Token based Self-Stabilizing Uniform Algorithms [1]

Joffroy Beauquier
LRI CNRS-UMR 8623, Bat 490, Université Paris Sud
F-91405 Orsay Cedex, France
E-mail: jb@lri.fr

and

Jérôme Durand-Lose
I3S, CNRS UPREES-A 6070, 930 Route des Colles, BP 145
06903 Sophia Antipolis Cedex, France
E-mail: jdurand@unice.fr

and

Maria Gradinariu and Colette Johnen
LRI CNRS-UMR 8623, Bat 490, Université Paris Sud
F-91405 Orsay Cedex, France
E-mail: mgradina@irisa.fr
E-mail: colette@lri.fr

Version: February 15, 2002

This work focuses on self-stabilizing algorithms for *mutual exclusion* and *leader election* — two fundamental tasks for distributed systems. Self-stabilizing systems are able to recover by themselves, regaining their consistency from any initial or intermediary faulty configuration.

The proposed algorithms are designed for any directed, anonymous network and stabilize under any distributed scheduler. The algorithms keystone is the token management and routing policies. In order to break the network symmetry, randomization is used. The space complexity is $O((D^+ + D^-) \cdot (log(snd(n)) + 2))$ where n is the network size, $snd(n)$ is the the smallest integer that does not divide n and D^+ and D^- are the maximal out and in degree respectively. It should be noticed that $snd(n)$ is constant on the average and equals 2 on odd size networks.

Key Words: self-stabilization, randomized protocol, unfair scheduler, leader election, mutual exclusion, directed network.

1. INTRODUCTION

In a distributed system, several processors cooperate to achieve some global task. A prerequisite for cooperation is to implement a distributed control, i.e. reaching or maintaining a global predicate despite of partial (or local) access to the system state. Token circulation and leader election algorithms constitute well-known examples of global tasks. The former manages the fair circulation of exactly one token, while the

[1] contact author : Colette Johnen, LRI, université Paris-Sud, centre d'Orsay, 91405 Orsay Cedex, France, colette@lri.fr, tel : +33 1 69 15 67 02, fax : +33 1 69 15 65 86; running head : Token Based Self-Stabilizing Uniform Algorithms

latter consists in distinguishing one processor called the *leader*. The fair circulation of exactly one token can be used to solve the mutual exclusion problem. Once a leader is elected, many other tasks can be solved using a centralized control (for instance, resource allocation or synchronization).

Self-stabilization is a framework for dealing with channel or memory transient failures. After a failure, the system is allowed to temporarily exhibit an incorrect behavior, but after a period of time (as short as possible) it must behave correctly, without external intervention. A self-stabilizing leader election or token circulation protocol starting, for example, in a symmetric configuration requires a way to break the symmetry. The id-based systems (every processor has an unique identifier) prevent the existence of symmetric configurations. In anonymous systems (all processors are identical), the symmetry can only be broken by randomization [3].

Related works. Self-stabilization was introduced by Dijkstra in [11]; three self-stabilizing deterministic token circulation algorithms for semi-uniform systems are presented. In a semi-uniform algorithm, some specific processors do not perform the same algorithm as the other processors. In [17], Israeli and Jalfon provide a token management policy and a graph traversal scheme (token routing scheme) yielding self-stabilizing mutual exclusion for undirected (bidirectional) networks. In order to break symmetry they use the random walks technique described by Aleliunas *et al.* in [1]. Self-stabilizing token circulation algorithms coping with anonymous systems are presented in [16, 5]. These solutions are designed for directed (unidirectional) rings. In [16], Herman presents an algorithm for odd size rings. In [5], Beauquier *et al.* present an algorithm which ensures token circulation on directed rings of any size. To guarantee the presence of a token in the ring the smallest non divisor of n (n being the network size), $snd(n)$ — the "magic" number as it was defined in [17] — is used. Alstein *et al.* present in [2] two mutual exclusion algorithms for directed arbitrary networks with identifiers requiring the preprocessing of a spanning tree. Kakugawa and Yamashita present in [19] a self-stabilizing token circulation protocol under unfair scheduler on rings. In [15], Durand-Lose reports an original token management solution on undirected networks which ensures the existence of a single token in the network (the "magic number" is also used). Random walks are used for breaking symmetry. The space complexity of this protocol is $O(D \cdot log(snd(n)))$ where D is the maximal processor's degree. In [21], Rosaz presents a randomized mutual exclusion algorithm in the message passing model. The solution has a polynomial stabilization time. Awerbuch and Ostrovsky present in [4], a self-stabilizing leader election protocol on undirected id-based networks. It requires $\log^*(N)$ states per processor (N is the network size). A basic protocol is given, requiring N states per processor, and the result is obtained by using a data structure that stores distributively the variables. In an appendix of [18], Itkis and Levin use another data structure based on the Thue-Morse sequence, requiring $O(1)$ bits per edge to store, in a distributed manner, variables having possibly N values. These two last algorithms require undirected networks. Assuming that the deadlock freedom property is guaranteed externally, Mayer *et al.* propose in [20] a randomized self-stabilizing leader election protocol in the message passing model. In [13], Dolev *et al.* present two leader election protocols in complete networks. The protocols are self-stabilizing under read/write atomicity. Using the scheduler-luck game technique, polynomial bounds for the stabilization time are provided. In [14] a dynamic leader election algorithm under read/write atomicity is reported by Dolev *et al.*. Randomization is used for breaking symmetry but an unbounded memory

space is required. Beauquier *et al.* propose in [6] a space optimal leader election on the ring topology. The propose bound for the space complexity is $O(log(snd(n)))$.

Our Contributions. We propose a self-stabilizing token circulation algorithm and a leader election algorithm under an arbitrary scheduler for any anonymous directed network : there is no requirement on the scheduler and on the network topology (except strong connectivity). The optimality of the result is proven in [6] for the ring topology. Protocols are based on a token management policy that guarantees the presence of at least one token. We also provide a token routing policy which ensures the token circulation. The token routing policy provides an upper bound for the number of steps executed by a processor between two successive steps of any other processor. This policy is used in the automatic construction of Algorithm 6.2 — self-stabilizing leader election under an arbitrary scheduler. A probabilistic version of token circulation (Algorithm 4.1) yields a mutual exclusion protocol. The space complexity of our algorithms is $O((D^+ + D^-) \cdot (log(snd(n)) + 2))$ bits per processor where D^+ and D^- are the maximal out and in network degrees. It should be noticed that $snd(n)$ is constant on the average and equals 2 on odd-size networks.

2. MODEL

Transition System. A distributed system is a collection of intercommunicating state machines. We formally specify a distributed system by a *transition system* $TS = (\Sigma, C, T, I)$ where Σ is an alphabet, C is the set of system configurations, $T \subseteq C \times C$ is the set of *transitions* and $I \subseteq C$ is the set of initial configurations. Each transition of T is labeled with a symbol from Σ. A *probabilistic distributed system* is a distributed system where a probabilistic distribution is defined on transitions.

A *computation* e of TS starting in a configuration $c_1 \in I$ is a maximal sequence of transitions $e = ((c_1, c_2), (c_2, c_3) \ldots)$ such that $(c_i, c_{i+1}) \in T$, $\forall i \geq 1$. The length of a finite prefix h of a computation is denoted by $length(h)$, the last configuration in h is represented by $last(h)$, and the first configuration in h is $first(h)$ ($first$ can be also used for an infinite computation). A *computation factor* is a finite sequence of computation steps. If h and x are computation factors such that $first(x) = last(h)$ then hx denotes the factor corresponding to the sequence h followed by x.

Let c be a system configuration. A *TS-tree* rooted in c, $\mathcal{T}ree(c)$, is the tree-representation of all computations starting from c. Let nd_1 be a node in $\mathcal{T}ree(c)$, the *α-branch* rooted in nd_1 is the set of all $\mathcal{T}ree(c)$ computations starting in nd_1 having the first transitions labeled with α (a letter of Σ). The degree of nd_1 is the number of branches rooted in nd_1. A *sub-TS-tree of degree* 1 rooted in c is a restriction of $\mathcal{T}ree(c)$ in which any node has the degree 0 (i.e. there is a deadlock) or 1. On a sub-tree of degree 1 of a probabilistic distributed system, the set of first transitions of a branch is the base set for a discrete probabilistic space. Any transition in this set has a positive probability and the sum of probabilities is 1 for every node.

