Self-Stabilizing Mutual Exclusion Using Unfair Distributed Scheduler

Ajoy K. Datta[†] Maria Gradinariu[⋆] Sébastien Tixeuil[⋆]

[†]Department of Computer Science, University of Nevada Las Vegas (datta@cs.unlv.edu)
[⋆]Laboratoire de Recherche en Informatique, Université de Paris Sud, France ({mariag,tixeuil}@lri.fr)

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 without the need for explicit exception handler of backward recovery. Mutual exclusion is fundamental in the area of distributed computing, by serializing the accesses to a common shared resource. All existing probabilistic self-stabilizing mutual exclusion algorithms designed to work under an unfair distributed scheduler suffer from the following common drawback: Once stabilized, there exists no upper bound of time between two executions of the critical section at a given node. We present the first probabilistic self-stabilizing algorithm that guarantees such a bound ($O(n^3)$, where n is the network size) while working using an unfair distributed scheduler. As the scheduling adversary gets weaker, the bound gets better. Our algorithm works in an anonymous unidirectional ring of any size and has a $O(n^3)$ expected stabilization time. Due to space restiction, proofs are omitted from this extended abstract and can be found in [6].

1. Introduction

Self-Stabilizing Mutual Exclusion One of the most inclusive approaches to fault-tolerance in distributed systems is *self-stabilization* [7, 16]. Introduced by Dijkstra [7], this technique guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior. Since most self-stabilizing fault-tolerant algorithms are nonterminating, if the distributed system is subject to transient faults corrupting the internal node state but not its behavior, once faults cease, the algorithms themselves guarantee to recover in a finite time to a safe state without any human intervention.

The mutual exclusion is fundamental 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: *(i) Exclusion*: At most one processor is allowed to execute its critical section at any time; and *(ii) Fairness*: Every processor must be able to execute its critical section infinitely often.

Related Work Since the pioneering paper of Dijkstra [7] including three self-stabilizing mutual exclusion algorithms on unidirectional rings, numerous papers have been written in this topic. Dijkstra's three algorithms are deterministic and *non-uniform* (in such an algorithm, some processors are distinguished from the others in the sense that these distinguished processors are allowed to execute a program that is different from that of the other processors). In [4], 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. For bidirectional rings, Huang presented in [13] a deterministic self-stabilizing mutual exclusion algorithm.

Several papers investigated the mutual exclusion problem in the probabilistic (or randomized) setting. Randomization was used to reduce the space in [10, 12], and to cope with the case of anonymous networks in [1, 11]. However, a common problem to all these probabilistic self-stabilizing 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 exists 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 [15] 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 [15] works *only* under the *central scheduler* (which allows

exactly one enabled processor at any time).

Our Contribution We answer the open question of [15] and provide a *strong probabilistic stabilizing* algorithm for the fair 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 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), and finally, use the composition technique described in [3] to stabilize the algorithm under an unfair distributed scheduler. 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. Once stabilized, the upper bound between two occurrence 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 The rest of the paper is organized as follows: The model for probabilistic self-stabilizing algorithms is presented in Section 2 In Section 3, three strong probabilistic self-stabilizing algorithms for mutual exclusion in unidirectional rings are presented. These algorithms are designed for three different schedulers, in the order from the weakest to the strongest adversary: synchronous, k-bounded, and unfair. Section 5 provides some concluding remarks.

2. Model

Distributed Systems A distributed system is a collection of individual computing devices that can communicate with each other. We model a distributed system $S = (C, T, I)$ 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 ring networks where the processors maintain two types of variables: *local variables* and *field variables*. The local variables of P_i cannot be accessed by its right neighbor, whereas the field variables are part of the shared register which is used to communicate to 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 P_i's left neighbor'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 the set 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 reached 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 computational 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 of computation e is denoted as $h_n = [h'_{n-1}(c, CH, c')]_n$, where h'_{n-1} is a history of length $n-1$ and (c, CH, c') is a computation step from the final configuration of the history h'_{n-1}. In the following, if h_n is a history such that $h_n = [(c_0, CH_0, c'_0)h'_{n-2}(c_{n-1}, CH_{n-1}, c'_{n-1})]_n$, 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 (equal to c'_{n-1}) is represented by $last(h_n)$, and the first configuration in h_n (equal to c_0) is referred to as $first(h_n)$ ($first$ can also be used for an infinite computation). In some context in this paper, the length n may not be necessary and hence, will not be used. The probabilistic value of a history is the product of the probabilities of all the computation steps in the history. 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.

