(\frac{1}{4}\right)^{(m-2)max(d_1,2d_m)}$$
   Thus,
   $$\delta_0 \geq \left(\frac{1}{4}\right)^{(m-1)max(d_1,2d_m)+d_m}$$
   In $last(h0)$,
   $$dist(t_m, t_1) \leq 2 \times d_m$$
   and
   $$length(h0) \leq max(2 \times d_m, d_1)$$

2. Let us calculate the probability δ_1 to obtain a subcone $\mathcal{C}_{h1}(s)$ of cone $\mathcal{C}_{h0}(s)$ such that $Priv(last(h1)) \leq Priv(c_0) - 1$. This situation is possible because the probability of t_m to have twice the speed of t_1 is positive. We have:
$$\begin{array}{rcl}\delta_1 & \geq & \delta_0 \times \left(\frac{1}{4}\right)^{(m+2)d_m} \\ & \geq & \left(\frac{1}{4}\right)^{(m-1)max(d_1,2d_m)+(m+3)d_m}\end{array}$$

and

$$\begin{aligned} length(h1) &\leq d_m + length(h0) \\ &\leq (d_m + max(2 \times d_m, d_1)) \end{aligned}$$

Hence

$$\delta_1 \geq \left(\frac{1}{4}\right)^{2\times n^2}$$

and $length(h1) \leq 2n$. In $last(h1)$, we have *Priv*$(last(h1)) \leq m - 1$.

□

Proposition 4.1 still holds for cone $\mathcal{C}_{h1}(s)$. Thus, the probability to obtain a subcone $\mathcal{C}_{h2}(s)$, where *Priv*$(last(h2)) \leq m-2$, is strictly positive. By induction, cone $\mathcal{C}_{hm-1}(s)$ (where *Priv*$(last(hm-1)) = 1$) is obtained with positive probability

$$\delta \geq \left(\frac{1}{4}\right)^{2\times n^3}$$

and

$$length(hm-1) \leq 2 \times n^2$$

□

From Lemmas 4.1 and 4.2, and Theorem 3.1, we claim:

Theorem 4.1 *Algorithm 4.1 is strong probabilistic stabilizing for $\mathcal{SP}_{ME}$ under a synchronous scheduler.*

4.2 k-Bounded Scheduler

In Algorithm 4.2, the variable *go_ahead*$_p$ is now extended to include values between 0 and $2k+1$ (where k is a parameter of the algorithm). When *go_ahead*$_p = (2k+1)$, the processor must pass the privilege and randomly change its value. Otherwise, the processor strictly increases its *go_ahead*$_p$ value. All the values between 0 and $2k$ represent *wait* states and $2k+1$ represents a *pass* state.

Lemma 4.3 (Strong Correctness) *Starting from any legitimate configuration, each processor is privileged within $(2k+2) \times n$ computation steps, where n is the size of the ring.*

Proof: The worst scenario in terms of a privilege being held at processor p is as follows: p has its *go_ahead*$_p = 0$ and every time p is chosen, $rand\_bool_p = 0$. In the worst case, the privilege remains at p for $(2k+2)$ steps (for $2k+1$ steps, *go_ahead*$_p$ is incremented, and finally, in step $2k+2$, the privilege is passed). □

Lemma 4.4 (Probabilistic Convergence) *Let s be a strategy of Algorithm 4.2 under a k-bounded scheduler starting in configuration c. There exist $\delta > 0$ and $N \geq 1$ such that every cone of s satisfies $LC(\text{true}, \mathcal{L}_{ME}, \delta, N)$.*

Proof: In the following, $dist(p,q)$ denotes the distance between Processors p and q, which is equal to the number of the processors between p and q in the clockwise direction, plus 1. Assume that in c, there are m privileges $t_1, \ldots, t_m$, where