Scheduler and strategy. A scheduler is usually presented in the literature (see [12], [13], [22]) as an adversary for a distributed system which "chooses" at each configuration the next transition. Classically, a scheduler is defined as a function over the distributed system executions which, for a given configuration, returns the

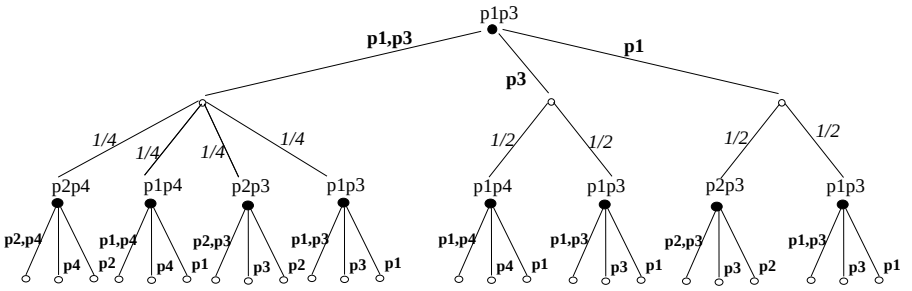

FIG. 1 The beginning of $\mathcal{T}ree(c)$ of the Algorithm 4.1

next transition. In the process of choosing a transition, a scheduler may have access to partial or total information on the system history. Note that some important scheduler types cannot be specified as functions over the finite history of system executions, like for instance the fair scheduler.

In the model that we use, a *scheduler* is a *predicate* over the system computations. This definition copes up with any type of scheduler even with those having a dynamical behavior, according to the system evolution. In the sequel, we use the k-bounded scheduler (during a system computation, while a processor is enabled another processor can perform at most k actions) and the distributed unfair scheduler (during a computation, some enabled processors may starve — they never perform an action).

The interaction between a scheduler and the distributed system generates what we call here *strategies*, defined as follows :

DEFINITION 1 (Strategy). Let TS be a transition system and let c be a TS configuration. A *strategy* rooted in c is a sub-TS-tree of degree 1 of $\mathcal{T}ree(c)$.

Let st be a TS strategy, and A be a scheduler (i.e. a predicate over the computations). st is a strategy under A iff all computations of st verify A.

In Figure 1, we present the beginning of the $\mathcal{T}ree(c)$ of the Algorithm 4.1 on the 4-ring (p1, p2, p3 and p4). The Algorithm 4.1 provides a self-stabilizing token circulation. c is the configuration where both processor p1 and p3 have a token. Figure 2 presents the beginning of a specific strategy of $\mathcal{T}ree(c)$: at each step, all processors having a token perform their action.

Note that a $\mathcal{T}S$ tree can be decomposed in a infinity of strategies.

Let st be a strategy. An *st-cone* $\mathcal{C}_h$ corresponding to a prefix h is the set of all possible computations in st with the same prefix h. The measure of an st-cone $\mathcal{C}_h$ is the measure of the prefix h (i.e., the product of the probabilities of all the transitions occurring 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 factor.

In [7] it is proven, following the classical theory of probabilistic automata (see [22]), that for any strategy, it can be built a probabilistic space having the strategy as a base set.

Distributed system topology. Throughout this paper we consider distributed systems of n intercommunicating computing devices mapped as a strongly connected directed graph $DG = (V, E)$ where V is the set of graph nodes and E the

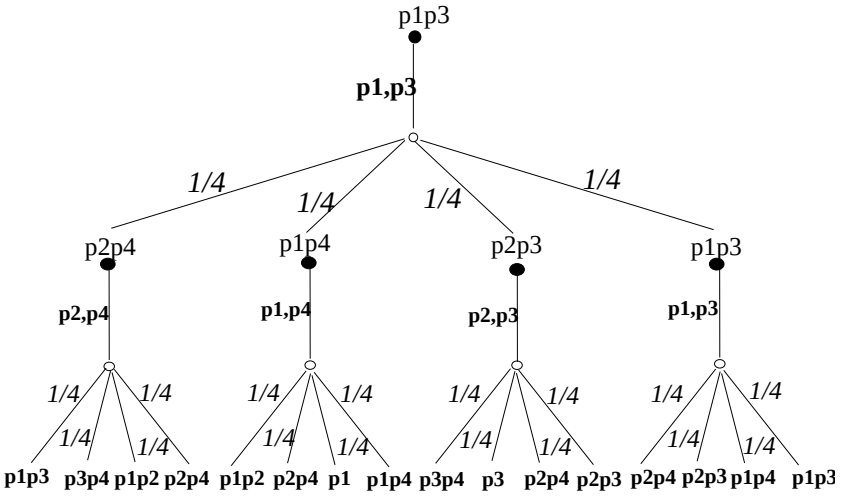

FIG. 2 The beginning of a strategy from c of the Algorithm 4.1

set of directed edges. Each node represents a computing device, also called *processor*. If $(p,q) \in E$ then p is an *in-neighbor* of q and q is an *out-neighbor* of p (p may send some data to q on (p,q), but q cannot). In the sequel, the set of node p in-neighbors is denoted by $In(p)$ and their number is denoted by $D^-(p)$. Similarly the set of out-neighbors is denoted by $Out(p)$ and the number of those neighbors by $D^+(p)$. Any node p in the network shares registers with its *in* and *out* neighbors. The node p reads the shared registers with its in-neighbors (denoted $R^p_{in}[q], q \in In(p)$) and is allowed to freely perform write and read operations on the shared registers with the out-neighbors (denoted $R^p_{out}[z], z \in Out(p)$). For a node p all the edges oriented to its out-neighbors are called *outgoing edges* and all the edges oriented towards p are called *incoming edges*.

Distributed algorithm. Any processor in a distributed system executes an algorithm which has two parts : a declarative part and a finite set of guarded actions part (i.e. $label :: \langle \text{guard} \rangle \longrightarrow \langle \text{statement} \rangle$). The values of local variables and out-registers of a processor define the processor current *state*. The guard of a processor p is a boolean expression involving the state of p and the values of p's in-neighbors registers. A guarded action (also called rule) is *enabled* if its guard is true. We assume that for any processor there is at most one enabled action at a time. A processor having an enabled action is also called *enabled processor*.

Our model deals with all kinds of atomic step. For instance, our model deal with the read/write atomicity ([12]) where a processor *atomic step* consists of an internal operation followed by either a read or a write operation (into a processor's out-register) but not both. The presented algorithms are designed for the model of composite atomicity where a processor atomic step contains both read and write operations : in one atomic step, a processor evaluates its guards and executes the statement of one enabled rule.

When an algorithm contains guarded actions with random outputs the algorithm is probabilistic (randomized) otherwise it is deterministic. The processors executing probabilistic algorithms are called randomized processors.

Distributed system versus transition system. Let S be a distributed system. We specify the distributed system S by the transition system TS. A *configuration* of TS is a vector containing the states of all processors from S. Let c be a configuration of TS, a *transition* from c is determined by the execution of one atomic step from c by one or several processors. A *local configuration* is the part of a configuration that can be "seen" by a processor (i.e. its state and the state of its neighbors). A configuration is *symmetrical* if all processors have the same local configuration.

Probabilistic Self-Stabilizing Systems A probabilistic self-stabilizing system is a probabilistic distributed system satisfying two properties : *probabilistic convergence* (the system converges to configurations satisfying a *legitimacy predicate*) and *correctness* (all the computations starting from configurations satisfying a legitimacy predicate satisfies the system specification).

A predicate P is *closed* for the computations of a distributed system if and only if when P holds in a configuration c, P also holds in any configuration reachable from c.

NOTATION 1. Let $\mathcal{S}$ be a system, A be a scheduler and st be a strategy under A. Let CP be the set of all system configurations satisfying a closed predicate P (formally $\forall c \in CP, c \vdash P$). The set of computations of st that reach configurations in CP is denoted by $\mathcal{EP}_{st}$ and its probability by $Pr_{st}(\mathcal{EP}_{st})$.

DEFINITION 2 (Probabilistic Stabilization). A system $\mathcal{S}$ is *self-stabilizing* under a scheduler A for a specification SP if and only if there exists a closed legitimacy predicate L on configurations such that in any strategy st of $\mathcal{S}$ under A, the two following conditions hold :

1. The probability of the set of computations of st, starting from c, reaching in a finite number of steps a configuration c', such that c' satisfies L is 1 (probabilistic convergence). Formally, $\forall st, Pr_{st}(\mathcal{EL}_{st}) = 1$
2. All computations, starting from a configuration c' such that c' satisfies L, satisfy SP (strong correctness). Formally, $\forall st, \forall e \in st : e = e'e''$ with $last(e') \vdash L$ then $e'' \vdash SP$.

Note that this definition is stronger than the one used in [5, 16] where the system correctness is probabilistic : for all strategies the probability of the set of computations reaching legitimate configurations and satisfying the system specification is 1. The probabilistic correctness will be called in the sequel *weak correctness* and systems satisfying a weak correctness will be called *weak self-stabilizing* systems.