Scheduler A scheduler can be considered as an adversary. Intuitively the "luck" (i.e., the probabilistic part) of the algorithm and the scheduler play an infinite game. In this game, in any configuration c, during a computation e, the scheduler might use the history of the computation up to configuration c, and then chooses a non-empty subset of the enabled processors in c (according to some internal rules in the scheduler) to execute their enabled action.

The scheduler-luck game was introduced in [8]. The scheduler as described in the above paragraph is formally defined in [3] and will be used in this paper. A *choice function* for a history h returns a subset of enabled processors CH in $last(h)$. A scheduler can be defined by a collection of choice functions. A computation e *satisfies* a choice function f if and only if for any history h' of e such that $e = (h'(c, CH, c')\ldots)$, $f(h') = CH$.

A *strategy* st is a tuple $st = (c, f)$, where c is a configuration $\mathcal{S}$ and f is a choice function of a scheduler. A computation e of $\mathcal{S}$ is called a st*-computation* where $st = (c, f)$ if and only if *(i)* $first(e) = c$ and *(ii)* if e satisfies the choice function f. In other words, an st-computation e is such that every choice of an enabled processor (by the scheduler) is the result of an application of the choice function of strategy st to the history (of e) at the time the choice was made.

Let st be a strategy. An st*-cone* $\mathcal{C}_h$ corresponding to a history h is the set of all possible st-computations which create the same history h (more details in [17]). The probabilistic value of an st-cone $\mathcal{C}_h$ is the probabilistic value of the history h (i.e., the product of the probability of every computation step in h). An st-cone $\mathcal{C}_{h'}$ is called a *sub-cone* of $\mathcal{C}_h$ if and only if $h' = [hx]$, where x is a computation fragment.

In [3], each strategy is the base for a probabilistic space in which any set of st-computations has an associated probabilistic value.

Probabilistic Self-Stabilizing Systems A probabilistic self-stabilizing system is a probabilistic distributed system satisfying two important properties: *probabilistic convergence* (the system converges to a configuration satisfying a *legitimacy predicate*) and *correctness* (once the system is in a configuration satisfying a legitimacy predicate, it satisfies the system specification). The literature on self-stabilization discusses two variants of the probabilistic self-stabilizing systems: the systems with *weak correctness*—the system correctness is only probabilistic, and the systems with *strong correctness*—the system correctness is certain.

The problem with systems satisfying the weak correctness is that there is always at least one infinite, incorrect computation. Even though the theoretical probability to obtain an incorrect computation is zero, this is a weakness of the system for any actual applications.

Notation 2.1 *Let S be a system, D be a scheduler, and $st = (c, f)$ be a strategy such that c is a configuration of S and f is a choice function of D. We use $\mathcal{EPRED}_{st}$ to represent the set of st-computations that reach a configuration c' such that c' satisfies the predicate $PRED$ (denoted as $c' \vdash PRED$). The notation $Pr(\mathcal{EPRED}_{st})$ is used to express the probabilistic value associated with $\mathcal{EPRED}_{st}$.*

Definition 2.1 (Closed Predicate) *A predicate $PRED$ defined on the system configurations is* closed *if and only if the following condition is true: $PRED$ holds in configuration c implies that $PRED$ also hold in any configuration c' reachable from c.*

The weak self-stabilizing systems is defined below using a special predicate, called *legitimacy predicate*, defined on the configurations. A computation e of S satisfying a predicate SP is denoted as $e \vdash SP$.

Definition 2.2 (Weak Probabilistic Stabilization) *A system S is* weak self-stabilizing *under D for a specification SP if and only if there exists a closed predicate L on configurations such that in any strategy $st = (c, f)$ of S under D, any st-computation e satisfies the following two conditions:*
(i) The probability of the set of st-computations, starting from c, reaching a configuration c', such that c' satisfies L (the legitimacy predicate), is 1 (probabilistic convergence); (Formally, $\forall st, Pr(\mathcal{EL}_{st}) = 1$.) and
(ii) The probability of the set of computations, starting from a configuration c' such that c' satisfies L, satisfies SP is 1 (weak correctness). (Formally, $\forall st, Pr(\{e \in st : e = (e'e''), last(e') \vdash L$ and $e'' \vdash SP\}) = 1$.)