$$dist(t_m, t_1) = min_{1\leq i\leq m-1}(dist(t_i, t_{i+1}))$$

Algorithm 4.2 Mutual exclusion under a k-bounded scheduler (for p)

```
Field:
    $t_p \in [0, mnd(n) - 1]$ (the privilege)
Variables:
    $rand\_bool_p$ holds any value in {0, 1}. Each value has a probability of 1/2.
    $go\_ahead_p$ holds any integer value in [0..(2k+1)].
Predicate:
    $Privilege(p) \equiv t_p - t_{lp} \neq 1 \bmod mnd(n)$
Macro:
    $Pass\_privilege(p) : t_p := (t_{lp} + 1) \bmod mnd(n)$
Actions:
$\mathcal{A}_1$:: $Privilege(p) \wedge go\_ahead_p \neq$ (2k+1) $\longrightarrow$
        if ($rand\_bool_p = 1$) then $go\_ahead_p$:=(2k+1) else $go\_ahead_p := go\_ahead_p + 1$;
$\mathcal{A}_2$:: $Privilege(p) \wedge go\_ahead_p$=(2k+1) $\longrightarrow$
        $Pass\_privilege(p)$; if ($rand\_bool_p = 0$) then $go\_ahead_p$:=random(0..2k+1);
```

Let d_i be the distance in c between t_i and t_{i+1}. In the worst case, every processor p between t_m and t_1 is in the "worst" *wait* state ($go\_ahead = 0$), and every processor p' between t_1 and t_2 is in the *pass* state ($go\_ahead_{p'} = 2k + 1$). We now consider the following two cases:

1. We calculate the probability δ_0 to obtain a cone $\mathcal{C}_{h0}(s)$, where $last(h0)$ satisfies the following three properties: *(1)* t_m reached the processor which held t_1 in c, *(2)* t_1 reached the processor which held t_2 in c, and *(3)* all the processors visited by t_1 set their variable go_ahead to $2k + 1$, and the processors visited by all other privileges set their variable go_ahead to 0. Then in $last(h0)$,

$$dist(t_m, t_1) \leq (2k + 2) \times d_m,$$

$$length(h0) \leq max(d_1, (2k + 2)d_m)$$

and

$$\delta_0 = \delta_0^1 \times \delta_0^2 \times \delta_0^3$$

where:

   (a) δ_0^1 represents the probability for t_m to reach the processor which had t_1 in c

$$\delta_0^1 \geq \left(\frac{1}{2}\right)^{(2k+2)d_m}$$

   (b) δ_0^2 represents the probability for t_1 to reach the processor which had t_2 in c, while all processors visited by t_1 set their go_ahead variable to $2k + 1$

$$\delta_0^2 \geq \left(\frac{1}{2} \times \frac{1}{2k + 2}\right)^{d_1}$$

(c) δ_0^3 represents the probability that all processors visited by all other privileges set their go_ahead variable to 0

$$\delta_0^3 \geq \left(\frac{1}{4} \times \frac{1}{2k+2}\right)^{(m-2)max((2k+2)d_m,d_1)}$$

Thus,

$$\begin{array}{rcl} \delta_0 & \geq & \left(\frac{1}{2}\right)^{max((2k+2)d_m,d_1)m} \left(\frac{1}{2k+2}\right)^{d_1+(m-2)max((2k+2)d_m,d_1)} \\ & \geq & \left(\frac{1}{2} \times \frac{1}{2k+2}\right)^{max((2k+2)d_m,d_1)m} \end{array}$$

2. We calculate the probability δ_1 that cone $\mathcal{C}_{h0}(s)$ has a subcone $\mathcal{C}_{h1}(s)$ such that

$$Priv(last(h1)) \leq Priv(c) - 1$$

In order to keep tokens t_m and t_1 as far as possible from each other, the k-bounded scheduler chooses the processors holding t_1 as often as possible. Remark that in order to keep distant the tokens t_m and t_1 the k-bounded scheduler privilegiates the processors holding t_1. By the scheduler definition, anytime the processor holding t_1 is chosen k times, every other processor is chosen at least once. Nevertheless, after $2k+2$ choices of the processor holding t_1, the distance between t_m and t_1 is decreased by 1 (*i.e.* t_1 moves one step forward, while t_m moves two steps forward). Then, after at most $(2k+2)^2 d_m$ choices of t_1, tokens t_m and t_1 merge. Hence, the probability that tokens t_m and t_1 merge and conditions *(2)* and *(3)* be verified is

$$\delta_1 \geq \delta_0 \left(\frac{1}{2}\right)^{2kd_m(2k+2)} \left(\frac{1}{2} \times \frac{1}{2k+2}\right)^{2d_m(2k+2)} \left(\frac{1}{4} \times \frac{1}{2k+2}\right)^{(m-2)d_m(2k+2)}$$