Convergence of Probabilistic Stabilizing Systems Based on previous works on the probabilistic automata (see [22], [23], [24]) [7] presents a detailed framework for proving self-stabilization of probabilistic distributed systems. A key notion is *local convergence* denoted LC. The LC property is a progress statement as those presented in [9] (for the deterministic systems) and [22] (for the probabilistic systems). Informally, the LC property for a probabilistic self-stabilizing system and two predicates P_1 and P_2 means that starting in a configuration satisfying P_1, the system will reach a configuration which satisfies a particular predicate P_2, in a bounded number of computation steps with positive probability. Formally the *local convergence* property is defined as follows :

DEFINITION 3 (Local Convergence). Let st be a strategy, P_1 and P_2 be two predicates on configurations, where P_1 is a closed predicate. Let δ be a positive probability and N a positive integer. Let $\mathcal{C}_h$ be a st-cone with $last(h) \vdash P_1$ and let M be the set of sub-cones $\mathcal{C}_{h'}$ of the cone $\mathcal{C}_h$ such that for every sub-cone $\mathcal{C}_{h'}$: $last(h') \vdash P_2$ and $length(h') - length(h) \leq N$. The cone $\mathcal{C}_h$ satisfies LC (P_1, P_2, δ, N) if and only if $Pr_{st}(\bigcup_{\mathcal{C}_{h'} \in M} \mathcal{C}_{h'}) \geq \delta$.

Now, if in strategy st, there exist $\delta_{st} > 0$ and $N_{st} \geq 1$ such that any st-cone, $\mathcal{C}_h$ with $last(h) \vdash P_1$, satisfies $LC(P_1, P_2, \delta_{st}, N_{st})$, then the main theorem of the framework presented in [7] states that the probability of the set of computations of st reaching configurations satisfying both P_1 and P_2 is 1. Formally :

THEOREM 1. *[7] Let st be a strategy. Let P_1 and P_2 be closed predicates on configurations such that $Pr_{st}(\mathcal{EP}_1) = 1$. If $\exists \delta_{st} > 0$ and $\exists N_{st} \geq 1$ such that any st-cone $\mathcal{C}_h$ with $last(h) \vdash P_1$, satisfies the LC $(P_1, P_2, \delta_{st}, N_{st})$ property, then $Pr_{st}(\mathcal{EP}) = 1$, where $P = P_1 \wedge P_2$.*

Remark 1. If any strategy, st, of a distributed system satisfies $LC(PR1, PR2, \delta_{st}, N_{st})$ with $PR1$ the true predicate (verified by any configuration) and $PR2$ being the legitimacy predicate then the system satisfies the *probabilistic convergence* as defined in Definition 2.

3. TOKEN MANAGEMENT AND TOKEN ROUTING POLICY

The notions of token management and token routing policies were introduced for self-stabilizing systems in [17]. In order to implement a token management, one needs to design a pattern that (i) allows a processor to decide if it has a token through its local information (its state, and the out-registers of its out-neighbors). But also, the pattern should ensure that there is always at least one token in the network. In [17] it is suggested to use "the magic" number (the smallest non divisor of the network size) for solving this problem. [17] deals with undirected general graphs and directed rings. In this section, we present a token management and token routing policies for general directed graphs.

3.1. Token management policy

A "token" is represented by a predicate. A processor with the "token" predicate true is said to be "privileged". The self-stabilizing systems achieving mutual exclusion or leader election needs to guarantee that in the system there is always a privileged processor. Descriptions of such predicates can be found in [5] for directed rings and [15] for undirected networks. In the following, we define the token predicate for directed networks.

3.1.1. Token definition

We define tokens for directed networks and then, prove that there is at least one token in any system configuration.

NOTATION 2. Let $snd(n)$ be the smallest non divisor of n (the number of processors). Let ΔT_p be the difference (modulo snd(n)) between the sum of in-register

values and the sum of out-register values of a processor p. Formally :

$$\Delta T_p = \left(\sum_{q \in In(p)} R^p_{in}[q] - \sum_{q \in Out(p)} R^p_{out}[q] \right) \bmod snd(n) \ . \tag{1}$$

DEFINITION 4. A processor p holds a token if and only if $\Delta T_p \neq 1$. A processor holding a token is a *privileged* processor.

Using the same reasoning as in [5] or [15], we find out that this convention is sufficient to guarantee the presence of at least one token in any configuration.

LEMMA 1. *Let DS be a distributed system. In any DS configuration there is at least one privileged processor.*

Proof. Suppose that there is no privileged processor, hence $\Delta T_p = 1$ for any processor p in the network. By summing the ΔT_p for all p, we get :

$$\sum_{p \in V} \left(\sum_{q \in In(p)} R^p_{in}[q] - \sum_{q \in Out(p)} R^p_{out}[q] \right) = 0 = n \bmod snd(n) \ . \tag{2}$$

Equation (2) means that $snd(n)$ divides n which is impossible from the definition of $snd(n)$. ∎

3.1.2. Switch technique

Passing a token from a processor p to one of its out-neighbors is made according to the switch technique ([8]). Suppose without loss of generality, that the outgoing edges of a processor are labeled $0, 1, \ldots D^+(p) - 1$. A processor passes the tokens that it receives according to this labeling : if the last token has been passed on the edge i then the next one will be passed on the edge $(i + 1) \bmod D^+(p)$.

Token passing. A processor p passes a token to an adjacent processor $q \in Out(p)$ by modifying the value of $R_{out}[q]$ in the following way : $R^p_{out}[q] = R^p_{out}[q] + \Delta T_p - 1$. Hence the new value of ΔT_q is increased by $\Delta T_p - 1$ and the new value of ΔT_p is set to 1 : the token is passed from p to q.

Tokens meeting. When two tokens are passed to the same processor q, then ΔT_q is increased twice. Either the tokens annihilate each other, or they merge into a single token. The same phenomenon happens when a processor q having a token receives another token.

Remark 2. The number of tokens in a network does not increase.

3.2. Fair token routing policy

The fair token routing policy is provided by Algorithm 3.1 which performs token circulation in deterministic networks. Due to the particular encoding of a token one or more tokens are always present. A processor holding a token sends it deterministically to one of its out-neighbors. The interesting property of the algorithm is that, even if the scheduler is unfair, in any computation each processor receives infinitely many often a token. Algorithm 3.1 will be used later, in a hierarchical composition for ensure fairness from an unfair scheduler.

Algorithm 3.1 Fair token routing algorithm for processor p

Shared registers with the in-neighbors :
$R_{in}^{FT}[1..D^{-}(p)]$ where $R_{in}^{FT}[i] \in [0, snd(n) - 1]$

Shared registers with out neighbors :
$R_{out}^{FT}[1..D^{+}(p)]$ where $R_{out}^{FT}[j] \in [0, snd(n) - 1]$

Variables on p :
$direction^{FT} \in [0, D^{+}(p) - 1]$ (*the outgoing direction of the last sent fair token*)

Functions :
$\Delta FT = \left(\sum_{q \in In(p)} R_{in}^{FT}[q] - \sum_{q \in Out(p)} R_{out}^{FT}[q]\right) \bmod snd(n)$

Macros :
$New\_Dir\_Fair\_Token$ :: $direction^{FT} := (direction^{FT} + 1) \bmod D^{+}$
$Pass\_Fair\_Token$ :: $R_{out}^{FT}[direction^{FT}] := (R_{out}^{FT}[direction^{FT}] + \Delta FT - 1) \bmod snd(n)$

Predicates :
$Fair\_Token \equiv [\ \Delta FT \neq 1\ ]$

Action :
$\mathcal{FA}$:: $Fair\_Token \longrightarrow New\_Dir\_Fair\_Token$; $Pass\_Fair\_Token$

Algorithm 3.1 description. Description is very simple. In any system configuration there is a processor holding a token according to the Definition 4. It can pass this token according to the switch technique. The switch technique in encoded in the macro $New\_Dir\_Fair\_Token(p)$ where the new destination for the token is computed.

Algorithm 3.1 analysis. The switch technique guarantees that in any computation, any processor holds a token infinitely many times (fairness of the token circulation). Moreover, the number of steps taken by the other processors between two successive actions of a given processor is bounded.

LEMMA 2. *Let e be an arbitrary computation of Algorithm 3.1 starting in a configuration c with m tokens ($1 \leq m \leq n$). Let p be a processor holding a token in c. Any in-neighbor p_j of p, $p_j \in In(p)$, executes at most $m \cdot D^{+}(p_j)$ actions between two consecutive actions of p in e.*

Proof. Let us consider a factor f of computation e such that f starts by a p action, finishes by a p action too, and along the factor f the processor p does not execute any action. Let us determine the maximal number of actions which can be done by p_j in f. Every execution of a p_j action produces a token passage to one of the p_j out-neighbors chosen according to the switch technique ($direction^{FT}$ is incremented). Therefore after at most $D^{+}(p_j)$ actions of p_j a token will be sent to the processor p. Processor p keeps the token until the end of f since p is not activated. Assume that the processor p_j executes again $D^{+}(p_j)$ actions hence another token is sent to the processor p. The processor p may or not execute it

action — in the first case the factor f ends and the number of actions executed by p_j in f is $2 \cdot D^+(p_j)$. In the second case p keeps another token; thus, there are at most $m-2$ tokens that can freely move.