The Weak probabilistic stabilizing systems only guarantee the *weak correctness* (once the system is in a configuration satisfying the legitimacy predicate, the system satisfies the specification with probability 1). A desirable property for the probabilistic distributed self-stabilizing systems would be a strong correctness. This would ensure that starting from a configuration satisfying the legitimacy predicate, the system unconditionally satisfies its specification. Such systems are called *strong stabilizing systems*.

Definition 2.3 (Strong Probabilistic Stabilization) *A system S is* strong self-stabilizing *under D for a specification SP if and only if there exists a closed predicate L on configurations such that in any strategy $st = (c, f)$ of S under D, the two following conditions hold:*
(i) The probabilistic convergence property is satisfied; (Formally, $\forall st, Pr(\mathcal{EL}_{st}) = 1$)
(ii) All computations, starting from a configuration c' such that c' satisfies L, satisfy SP (strong correctness). (Formally, $\forall st, \forall e \in st : e = (e'e'')$, $last(e') \vdash L$ and $e'' \vdash SP$).

Convergence of Probabilistic Stabilizing Systems Building on previous works on probabilistic automata (see [18, 19, 17]), [3] presented a framework for proving self-stabilization of probabilistic distributed systems. In the following we recall the main results of [3], which are based on a key property of the system called *local convergence*

and denoted by LC. This LC property is a progress statement that has positive probability as those presented in [5] (for the case of deterministic systems) and [17] (for the case of probabilistic systems).

An st-cone $\mathcal{C}_h$ satisfies the $local\_convergence$ property, denoted as $local\_convergence(PR1, PR2, \delta, n)$, where $PR1$ and $PR2$ are two predicates defined on configurations and $PR1$ is a closed predicate, if the following two conditions hold: *(i)* $last(h)$ satisfies $PR1$; and *(ii)* The probability of all st-computations in $\mathcal{C}_h$ reaching a configuration c', such that c' satisfies $PR2$, in n computation steps, is at least δ. Informally, the $local\_convergence$ 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. We now formally capture the notion of $local\_convergence$ below:

Definition 2.4 (Local Convergence) *Let st be a strategy, $PR1$ and $PR2$ be two predicates on configurations, where $PR1$ is a closed predicate. Let $\mathcal{C}_h$ be a st-cone. $\mathcal{C}_h$ satisfies $local\_convergence(PR1, PR2, \delta, n)$ if and only if the following conditions are true:*
(1) $last(h) \vdash PR1$, and
(2) *Let M denote the set of sub-cones $\mathcal{C}_{h'}$ such that the following is true for every sub-cone $\mathcal{C}_{h'}$: $last(h') \vdash PR2$ and $length(h') - length(h) \leq n$. Then, $Pr(\bigcup_{\mathcal{C}_{h'} \in M} \mathcal{C}_{h'}) \geq \delta$.*

Now, if in strategy st, there exist $\delta_{st} > 0$ and $n_{st} \geq 1$ such that any st-cone, $\mathcal{C}_h$ with $last(h) \vdash PR1$, satisfies $local\_convergence(PR1,\ PR2,\ \delta_{st},\ n_{st})$, then the main theorem from [3] states that the probability of the set of st-computations reaching configurations satisfying $PR1 \wedge PR2$ is 1. We give the formal statement of the theorem below:

Theorem 2.1 ([3]) *Let st be a strategy. Let $PR1$ and $PR2$ be closed predicates on configurations such that $Pr(\mathcal{EPR}1_{st}) = 1$. If $\exists \delta_{st} > 0$ and $\exists n_{st} \geq 1$ such that any st-cone, $\mathcal{C}_h$ with $last(h) \vdash PR1$, satisfies the $local\_convergence(PR1, PR2, \delta_{st}, n_{st})$ property, then $Pr(\mathcal{EPR}12) = 1$, where $PR12 = PR1 \wedge PR2$.*

3. Strong Probabilistic Stabilizing Mutual Exclusion

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.

The specification of the mutual exclusion problem reveals two points: *(i)* a static part—there is exactly one privilege in the system and *(ii)* a dynamic part—the fairness issue.

In [15], Kakugawa and Yamashita point out that 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 [10]) and *(ii)* by requiring that the correctness of the system is probabilistic (as in [1] and [11]). One can characterize the previous works in two ways. Some of them are self-stabilizing but solve a weaker problem than the mutual exclusion problem. Others are weak stabilizing algorithms to solve the mutual exclusion problem. The open question in [15] was to design a strong probabilistic stabilizing algorithm that solves the mutual exclusion problem under an unfair distributed scheduler. We present such an algorithm in this paper.