The length of the history $h1$ is

$$\begin{array}{rcl} length(h1) & \leq & length(h0) + d_m \times (2k+2)^2 \\ & \leq & max(d_1, (2k+2) \times d_m) + d_m \times (2k+2)^2 \end{array}$$

and

$$\delta_1 \geq \left(\frac{1}{2}\right)^{2(2k+2)n(n+2(k+1))} \left(\frac{1}{2k+2}\right)^{2(2k+2)n(n+1)}$$

Reapplying (2) for the cone $\mathcal{C}_{h1}(s)$, there is a positive probability to obtain a sub-cone $\mathcal{C}_{h2}(s)$ such that $Priv(last(h2)) \leq m-2$, and then a positive probability δ to obtain a subcone $\mathcal{C}_{hm-1}(s)$, where the number of privileges is 1. We now have:

$$\delta \geq \left(\frac{1}{2}\right)^{2(2k+2)n^2(n+2(k+1))} \left(\frac{1}{2k+2}\right)^{2(2k+2)n^2(n+1)}$$

and

$$length(hm-1) \leq (2k+2)^2 \times n^2$$

$\square$

>From Lemmas 4.3 and 4.4, and Theorem 3.1, we can claim the following result:

Theorem 4.2 *Algorithm 4.2 is strong probabilistic stabilizing for $\mathcal{SP}_{ME}$.*

4.3 Unfair Scheduler

In this section, we extend Algorithm 4.2 for an unfair scheduler. The idea of *cross-over* composition (introduced in [2]) is used to implement this extension. The cross-over composition can be seen as a black box with two algorithms as input (denoted by A and B) and one algorithm as output (denoted by O). The composition goal is to improve Algorithm A with nice properties of Algorithm B. The output Algorithm O solves the same problem as A, but benefits from the properties of algorithm B.

Definition 4.1 (Cross-over Composition) *Let A be an algorithm with n rules as follows:*

$$\forall i \in \{1, \dots, n\}, < guard\ a_i > \Rightarrow < action\ a_i >$$

Let B be an algorithm with m rules as follows:

$$\forall j \in \{1, \dots, m\} < guard\ b_i > \Rightarrow < action\ b_j >$$

The cross-over composition with A and B as entries, is the algorithm with the following rules:
$\forall i \in \{1, \dots, n\}, \forall j \in \{1, \dots, m\}$

$$< guard\ a_i > \wedge < guard\ b_j > \Rightarrow < action\ a_i >; < action\ b_j >$$

and $\forall j \in \{1, \dots, m\}$

$$\neg < guard\ a_i > \wedge \dots \wedge \neg < guard\ a_n > \wedge < guard\ b_j > \Rightarrow < action\ b_j >$$

Algorithm 4.3 results from crossover composition of Algorithm 4.2 and the deterministic token passing algorithm of [2] (the tokens related to this algorithm are referred to as *fair privileges*). Algorithm 4.3 combines the best of both algorithms, retaining the strong probabilistic stabilization of Algorithm 4.2 and the unfair distributed scheduler support of the token passing algorithm. In the worst case, a $(n-1)$-bounded scheduler is guaranteed, which gives an $O(n^3)$ computation step time complexity.

Theorem 4.3 *Algorithm 4.3 is strong probabilistic stabilizing for $\mathcal{SP}_{ME}$ under an unfair scheduler.*

Proof: The proof directly follows from Theorems 3.1 and 4.2. □

5 Complexity

5.1 Space Complexity

The minimum non-divisor of n is $O(\log(n))$ [13]. Therefore, Algorithm 4.3 needs $O(\log(n-1) + 2 \times \log(\log(n)))$ bits per processor. Algorithms 4.1 and 4.2 use $O(\log(\log(n)))$ and $O(\log(k) + \log(\log(n)))$ bits per processor, respectively. Moreover, since it has been shown in [10] that $mnd(n)$ is constant on average (over all values of n), it follows that on average, the space complexity of Algorithm 4.3 is $log(n)$, and that the space complexity of Algorithms 4.1 and 4.2 are $O(1)$ and $O(\log(k))$, respectively.