After at most $m \cdot D^+(p_j)$ actions of p_j in f the processor p holds the only token in the network. Thus p is the only processor which can execute an action; the factor f has to end. The maximal number of actions executed by p_j in f is $m \cdot D^+(p_j)$. ∎

Let us consider two processors p and q. The distance between p and q (the length of a shortest directed path between p and q) is denoted $dist(p,q)$. $Shortest\_path(p, q)$ denotes the set of processors on a shortest directed path from p to q.

LEMMA 3. *Let e be an arbitrary computation of Algorithm 3.1 starting in a configuration with m tokens. For any two distinct processors, p and q, between two actions of p the processor q computes at most $\prod_{i=1}^{d} m \cdot D^+(q_i)$ where $q_i \in Shortest\_path(q,p)$ and $dist(q,p) = d$.*

Proof. We call the i-th processor on the shortest past between q and p is q_i, with $q = q_1$. From Lemma 2, we know that between two actions of p the processor q_d executes at most $m \cdot D^+(q_d)$ actions; and between two actions of q_d, the processor q_{d-1} executes at most $m \cdot D^+(q_{d-1})$. Therefore between two actions of p the processor at distance 2 of p executes its actions $m^2 \cdot D^+(q_d) \cdot D^+(q_{d-1})$ times. Repeating the reasoning, between two actions of p the processor q executes at most $\prod_{i=1}^{d} m \cdot D^+(q_i)$ actions where $q_i \in Shortest\_path(q,p)$. ∎

Let us denote by D^+ the maximal out degree of the network processors and by $Diam$ the network diameter ($Diam = max_{p,q} dist(p,q)$).

COROLLARY 1. *In any computation of Algorithm 3.1 starting in a configuration with $1 \leq m \leq n$ tokens, where n is the network size, between two actions of a processor any other processor executes at most $(m \cdot D^+)^{Diam}$ actions under any scheduler.*

LEMMA 4. *Let e be a computation of Algorithm 3.1 starting in a configuration with $1 \leq m \leq n$ tokens, where n is the network size. In e, any processor executes it action within $(n-1)(m \cdot D^+)^{Diam} + 1$ computation steps.*

Proof. Let p be an arbitrary processor. From the Lemma 3 and the Corollary 1, between two actions of p another processor executes at most $(m \cdot D^+)^{Diam}$ actions. The system size is n hence the processor p executes it action after at most $(n-1)(m \cdot D^+)^{Diam}$ computation steps. ∎

COROLLARY 2. *A processor computes the actions of Algorithm 3.1 infinitely often.*

The following Corollary provides the bound for Algorithm 3.1 k-fairness defined as follows :

DEFINITION 5. A distributed algorithm is *k-fair* if and only if on every computation, 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 by any other processor.

COROLLARY 3. *Algorithm 3.1 is an $(n \cdot D^+)^{Diam}$-fair algorithm.*

Proof. The proof results from the direct application of the Corollaries 1 and 2. ∎

The lemmas 4 provide also the bound for the length of a round in an arbitrary computation e of Algorithm 3.1, defined as follows :

DEFINITION 6. Let e be a computation of Algorithm 3.1. A round in e is a factor of e in which any processor holds a token at least once.

COROLLARY 4. *In any computation of Algorithm 3.1 the maximal bound for a round length is* $B = (n-1) \cdot (n \cdot D^+)^{Diam} + 1$

4. MUTUAL EXCLUSION UNDER A K-BOUNDED SCHEDULER

In the sequel, we present a self-stabilizing mutual exclusion algorithm under a k-bounded scheduler (Algorithm 4.1). A scheduler is k-bounded iff while a given processor is enabled, another processor can perform at most k times its actions. This algorithm uses the routing policy previously presented but the token moves depend on a coin tossing.

4.1. Algorithm 4.1 description

The main difference with the random walks presented by Israeli and Jalfon in [17] is the fact that randomization is used here to decide whether or not the token will be sent (it is not used to decide to which of the neighbors it will be sent). The destination out-neighbor is still determined by the switch technique. The random walks method copes only with the undirected networks. Our method also copes with directed, strongly connected networks.

4.2. Algorithm 4.1 analysis

We prove Algorithm 4.1 weak self-stabilizing under a k-bounded scheduler for the mutual exclusion specification defined as follows :

DEFINITION 7 (Token circulation specification - $\mathcal{S}_{TC}$). In the network "there is only one token" and any processor in the network holds the token infinitely often.

Let us denote by $\mathcal{L}_{TC}$ the following predicate over configurations : there is exactly one token. All the configurations of Algorithm 4.1 which satisfy Predicate $\mathcal{L}_{TC}$ are called *legitimate configurations.*
According to Remark 2, we have :

LEMMA 5. *The predicate* $\mathcal{L}_{TC}$ *is closed for Algorithm 4.1.*

Convergence proof. In the following we prove Algorithm 4.1 convergence for $\mathcal{L}_{TC}$ under a k-bounded scheduler. In order to show the system convergence we prove that any system strategy *st* under a k-bounded scheduler verifies the local convergence property of Definition 3 for $\mathcal{L}_{TC}$.

DEFINITION 8. Let e be a computation of Algorithm 4.1. A *round* in e is a factor in which a token visits all processors.

Algorithm 4.1 Routing protocol for the probabilistic token for processor p

Shared registers with the in-neighbors :
$R_{in}^{PT}[1..D^-(p)]$ where $R_{in}^{PT}[i] \in [0, snd(n) - 1]$

Shared registers with the out neighbors :
$R_{out}^{PT}[1..D^+(p)]$ where $R_{out}^{PT}[j] \in [0, snd(n) - 1]$

Variables :
$direction^{PT} \in [0, D^+(p) - 1]$ (*the previous direction of the probabilistic token*)

Functions :
$\Delta PT = \left(\sum_{q \in In(p)} R_{in}^{PT}[q] - \sum_{q \in Out(p)} R_{out}^{PT}[q]\right) \bmod snd(n)$

Macros :
$New\_Dir\_Probabilistic\_Token :: direction^{PT} := (direction^{PT} + 1) \bmod D^+(p)$
$Pass\_Probabilistic\_Token :: R_{out}^{PT}[direction^{PT}] := (R_{out}^{PT}[direction^{PT}] + \Delta PT - 1) \bmod snd(n)$

Predicates :
$Probabilistic\_Token \equiv [\ \Delta PT \neq 1\ ]$

Actions :
$\mathcal{A}::\ Probabilistic\_Token \longrightarrow$
if (random(0, 1) = 0) then { $New\_Dir\_Probabilistic\_Token$; $Pass\_Probabilistic\_Token$ }

LEMMA 6. *Let st be a strategy of Algorithm 4.1 under a k-bounded scheduler. There exist $\epsilon > 0$ and $N \geq 1$ such that any st-cone verifies the property $LC(true, \mathcal{L}_{TC}, \epsilon, N)$.*

Proof. Let C_{h_1} be an arbitrary st-cone with $last(h_1) = c_1$. Assume that the number of tokens in c_1 is m. Denote by $(p_i)_{i=1,\ldots,m}$ the processors holding these tokens. Consider the following scenario : the token held by processor p_1 (called token t_1) merges with the token held by the processor p_2 (called token t_2). We prove that : (i) the scenario holds with positive probability and (ii) the scenario is repeated until there is only one token in the network.

- We call h_2 the computation from $last(h_1)$ having the following properties : (1) when the scheduler chooses the processor holding the token t_1 the result of coin tossing is 1 (hence the token circulates); (2) when the scheduler chooses another token the result of coin tossing is 0 (the token is frozen) (3) the moving token reaches p_2 in the last configuration of h_2. In $last(h_1h_2)$ the number of tokens is lesser than $m - 1$.

  The t_1 token circulates "pseudo-deterministically" : when a process holding the t_1 token, performs an action it releases the token. Therefore within B computation steps of t_1, the t_1 token has reached all processors.

  In the worst case, the scheduler chooses t_1 when it cannot do another choice : the other privileged processors have performed k actions (the scheduler is k-

bounded). Therefore, within $k \cdot (m-1)+1) \cdot B$ computation steps, the t_1 token reached all processors (i.e. has merged with another token). We have : $Pr_{st}(\mathcal{C}_{h_1 h_2}) \leq Pr_{st}(\mathcal{C}_{h_1}) \cdot (\frac{1}{2})^{(k \cdot (m-1)+1) \cdot B}$ and $length(h_2) \leq (k \cdot (m-1)+1) \cdot B$.