3.1. Synchronous Scheduler

The algorithm for mutual exclusion under the synchronous scheduler is presented as Algorithm 3.1.

Algorithm 3.1 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 3.1, 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.

Theorem 3.1 *Algorithm 3.1 is strong probabilistic stabilizing for $\mathcal{SP}_{ME}$ under a synchronous scheduler.*

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

3.2. k-Bounded Scheduler

In this section, we present a generalization of Algorithm 3.1. In Algorithm 3.2, the local variable, go_ahead, which enables the privilege passing has $(k+2)$ states (from 0 to $k+1$); the states from 0 to k are *wait* states, and state $(k+1)$ is a *pass* state. Intuitively, the go_ahead variable is an internal clock for the processor. When this clock equals $(k+1)$, the processor must pass the privilege and change its clock value randomly. Otherwise, the processor randomly increases its clock.

Algorithm 3.2 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 {1, 0}. Each value has a probability of 1/2.
go_ahead_p holds any value in [0, (k+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:
$\mathcal{A}_1$:: $Privilege(p) \wedge go\_ahead_p \neq$ (k+1) $\longrightarrow$
  if ($rand\_bool_p$ = 1) then go_ahead_p=(k+1);
  else $go\_ahead++$;
$\mathcal{A}_2$:: $Privilege(p) \wedge go\_ahead_p$=(k+1) $\longrightarrow$
  $Pass\_privilege(p)$;
  if ($rand\_bool_p$ = 0) then
    go_ahead_p=random(0..k+1);

Theorem 3.2 *Algorithm 3.2 is strong probabilistic stabilizing for $\mathcal{SP}_{ME}$ under a k-bounded scheduler.*

3.3. Unfair Scheduler

In this section, we extend Algorithm 3.2 so that it can cope up with an unfair scheduler. For this purpose, the idea of *cross-over* composition (introduced in [2]) is used to compose our algorithm with an algorithm for the k-fairness specification (see [3]), which is defined below:

Definition 3.1 (k-Fairness) *An algorithm is* k-fair *if and only if in any computation of the algorithm, the two following properties hold:* (i) *Every processor executes an action infinitely often and* (ii) *Between any two actions of a processor, at most k actions are executed at any other processor.*

The cross-over composition guarantees that a stabilizing algorithm for specification $\mathcal{SP}$, that works under the k-bounded scheduler composed with a k-fair algorithm under an arbitrary scheduler, is stabilizing under an unfair distributed scheduler for specification $\mathcal{SP}$. The Deterministic Token Circulation algorithm from [3] (DTC) is an $(n-1)$-fair algorithm for uniform unidirectional rings of size n running under an unfair scheduler.

Cross-Over Composition The cross-over composition technique combines two algorithms —the *weaker* and the *stronger*—and outputs a new algorithm. In this paper, the stronger (DTC) supports the stronger adversary (unfair scheduler), while the weaker (Algorithm 3.2) provides an extended functionality (mutual exclusion) under a weaker adversary (the k-bounded scheduler). The resulting algorithm is obtained as follows:
1. For every weaker action and for every stronger action, the resulting action is:
$< guard\ weaker > \wedge < guard\ stronger > \longrightarrow < statement\ weaker >; < statement\ stronger >$.
2. For every stronger action and for the all negated weaker guards, the resulting action is:
$< no\ weaker\ guard\ holds > \wedge < guard\ stronger > \longrightarrow < statement\ stronger >$.

We obtain Algorithm 3.3 as a result of cross-over composition.

Theorem 3.3 *Algorithm 3.3 is strong probabilistic stabilizing for $\mathcal{SP}_{ME}$ under an unfair distributed scheduler.*

4 Complexity

Space Complexity From [14], the minimum non-divisor of n is $O(\log(n))$. Therefore Algorithm 3.3 needs $O(\log(n-1) + 2 \times \log(\log(n)))$ bits per processor. Algorithms 3.1 and 3.2 use $O(\log(\log(n)))$ and $O(\log(k) + \log(\log(n)))$ bits per processor, respectively.

Stabilization Time Complexity As the convergence of our algorithms is only probabilistic, we can only guarantee maximum expected stabilization time. In the literature (such as in [8]), the maximum expected stabilization time is expressed in terms of *rounds*, where a round is a computation fragment where every processor executes at least one action. A round is $O(n)$ computation steps under the synchronous scheduler, and $O(nk)$ computation steps using the k-bounded and unfair scheduler.