Algorithm 4.3 Mutual exclusion under an unfair scheduler (for p) with $k = n - 1$

```
Fields:
  $t_p \in [0, mnd(n) - 1]$ (the privilege)
  $ft_p \in [0, mnd(n) - 1]$ (xsthe fair privilege)
Variables:
  $rand\_bool_p$ holds any value in {0, 1}. Each value has a probability of 1/2.
  $go\_ahead_p$ holds any integer value in [0..(2k+1)].
Predicates:
  $Privilege(p) \equiv t_p - t_{lp} \neq 1 \bmod mnd(n)$
  $Fair\_privilege(p) \equiv ft_p - ft_{lp} \neq 1 \bmod mnd(n)$
Macros:
  $Pass\_privilege(p) : t_p := (t_{lp} + 1) \bmod mnd(n)$
  $Pass\_Fair\_privilege(p) : ft_p := (ft_{lp} + 1) \bmod mnd(n)$
Actions:
$\mathcal{A}_1$:: $Privilege(p) \wedge go\_ahead_p \neq$ (2k+1) $\wedge Fair\_privilege(p) \longrightarrow$
        $Pass\_Fair\_privilege(p)$
        if ($rand\_bool_p$ = 1) then $go\_ahead_p$=(2k+1) else $go\_ahead + +$;
$\mathcal{A}_2$:: $Privilege(p) \wedge go\_ahead_p$=2k+1 $\wedge Fair\_privilege(p) \longrightarrow$
        $Pass\_Fair\_privilege(p)$ ; $Pass\_privilege(p)$;
        if ($rand\_bool_p$ = 0) then $go\_ahead_p$=random(0..2k+1);
$\mathcal{A}_3$:: $\neg Privilege(p) \wedge Fair\_privilege(p) \longrightarrow$
        $Pass\_Fair\_privilege(p)$
```

5.2 Time Complexity

In this section, we first provide the stabilization time for all three algorithms (*i.e.*, the time to recover from a fault), and then the propagation delay of the token once the system is stabilized (*i.e.*, the service time needed to grant a processor to enter its critical section).

5.2.1 Stabilization Time

As the convergence of our algorithms is only probabilistic, we can only guarantee maximum expected stabilization time. In the literature (e.g., [7]), the maximum expected stabilization time is expressed in terms of *rounds*, where a round is a minimal computation fragment during which every processor executes one action.

From Lemmas 4.1 and 4.3, a round is $O(n)$ computation steps under the synchronous scheduler, and $O(nk)$ computation steps using the k-bounded and unfair scheduler.

Algorithm 4.1. We consider the behavior of m pairs of successive tokens during one round. The probability that no two consecutive tokens merge in this round is less than

$$p = \left(\frac{1}{2}\right)^{m \times n}$$

Thus, the probability that at least one pair of two consecutive tokens merges is more than $q = 1 - p$. Then, the expected number of rounds before the system reaches a configuration with $m-1$ tokens is $E[m, m-1] < \frac{1}{q}$, *i.e.*,

$$E[m, m-1] < \frac{2^{m \times n}}{2^{m \times n} - 1}$$

The maximum expected number of rounds (let us denote it by $T_{4.1}$) to converge from an arbitrary configuration (where the number of privileges is m) to a legitimate configuration (where the number of privileges is 1) is given by the formula

$$T_{4.1} \leq \sum_{i=2}^{m} \frac{2^{in}}{2^{in} - 1} \leq m + 2$$

If $T_{4.1}$ is $O(m)$ rounds and $m \leq n$, then $T_{4.1}$ is $O(n^2)$ computation steps.

Algorithm 4.2 where $k = n - 1$ and Algorithm 4.3. We now calculate the maximum expected number of rounds for Algorithm 4.2 (where $k = n - 1$) to stabilize starting from the worst possible configuration (with m tokens).

First, we find an upper bound on the expected number of rounds needed to reach a configuration where the number of tokens is one less than that in the starting configuration. We consider the behavior of m pairs of successive tokens during one round. The probability that no two consecutive tokens merge is less than

$$p = \left(\frac{1}{2} \times \frac{1}{(2k+2)}\right)^{m \times n}$$

Thus, the probability that at least one pair of consecutive tokens merges is more than $q = 1 - p$. Then, the expected number of rounds before the system reaches a configuration with $m - 1$ tokens is $E[m, m-1] < \frac{1}{q}$, *i.e.*

$$E[m, m-1] < \frac{2(2k+2)^{m \times n}}{2(2k+2)^{m \times n} - 1}$$

The maximum expected number of rounds $T_{4.2}$ before stabilization of Algorithm 4.2 (where $k = n - 1$) from a configuration with m privileges to a configuration with 1 privilege is given by the formula

$$T_{4.2} \leq \sum_{i=2}^{m} E[i, i-1] = \sum_{i=2}^{m} \frac{2(2n-1)^{in}}{2(2n-1)^{in} - 1} \leq m$$