- By successive applications of the previous scenario we built some sub-cone $\mathcal{C}_{h_m}$. In $last(h_m)$, the number of tokens is 1, $Pr_{st}(h_m) \geq Pr_{st}(h_1) \cdot (\frac{1}{2})^{[(m-1)+\frac{k \cdot m \cdot (m-1)}{2}] \cdot B}$ and $length(h_m) \leq [(m-1)+\frac{k \cdot m \cdot (m-1)}{2}] \cdot B$

Therefore the property $LC(true, \mathcal{L}_{TC}, \epsilon, N)$ where $\epsilon \geq (\frac{1}{2})^{\frac{(k \cdot n^2+2 \cdot n) \cdot B}{2}}$ and $N \leq \frac{(k \cdot n^2+2n) \cdot B}{2}$ is verified. ∎

Remark 3. The previous result holds only under a k-bounded scheduler. Under an unfair scheduler, the Algorithm 5.1 does not converge to $\mathcal{L}_{TC}$. For example, on a directional ring, an unfair scheduler may have the following strategy : selects the same privileged processor till it passes its token; then selects another privileged processor till it passes its token, and so on. With this strategy, all the tokens move at the same speed in the ring; they will never merge.

LEMMA 7. *Algorithm 4.1 has a finite expected stabilization time.*

Proof. In order to establish the expected stabilisation time we use the technique presented in [14] and the ϵ value showed in Lemma 6, $\epsilon \geq (\frac{1}{2})^{\frac{(k \cdot n^2+2 \cdot n) \cdot B}{2}}$ (B provided by Corollary 4). The expected stabilisation times is bounded by $\frac{1}{\epsilon} \leq 2^{\frac{(k \cdot n^2+2 \cdot n) \cdot B}{2}}$. ∎

Remark 4. Note that majorations used in proving Lemma 6 are brutal, hence the provided exponential bound for the stabilisation time.

LEMMA 8. *Let st be a strategy of Algorithm 4.1 under a k-bounded scheduler. There exist $RT > 0$ and $\epsilon > 0$ such that any st-cone $\mathcal{C}_h$ with $last(h)$ is a legitimate configuration has a sub-cone $\mathcal{C}_{hh'}$ with $lenght(h') \leq RT$ such that h' is a round and $Pr_{st}(\mathcal{C}_{hh'}) \geq Pr_{st}(\mathcal{C}_h) \cdot \epsilon$.*

Proof. Let h' be the computation from $last(h)$ where the only token moves at each computation step until the token has visited all processors. The probability of $\mathcal{C}_{hh'}$ is $\epsilon_1 \geq Pr_{st}(\mathcal{C}_h)(\frac{1}{2})^B$. As this scenario is "pseudo-deterministic" the token reaches all processors in at most B computations steps, The $length(h') \leq B$. ∎

From Lemmas 8 and Theorem 1, we get :

COROLLARY 5. *In any strategy of Algorithm 4.1 under a k-bounded scheduler the probability of the set of computations satisfying :* (1) *a legitimate configuration is reached and* (2) *after reaching a legitimate configuration there are an infinite number of rounds, is 1.*

COROLLARY 6 (Correctness proof). *In any strategy of Algorithm 4.1 the probability of the set of computations reaching a legitimate configuration and satisfying $\mathcal{S}_{TC}$ is 1.*

THEOREM 2. *Algorithm 4.1 is weak self-stabilizing for the specification $\mathcal{S}_{TC}$.*

Proof. The weak correctness is provided by the Corollary 6, the convergence is provided by the Lemma 6 and the Theorem 1. ∎

Remark 5. Algorithm 4.1 satisfies only the weak correctness. It could be easily transformed in a strong self-stabilizing algorithm using the technique reported in [10]. Clearly, the expected steps of stabilisation is exponential.

5. LEADER ELECTION UNDER A K-BOUNDED SCHEDULER

Informally, a self-stabilizing distributed system which solves the leader election problem must satisfy the property that once the system is stabilized there is only one, unchanged leader. Formally, this specification is defined as follows :

DEFINITION 9 (Leader election specification - $\mathcal{S}_{LE}$). Let $Leader\_Mark$ be a predicate over the local configurations. Any computation, e, of a self-stabilizing system verifies the leader election specification if and only if the two following properties holds : *(1)* e reaches a configuration, where the $Leader\_Mark$ predicate is true for one and only one processor, p (also called leader), and *(2)* in any configuration occurring afterward in e, p is always the unique leader.

In the following, we present an algorithm for leader election which stabilizes under a k-bounded scheduler.

5.1. Algorithm 5.1 description

Algorithm 5.1 has two distinct layers of tokens. The first (respectively second) layer ensures the circulation of $Leader\_Mark$ (respectively $Colored\_Token$) tokens following the routing policy of Section 4. Once the algorithm is stabilized, the $Leader\_Mark$ is frozen and the $Colored\_Token$ keeps circulating.

$Colored\_Token$ and $Leader\_Mark$ have a virtual "color" attribute. Each token has a different role and then colors are managed independently.

A processor holding a $Leader\_Mark$ token is considered as a leader. The color of the $Leader\_Mark$ token is the color of the processor holding it.

A $Colored\_Token$ is used in order to detect the presence of some other leaders. The value of the color attribute for $Colored\_Token$ is the color of the processor having passed the token (an in-neighbor of the processor holding the colored token). A processor, p, keeps a copy of the previous values of its in-registers (in the variable Old_{in}^{CT}), in order to find the sender of the colored token (only the sender changed the value of the corresponding out-register). When several colored tokens meet on the same processor, the color of the resulting colored token is the color of the first processor (according to the local switching order) that has sent a color token.

During its circulation a $Colored\_Token$ colors all the non leader processors with its color ($\mathcal{A}_3$). A leader which has sent a $Colored\_Token$ waits until it returns. At that time, if the color of $Colored\_Token$ is the same as its color, then it stays a leader but goes on checking by randomly selecting a new color and starting a new circulation of the colored token (Action $\mathcal{A}_2$). In this case, it has no information telling it that it is not the single leader.

Since color is randomly selected, if there are several leaders in the network, a leader will eventually get a colored token that does not have its color. In this case, the leader passes its leadership and $Colored\_Token$ with a new randomly chosen color (Action $\mathcal{A}_1$). In this case it supposes that there are several leaders.

Once the algorithm is stabilized, there remains only one frozen leader and only one colored token which may circulate.

5.2. Algorithm 5.1 analysis

Let us define the following predicates over configurations :

- $\mathcal{L}_{CT} \equiv$ there is exactly one colored token;

Declaration 1 Registers, variables, predicates and macros for p executing Algorithm 5.1

Shared registers with the in-neighbors :

$R_{in}^{LM}[1..D^-(p)]$ where $R_{in}^{LM}[i] \in [0, snd(n)-1]$ (for leader mark)
$R_{in}^{CT}[1..D^-(p)]$ where $R_{in}^{CT}[i] \in [0, snd(n)-1]$ (for colored token)
$R_{in}^{color}[1..D^-(p)]$ where $R_{in}^{color}[i] \in \{0,1\}$ (for the color)

Shared registers with out-neighbors :

$R_{out}^{LM}[1..D^+(p)]$ where $R_{out}^{LM}[j] \in [0, snd(n)-1]$ (for leader mark)
$R_{out}^{CT}[1..D^+(p)]$ where $R_{out}^{CT}[j] \in [0, snd(n)-1]$ (for color token)
$R_{out}^{color}[1..D^+(p)]$ where $R_{out}^{color}[j] \in \{0,1\}$ (for the color)

Variables :

$direction^{PT} \in [0, D^+(p)-1]$ (*the previous direction of a probabilistic token*)
$Old_{in}^{CT}[1..D^-(p)]$ (*the old values from the registers* R_{in}^{CT})
$color$ is a boolean : *0 = red and 1 = green* (*the color of the processor* p)

Functions :

$\Delta LM = \left(\sum_{q \in In(p)} R_{in}^{LM}[q] - \sum_{q \in Out(p)} R_{out}^{LM}[q]\right) \bmod snd(n)$
$\Delta CT = \left(\sum_{q \in In(p)} R_{in}^{CT}[q] - \sum_{q \in Out(p)} R_{out}^{CT}[q]\right) \bmod snd(n)$

Macros :

$New\_Dir\_Probabilistic\_Token :: direction^{PT} := (direction^{PT} + 1) \bmod D^+(p)$
$Pass\_Leader\_Mark :: R_{out}^{LM}[direction^{PT}] := (R_{out}^{LM}[direction^{PT}] + \Delta LM - 1) \bmod snd(n)$
$Pass\_Colored\_Token :: R_{out}^{CT}[direction^{PT}] := (R_{out}^{CT}[direction^{PT}] + \Delta CT - 1) \bmod snd(n)$
$Update\_Old :: \forall i \in [1..D^-(p)] Old_{in}^{CT}[i] := R_{in}^{CT}[i]$
$Randomly\_Change\_Color :: color := random(red, green); \forall j \in [1..D^+(p)], R_{out}^{color}[j] := color;$
$Change\_Color :: color := R_{in}^{color}[i]$, where $i \in [1..D^-(p)]$ such that $Old_{in}^{CT}[i] \neq R_{in}^{CT}[i]$; $\forall j \in [1..D^+(p)], R_{out}^{color}[j] := color;$

Predicates :