Algorithm 3.1 The maximum expected number of rounds (let us denote it by $T_{3.1}$) such that starting from a configuration where the number of privileges is m, to reach a configuration where the number of privileges is 1, is given by the formula $T_{3.1} \leq \sum_{i=2}^{m} \frac{2^{in}}{2^{in}-1} \leq m+2$. If $T_{3.1}$ is $O(m)$ rounds and $m \leq n$, then $T_{3.1}$ is $O(n^2)$ computation steps.

Algorithm 3.3 Mutual exclusion under an unfair scheduler (for p)

Field variables on p:
$t_p \in [0, mnd(n) - 1]$ (*the privilege*)
$dt_p \in [0, mnd(n) - 1]$ (*the deterministic token*)
Variables on p:
$rand\_bool_p$ holds any value in $\{1, 0\}$. Each value has a probability 1/2.
go_ahead_p holds any value [0, k+1].
Predicate:
$Privilege(p) \equiv t_p - t_{lp} \neq 1 \bmod mnd(n)$
$Deterministic\_token(p) \equiv dt_p - dt_{lp} \neq 1 \bmod mnd(n)$
Macro:
$Pass\_privilege(p) : t_p := (t_{lp} + 1) \bmod mnd(n)$
$Pass\_Deterministic\_token(p) : dt_p := (dt_{lp} + 1) \bmod mnd(n)$
Action on p:
$\mathcal{A}_1$:: $Deterministic\_token(p) \wedge Privilege(p) \wedge go\_ahead_p \neq (k+1) \longrightarrow$
  $Pass\_Deterministic\_token(p)$
  if ($rand\_bool_p$ = 1) then
    go_ahead_p=(k+1);
  else $go\_ahead++$;
$\mathcal{A}_2$:: $Deterministic\_token(p) \wedge Privilege(p) \wedge go\_ahead_p = (k+1) \longrightarrow$
  $Pass\_Deterministic\_token(p)$
  $Pass\_privilege(p)$;
  if ($rand\_bool_p$ = 0) then
    go_ahead_p=random(0..k+1);
$\mathcal{A}_3$:: $Deterministic\_token(p) \wedge \neg Privilege(p) \longrightarrow$
  $Pass\_Deterministic\_token(p)$

Algorithm 3.2 where $k = n - 1$ The maximum expected number of rounds (let us call it $T_{3.2}$), such that starting from a configuration where the number of privileges is m, to reach a configuration where the number of privileges is 1, is given by the formula $T_{3.2} \leq \sum_{i=2}^{m} \frac{2(n+1)^{in}}{2(n+1)^{in}-1} \leq m$. If $T_{3.2}$ is $O(m)$ rounds and $m \leq n$, then $T_{3.2}$ is $O(n^3)$ computation steps.

Algorithm 3.3 Algorithm 3.3 has the same maximum expected stabilization time as Algorithm 3.2.

Propagation Delay Once stabilized, in the worst case, the upper bound between two appearances of a privilege at the same processor p in Algorithms 3.1, 3.2, and 3.3 is $2 \times n$, $(k+2) \times n$, and n^3, respectively. On average, the delay is $\frac{3 \times n}{2}$, $\frac{(k+3) \times n}{2}$ and $\frac{n^2(n+1)}{2}$, respectively.

5. Conclusions

We presented a solution to the open problem of having a probabilistic self-stabilizing algorithm solving mutual exclusion and conforming to strong correction—once the system is stabilized, a processor only waits a bounded (polynomial) amount of time (and not an *expected* bounded amount of time). Although being presented on unidirectional rings, we believe the technique presented in this paper (bounding the coin tossing) can be extended to several other probabilistic algorithms (such as [9, 14]) in order to provide a bound on the service time.

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. Memory space requirements for self-stabilizing leader election protocols. In *PODC99*, pages 199–208, 1999.

[3] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing optimal leader election under arbitrary scheduler on rings. Technical Report 1225, Laboratoire de Recherche en Informatique, September 1999.

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

[5] K. Chandy and J. Misra. Parallel programs design: A foundation. New York, N.Y., 1988.

[6] A. K. Datta, M. Gradinariu, and S. Tixeuil. Self-stabilizing mutual exclusion using unfair distributed scheduler. Technical Report 1227, Laboratoire de Recherche en Informatique, 1999.

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

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

[9] 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.

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

[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] S. Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15:563–573, 1993.

[14] 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.

[15] 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.

[16] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[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 *concur94*, pages 513–528, 994.