Since $T_{4.2}$ is $O(m)$ rounds and $m \leq n$, $T_{4.2}$ is $O(n^3)$ computation steps. A processor executing Algorithm 4.3 executes a rule of Algorithm 4.2 if and only if it holds a fair token. For the time complexity analysis, the worst number of fair tokens is 1. Hence, the bound provided for Algorithm 4.2 holds for Algorithm 4.3 too. Therefore, its time complexity is $O(n^3)$.

5.2.2 Propagation Delay

Once stabilized, in the worst case, the upper bound between two appearances of a privilege at the same processor p in Algorithms 4.1, 4.2, and 4.3 is $2 \times n$, $(2k+2) \times n$, and n^3, respectively. The average delays are $\frac{3 \times n}{2}$, $\frac{(2k+3) \times n}{2}$ and $\frac{n^2(n+1)}{2}$, respectively.

6 Conclusion

We presented a solution to the open problem of having a strong probabilistic self-stabilizing mutual exclusion algorithm under an unfair distributed scheduler. Once the system is stabilized, a processor only waits a bounded (polynomial) amount of time. Bounding the coin tossing as presented in this paper can be applied to several other probabilistic algorithms (e.g., [8, 13]) to provide a bound of the service time. The service time provided by our solutions is $(2k+2) \times n$ (Algorithm 4.2) and n^3(Algorithm 4.3) respectively. The average service times are $\frac{(2k+3)\times n}{2}$ and $\frac{n^2(n+1)}{2}$ for the two algorithms.

References

[1] J. Beauquier, S. Cordier, and S. Delaët. Optimum probabilistic self-stabilization on uniform rings. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 15.1–15.15, 1995.

[2] J Beauquier, M Gradinariu, and C Johnen. Crossover composition. In *Proceedings of the Fifth Workshop on Self-stabilizing Systems (WSS 2001)*, pages 19–34, 2001.

[3] J. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.

[4] A K Datta, M Gradinariu, and S Tixeuil. Self-stabilizing mutual exclusion using unfair distributed scheduler. In *Proceedings of IPDPS'2000, Cancuun, Mexico*, pages 465–470, May 2000.

[5] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[6] S Dolev. *Self-stabilization*. The MIT Press, 2000.

[7] S. Dolev, A. Israeli, and S. Moran. Analyzing expected time by scheduler-luck games. *IEEE Transactions on Software Engineering*, 21:429–439, 1995.

[8] J. Durand-Lose. Randomized uniform self-stabilizing mutual exclusion. In *Proceedings of the Second International Comference on Principles of Distributed Systems*, pages 89–98, 1998.

[9] M. Flatebo and A. Datta. Two-state self-stabilizing algorithms for token rings. *IEEE Transactions on Software Engineering*, 20:500–504, 1994.

[10] M Gradinariu and S Tixeuil. Tight space uniform self-stabilizing l-exclusion. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS 2001)*, April 2001.

[11] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35:63–67, 1990.

[12] T. Herman. Self-stabilization: randomness to reduce space. *Distributed Computing*, 6:95–98, 1992.

[13] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *PODC90 Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 119–131, 1990.

[14] H. Kakugawa and M. Yamashita. Uniform and self-stabilizing token rings allowing unfair daemon. *IEEE Transactions on Parallel and Distributed Systems*, 8:154–162, 1997.

[15] Raida Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley Longman, 2000.

[16] A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of Aspen and Herlihy: a case study. *Distributed Computing*, 13(4):155–186, 2000.

[17] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, Departament of Electrical Engineering and Computer Science, 1995.

[18] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In Springer-Verlag, editor, *CONCUR '94, Concurrency Theory, 5th International Conference , LNCS:836*, Uppsala, Sweden, August 1994.

[19] S. H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic i/o automata. In *CONCUR'94, 5th International Conference Concurrency theory LNCS:836*, pages 513–528, 994.