$Leader\_Mark \equiv [\ \Delta LM \neq 1\ ]$
$Colored\_Token \equiv [\ \Delta CT \neq 1\ ]$
$Same\_Color \equiv [\ color = R_{in}^{color}[j]$ *where* $j \in [1..D^-(p)]$ such that $Old_{in}^{CT}[j] \neq R_{in}^{CT}[j]\ ]$

Algorithm 5.1 Randomized leader election algorithm under a k-bounded scheduler

Actions :

$\mathcal{A}_1$:: $Leader\_Mark \wedge Colored\_Token \wedge \neg Same\_Color \longrightarrow$
if (random(0, 1) = 0) then
{ $Randomly\_Change\_Color$; $Update(Old)$;
$New\_Dir\_Probabilistic\_Token$; $Pass\_Leader\_Mark$;
$Pass\_Colored\_Token$ }

$\mathcal{A}_2$:: $Leader\_Mark \wedge Colored\_Token \wedge Same\_Color \longrightarrow$
if (random(0,1) = 0) then
{ $Randomly\_Change\_Color$; $Update(Old)$;
$New\_Dir\_Probabilistic\_Token$; $Pass\_Colored\_Token$ }

$\mathcal{A}_3$:: $Colored\_Token \wedge \neg Leader\_Mark \longrightarrow$
if ($random(0,1) = 0$) then
{ $Change\_Color$; $Update(Old)$;
$New\_Dir\_Probabilistic\_Token$; $Pass\_Colored\_Token$ }

- $\mathcal{L}_{Color} \equiv$ *(i)* on any processor, for any value $j \in [1..D^+(p)]$, we have : $R_{out}^{color}[j] = color$ and *(ii)* on any processor, except the processor having the colored token p, we have $Old_{in}^{CT} = R_{in}^{CT}$; and on p, for any value $j \in [1..D^-(p)]$, except one, we have : $Old_{in}^{CT}[j] = R_{in}^{CT}[j]$.
- $\mathcal{L}_{LM} \equiv$ there is exactly one leader mark;
- $Same\_Color \equiv$ the unique leader mark and the unique colored token have the same color.

Definition 10. Let us denote by $\mathcal{L}_{LE}$ the predicate which is true when the following four predicates hold : *(1)* $\mathcal{L}_{CT}$, *(2)* $\mathcal{L}_{Color}$, *(3)* $\mathcal{L}_{LM}$, and *(4)* $Same\_Color$.

A legitimate configuration for Algorithm 5.1 is a configuration satisfying the predicate $\mathcal{L}_{LE}$.

Remark 6. Predicates $\mathcal{L}_{CT}$ and $\mathcal{L}_{LM}$ are closed.

Lemma 9. *Let st be a strategy of Algorithm 5.1 under a k-bounded scheduler. We have* $Pr_{st}(\mathcal{L}_{CT}) = 1$.

Proof. The colored token follows the routing policy as it was defined in the section 4. According to the Lemma 6 : $\exists \delta_{st} > 0$ and $\exists n_{st} \geq 1$ such that any st-cone satisfies the LC $(true, \mathcal{L}_{CT}, \delta_{st}, n_{st})$ property. We get $Pr_{st}(\mathcal{L}_{CT}) = 1$ by Theorem 1. ∎

Lemma 10. *Let st be a strategy of Algorithm 5.1;* $Pr_{st}(\mathcal{L}_{CT} \wedge \mathcal{L}_{color}) = 1$.

Proof. Let $\mathcal{C}_h$ be an arbitrary cone of the strategy st such as $last(h)$ satisfies the predicate $\mathcal{L}_{CT}$. Let $\mathcal{C}_{hh'}$ be an arbitrary sub-cone of $\mathcal{C}_h$. Let p be a processor that has performed an action in h'. After the p's action, p satisfies (i) for any value $i \in [1..D^+(p)]$, we have : $R_{out}^{color}[j] = color$ and (ii) $Old_{in}^{CT} = R_{in}^{CT}$ until one of its in-neighbor gives it the colored token. In this case, on p, for any index $j \in [1..D^+(p)]$, but one (called l), $Old_{in}^{CT}[j] = R_{in}^{CT}[j]$. $R_{in}^{CT}[l]$ is the in-register of p corresponding to the processor that has given the colored token to p.

We call h'' the computation from $last(h)$ where the colored token moves at each computation step until all processors have got the colored token. The configuration $last(xh'')$ verifies the predicate $\mathcal{L}_{CT} \wedge \mathcal{L}_{Color}$. The probability to obtain the cone $\mathcal{C}_{hh''}$ is $\epsilon_1 \geq Pr_{st}(\mathcal{C}_h)(\frac{1}{2})^B$. The $length(h'') \leq B$. ∎

LEMMA 11. *Predicates $\mathcal{L}_{CT} \wedge \mathcal{L}_{Color} \wedge \mathcal{L}_{LM}$ and $\mathcal{L}_{CT} \wedge \mathcal{L}_{Color}$ are closed for Algorithm 5.1.*

Proof. Let e be an arbitrary execution of Algorithm 5.1. Let c be a configuration satisfying the predicate $\mathcal{L}_{CT} \wedge \mathcal{L}_{Color}$ in e. In c, only the processor p that has the colored token has an index l such that $Old_{in}^{CT}[l] \neq R_{in}^{CT}[l]$. Only p may perform an action. In all cases, after the action of p, the *color* out-register of p has the same value as its *color* variable. After this action, either no variable value is changed : the predicate $\mathcal{L}_{CT} \wedge \mathcal{L}_{Color}$ is still verified. Or p updates the variable Old and gives the colored token to a neighbor q : now only the processor q has an index l' such that $Old_{in}^{CT}[l'] \neq R_{in}^{CT}[l']$ ($R_{in}^{CT}[l']$ being the out-register of p shared with q). Therefore, the next configuration in e is a configuration verifying $\mathcal{L}_{CT}$ and $\mathcal{L}_{LM}$. ∎

Remark 7. On a configuration c verifying the predicate $\mathcal{L}_{CT} \wedge \mathcal{L}_{Color}$, only Action $\mathcal{A}1$ or $\mathcal{A}2$ may change the color of the colored token.

NOTATION 3. Let us denote by $NLead(c)$ the number of leader marks in the configuration c.

LEMMA 12. *Let st be a strategy of Algorithm 5.1 under a k-bounded scheduler. There exist $\epsilon > 0$ and $N \geq 1$ such that any cone of st, $\mathcal{C}_h$ with $last(h) \vdash \mathcal{L}_{CT} \wedge \mathcal{L}_{Color}$, satisfies Local_Convergence ($\mathcal{L}_{CT} \wedge \mathcal{L}_{Color}$, $\mathcal{L}_{LM}$, ϵ, N).*

Proof. Assume that $last(h)$ does not satisfy the predicate $\mathcal{L}_{LM}$. $NLead(last(h)) = m$ with $m > 1$ and there is only one colored token in $last(h)$. The proof has the following informal steps : *(1)* we prove that with positive probability the colored token meets for the first time a processor holding a leader mark, let us denote this leader mark by lm_1, *(2)* with positive probability the leader mark, lm_1 circulates in the network until it merges with another leader mark. And we repeat the steps *(1)* and *(2)* until there is exactly one leader mark in the network.

Assume that in $last(h)$ the colored token is not on a leader. We call h' the computation from $last(h)$ where the colored token moves at each computation step until it reaches a leader (Action $\mathcal{A}3$). According to this scenario, the colored token is "pseudo-deterministic" : it moves at each computation step. The steps number to reach a leader is at most the length of a round of Algorithm 3.1. The probability of each computation step is $(\frac{1}{2})^B$. The probability of the cone $\mathcal{C}_{hh'}$ is $\epsilon_1 \geq Pr_{st}(\mathcal{C}_h)(\frac{1}{2})^B$. The $length(h') \leq B$ where B is the bound stated in Corollary 4. Once the colored token and the leader mark are on the same processor, there are two cases : *(a)* the colored token and the leader token have the same color, or *(b)* they have different colors (hh' is now called H).

- *a case* - Only Action $\mathcal{A}_2$ can be performed (by the leader having the color token - p). Let us called q the next leader that the colored token will meet. We study the history h'' where (1) p does not "choose" the color of q and (2) the colored token moves at each computation step until it reaches q. At the end, of this history, the case b is reached : the colored token and the leader mark are on the same processor, and they have different colors. The probability of the cone $\mathcal{C}_{Hh''}$ is $\epsilon_2 \geq Pr_{st}(\mathcal{C}_h)(\frac{1}{2})^{2B+1}$. The $length(h'h'') \leq 2B$. Now, Hh'' is called H.

- *b case* - Only Action $\mathcal{A}_1$ can be performed (by the processor having the color token - p). The probability that the processor p passes the both tokens to an out-neighbor q (having the color col) and colors the colored token with a color different of col is $\frac{1}{4}$. q is the same state as the p state before the move. q has the both tokens, but the color of the leader mark is not the color of the colored token. We call $h1$ the computation from $last(H)$ where the colored token and the leader mark move together until they meet another leader mark. The probability of the sub-cone $\mathcal{C}_{Hh1}$ of the cone $\mathcal{C}_H$ where $NLead(last(Hh1)) = m-1$ is $\epsilon_3 \geq Pr_{st}(\mathcal{C}_H)(\frac{1}{2})^{2B}$ and $length(h1) \leq B$.

The probability of the cone $\mathcal{C}_{hH_1}$ where $NLead(last(hH_1)) = m-1$ is $\epsilon'_1 \geq Pr_{st}(\mathcal{C}_h)(\frac{1}{2})^{4B+1}$. The $length(H_1) \leq 3B$. The probability of the cone $\mathcal{C}_{hH_{m-1}}$ where $NLead(last(hH_{m-1})) = 1$ is $\epsilon'_{m-1} \geq Pr_{st}(\mathcal{C}_h)(\frac{1}{2})^{(m-1)(4B+1)}$. $N_{m-1} = length(H_{m-1}) \leq 3(m-1)B$. Therefore, the property $Local\_Convergence(\mathcal{L}_{CT} \wedge \mathcal{L}_{Color}, \mathcal{L}_{LM}, \epsilon_{m-1}, N_{m-1})$ is satisfied. ∎

According to the Theorem 1, we have :

COROLLARY 7. *Let st be a strategy of Algorithm 5.1 under a k-bounded scheduler. We have* $Pr_{st}(\mathcal{L}_{CT} \wedge \mathcal{L}_{Color} \wedge \mathcal{L}_{LM}) = 1$.

LEMMA 13. *The predicate* $\mathcal{L}_{CT} \wedge \mathcal{L}_{Color} \wedge \mathcal{L}_{LM} \wedge Same\_Color$ *is a closed predicate for Algorithm 5.1.*

Proof. Let c be a configuration satisfying the predicate $\mathcal{L}_{CT} \wedge \mathcal{L}_{Color} \wedge \mathcal{L}_{LM} \wedge Same\_Color$. In c, there is a unique leader mark and only one colored token. Both have the same color. Only Action $\mathcal{A}2$ may change the color of the colored token. After that action, the both tokens have the same color. ∎

LEMMA 14. *Let st be a strategy of Algorithm 5.1 under a k-bounded scheduler. There exist* $\epsilon > 0$ *and* $N \geq 1$ *such that any cone of st,* $\mathcal{C}_h$ *such that* $last(h) \vdash L_{CT} \wedge L_{Color} \wedge L_{LM}$, *satisfies* $Local\_Convergence(L_{CT} \wedge L_{Color} \wedge L_{LM}, Same\_Color, \epsilon, N)$.

Proof. Assume that $last(h)$ does not satisfies the predicate $Same\_Color$. There are two cases; in the first case the leader mark and the colored token are on different processors, while in the second one the colored token and the leader mark are on the same processor.

In the first case, let h' be the computation from $last(h)$ where the colored token circulates until it reaches the leader. The probability of the cone $\mathcal{C}_{hh'}$ is $\epsilon \geq Pr_{st}(\mathcal{C}_h)(\frac{1}{2})^B$. The $length(h') \leq B$. Now both tokens are on the same processor. Assume now that the processors have different colors. The probability that p gives the both tokens to an out-neighbor q and colors the colored token with the q's color is $\frac{1}{4}$. After that, q is a leader, q has the colored token; and the colored token has the q's color.

Therefore the probability of cone $\mathcal{C}_{hh''}$ with $last(hh'') \vdash \mathcal{L}_{CT} \wedge \mathcal{L}_{Color} \wedge \mathcal{L}_{LM} \wedge Same\_Color$ is $\epsilon \geq Pr_{st}(\mathcal{C}_h)(\frac{1}{2})^{B+2}$ and $length(h'') \leq B+1$. ∎

LEMMA 15. *Algorithm 5.1 has a finite stabilisation time.*

Proof. Using Lemmas 6, 10, 12 and 14 the expected stabilisation time is bounded by $\frac{1}{\epsilon}$ where $\epsilon \leq (\frac{1}{2})^{\frac{kn^2+10nB+2n+4B+4}{2}}$ (B provided by Corollary 4). Therefore the expected stabilization time of Algorithme 5.1 is finite. ∎

LEMMA 16. *Any computation of Algorithm 5.1 starting in a legitimate configuration satisfies the specification* $\mathcal{S}_{LE}$.

Proof. Let e be a computation starting in a legitimate configuration c — the predicate $\mathcal{L}_{LE}$ is satisfied by c. The only applicable rules are those where the leader mark is not moved ($\mathcal{A}_2$ and $\mathcal{A}_3$), hence the problem specification is satisfied. ∎

THEOREM 3. *Algorithm 5.1 is self-stabilizing for the specification* $\mathcal{S}_{LE}$.

Proof. The convergence is given by Corollary 7, Lemmas 13 and 14 and Theorem 1. The correctness is given by Lemma 16. ∎

6. TOKEN BASED ALGORITHMS UNDER AN UNFAIR SCHEDULER

In this section, we extend Algorithms 4.1 and 5.1 to cope up with unfair scheduler. For this purpose, the idea of *cross-over* composition (introduced in [7]) is used to compose Algorithms 4.1 and 6.2 with a k-fair algorithm (see Definition 5). Algorithm 3.1 is an $(n \cdot D^+)^{Diam}$-fair algorithm under any unfair scheduler. 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 any unfair scheduler for specification $\mathcal{SP}$.

The *cross-over composition* combines two algorithms —the *weaker* and the *stronger*— to get a new algorithm. The algorithms are considered stronger or weaker according to their properties toward the scheduler. When an algorithm needs a special scheduler then it is considered "weaker". By contrary, when the algorithm is preserving its properties even under an unfair scheduler then it plays the "stronger" role.

In this paper, the stronger (Algorithm 3.1) supports the stronger adversary (unfair scheduler), while the weaker (Algorithms 4.1 or 5.1) provides its specification (token circulation or leader election) under a weaker adversary (the k-bounded scheduler).

Algorithm 6.1, the result of cross-over composition between Algorithm 3.1 and Algorithm 4.1, has the following actions [$\mathcal{A}$ is the label of Algorithm 4.1 rule and $\mathcal{FA}$ is the label of Algorithm 3.1 rule] :

- $\mathcal{B}_1 :: \langle guard\ \mathcal{A}\rangle \wedge \langle guard\ \mathcal{FA}\rangle \longrightarrow \langle statement\ \mathcal{A}\rangle; \langle statement\ \mathcal{FA}\rangle$
- $\mathcal{B}_2 :: \neg\langle guard\ \mathcal{A}\rangle \wedge \langle guard\ \mathcal{FA}\rangle \longrightarrow \langle statement\ \mathcal{FA}\rangle$

Algorithm 6.1 Randomized token circulation algorithm under unfair scheduler

Actions :

$\mathcal{B}_1$:: $Fair\_Token \wedge Probabilistic\_Token \longrightarrow$
$New\_Dir\_Fair\_Token$; $Pass\_Fair\_Token$;
if (random(0, 1) = 0) then { $New\_Dir\_Probabilistic\_Token$;
$Pass\_Probabilistic\_Token$ }

$\mathcal{B}_2$:: $Fair\_Token \wedge \neg Probabilistic\_Token \longrightarrow$
$New\_Dir\_Fair\_Token$; $Pass\_Fair\_Token$;

Algorithm 6.2, the result of cross-over composition between Algorithm 3.1 and Algorithm 5.1, has the following actions [$(\mathcal{A}_i)_{i=1,4}$ are the labels of the rules of Algorithm 5.1] :

- $\mathcal{C}_1 :: \langle guard\ \mathcal{A}_1\rangle \wedge \langle guard\ \mathcal{F}A\rangle \longrightarrow \langle statement\ \mathcal{A}_1\rangle; \langle statement\ \mathcal{F}A\rangle$
- $\mathcal{C}_2 :: \langle guard\ \mathcal{A}_2\rangle \wedge \langle guard\ \mathcal{F}A\rangle \longrightarrow \langle statement\ \mathcal{A}_2\rangle; \langle statement\ \mathcal{F}A\rangle$
- $\mathcal{C}_3 :: \langle guard\ \mathcal{A}_3\rangle \wedge \langle guard\ \mathcal{F}A\rangle \longrightarrow \langle statement\ \mathcal{A}_3\rangle; \langle statement\ \mathcal{F}A\rangle$
- $\mathcal{C}_4 :: \neg\langle guard\ \mathcal{A}_1\rangle \wedge \neg\langle guard\ \mathcal{A}_2\rangle \wedge \neg\langle guard\ \mathcal{A}_3\rangle \wedge \langle guard\ \mathcal{F}A\rangle \longrightarrow \langle statement\ \mathcal{F}A\rangle$

Algorithm 6.2 Randomized leader election algorithm under unfair scheduler

Actions :

$\mathcal{C}_1$:: $Fair\_Token \wedge Leader\_Mark \wedge Colored\_Token \wedge \neg Same\_Color \longrightarrow$
$New\_Dir\_Fair\_Token$; $Pass\_Fair\_Token$;
if (random(0, 1) = 0) then
{ $Randomly\_Change\_Color$; $Update(Old)$;
$New\_Dir\_Probabilistic\_Token$; $Pass\_Leader\_Mark$;
$Pass\_Colored\_Token$ }

$\mathcal{C}_2$:: $Fair\_Token \wedge Leader\_Mark \wedge Colored\_Token \wedge Same\_Color \longrightarrow$
$New\_Dir\_Fair\_Token$; $Pass\_Fair\_Token$;
if (random(0, 1) = 0) then
{ $Randomly\_Change\_Color$; $Update(Old)$;
$New\_Dir\_Probabilistic\_Token$; $Pass\_Colored\_Token$ }

$\mathcal{C}_3$:: $Fair\_Token \wedge \neg Leader\_Mark \wedge Colored\_Token \longrightarrow$
$New\_Dir\_Fair\_Token$; $Pass\_Fair\_Token$;
if (random(0, 1) = 0) then
{ $Change\_Color$; $Update(Old)$; $New\_Dir\_Probabilistic\_Token$;
$Pass\_Colored\_Token$ }

$\mathcal{C}_4$:: $Fair\_Token \wedge \neg Colored\_Token \wedge \neg Leader\_Mark \longrightarrow$
$New\_Dir\_Fair\_Token$; $Pass\_Fair\_Token$;

Algorithms 6.1 and 6.2 description. Algorithms 6.1 and 6.2 are the result of cross-over composition between Algorithms 5.1 (the weaker) and 3.1 (the stronger). The composed algorithm contains an extra layer which ensures the algorithm convergence under any unfair scheduler.

Algorithm 6.1 and 6.2 analysis. In [7], it is proven that the cross-over composition between a probabilistic algorithm self-stabilizing for a specification SP under a k-bounded scheduler, playing the weaker role, and a deterministic algorithm satisfying the k-fairness property, playing the stronger role, is a self-stabilizing algorithm for SP under any unfair scheduler.

THEOREM 4. *Algorithms 6.1 and 6.2 are self-stabilizing for the specifications $\mathcal{S}_{TC}$ and $\mathcal{S}_{LE}$ respectively under an unfair scheduler.*

LEMMA 17. *The stabilization time for Algorithms 6.1 and 6.2 is finite.*

Proof. The expected stabilization time for Algorithms 6.1 and 6.2 is bounded by the values provided by Lemmas 7 and 15 with $k = (n \cdot D^+)^{Diam}$ and $B = (n-1) \cdot (n \cdot D^+)^{Diam} + 1$ (corollaries 3 and 4). ∎

7. CONCLUSION

This work focuses on token based self-stabilizing algorithms. The considered networks are anonymous and directed. For this type of networks, we present a token management and routing policy as solutions to the open problem proposed by Israeli and Jalfon in [17]. Note that the current paper proposes the first general solution for this problem. Moreover, we present self-stabilizing algorithms for mutual exclusion and leader election on anonymous, directed networks based on this policies. In order to break the symmetry we use randomization. One of the proposed algorithms is weak self-stabilizing for the mutual exclusion specification, the other one is self-stabilizing for the leader election specification. The both algorithms function correctly under any unfair distributed scheduler. Finally, we present a probabilistic analysis for the proposed algorithms.

All the results are summarized in the following table (F.T.C. : fair token circulation, L.E. : leader election, M.E. : mutual exclusion) where the space complexity is given in number of states per processor.

Algorithm	**Scheduler**	**Correctness**	**Space Complexity**
F.T.C. (3.1)	unfair	deterministic	$n \cdot D^+ \cdot snd^{D^+}$
M.E. (4.1)	k-bounded	weak prob.	$n \cdot D^+ \cdot snd^{D^+}$
L.E. (5.1)	k-bounded	strong prob.	$n \cdot D^+ \cdot 2^{D^+ + 1} \cdot snd^{2 \cdot D^+ + D^-}$
M.E. (6.1)	dist. unfair	weak prob.	$n \cdot D^{+^2} \cdot snd^{2 \cdot D^+}$
L.E. (6.2)	dist. unfair	strong prob.	$n \cdot D^{+^2} \cdot 2^{D^+ + 1} \cdot snd^{3 \cdot D^+ + D^-}$

The space complexity of our algorithms is $O((D^+ + D^-) \cdot (log(snd(n)) + 2))$ bits per processor. Note that snd(n) (the smallest non divisor of n) is constant in the average and equals 2 for odd size networks.

Our algorithms are space optimal for the ring topology as it was proven in [6].

REFERENCES

[1] R. Aleliunas, R.M.Karp, R. Lipton, L. Lovasz, and C. Rackoff, Random walks, universal traversal sequences and the complexity of the maze problem, *in* "FOCS'79, Proceedings of the 20st Annual IEEE Symp. on Foundation of Computer Science," pp. 218–223, 1979.

[2] D. Alstein, J.H. Hoepman, B. Olivier, and P. Put, Self-stabilizing mutual exclusion on directed graphs, Technical Report 9513, CWI Amsterdam, 1995.

[3] D. Angluin, Local and global properties in networks of processors, *in* "STOC'80, Proceedings of the 12th Annual ACM Symp. on Theory of Computing," pp. 82–93, 1980.

[4] B. Awerbuch and R. Ostrovsky, Memory-efficient and self-stabilizing network reset, *in* "PODC'94, Proceedings of the 13th Annual ACM Symp. on Principles of Distributed Computing," pp. 254–263, 1994.

[5] J. Beauquier, S. Cordier, and S. Delaët, Optimum probabilistic self-stabilization on uniform rings, *in* "WSS'95, Proceedings of the Second Workshop on Self-Stabilizing Systems", pp. 15.1–15.15, 1995.

[6] J. Beauquier, M. Gradinariu, and C. Johnen, Memory space requirements for self-stabilizing leader election protocols, *in* "PODC'99, Proceedings of the 18th Annual ACM Symp. on Principles of Distributed Computing," pp. 199–208, 1999.

[7] 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.

[8] J. Beauquier, S. Kutten, and S. Tixeuil, Self-stabilization in eulerian networks with cut-through constraints, Technical Report 1200, Laboratoire de Recherche en Informatique, January 1999.

[9] K. Chandy and J. Misra, "Parallel Programs Design: A Foundation," Addison-Wesley, New York, 1988.

[10] A.K. Datta, M. Gradinariu, and S. Tixeuil, Self-stabilizing mutual exclusion using unfair distributed scheduler, *in* "IPDPS'00, Proceedings of the 14th Int. Parallel and Distributed Processing Symp.," pp. 465–470, 2000.

[11] E. Dijkstra, Self stabilizing systems in spite of distributed control, *Communications of the ACM*, **17**, 11 (Nov. 1974), 643–644.

[12] S. Dolev, A. Israeli, and S. Moran, Self-stabilizing of dynamic systems assuming only read/write atomicity, *Distributed Computing*, **7**, 1 (1993), 3–16.

[13] S. Dolev, A. Israeli, and S. Moran, Analyzing expected time by scheduler-luck games, *IEEE Trans. on Software Engineering*, **21**, 5 (May 1995), 429–439.

[14] S. Dolev, A. Israeli, and S. Moran, Uniform dynamic self-stabilizing leader election, *IEEE Trans. Parallel Distrib. Systems*, **8**, 4 (April 1997), 424–440.

[15] J. Durand-Lose, Randomized uniform self-stabilizing mutual exclusion *Inform. Process. Lett.*, **74**, 5-6 (June 2000), 203–207.

[16] T. Herman, Probabilistic self-stabilization, *Inform. Process. Lett.*, **35**, 2 (June 1990), 63–67.

[17] A. Israeli and M. Jalfon, Token management schemes and random walks yield self-stabilizing mutual exclusion, *in* "PODC'90, Proceedings of the 9th Annual ACM Symp. on Principles of Distributed Computing," pp. 119–131, 1990.

[18] G. Itkis and L. Levin, Fast and lean self-stabilizing asynchronous protocols, *in* "FOCS'94, Proceedings of the 35th Annual IEEE Symp. on Foundations of Computer Science," pp. 226–239, 1994.

[19] H. Kakugawa and M. Yamashita, Uniform and self-stabilizing token rings allowing unfair daemon, *IEEE Trans. Parallel Distrib. Systems*, **8**, 2 (Feb. 1997), 154–162.

[20] A. Mayer, Y. Ofek, R. Ostrovsky, and M Yung, Self-stabilizing symmetry breaking in constant-space, *in* "STOC'92, Proceedings of the 24th Annual ACM Symp. on Theory of Computing," pp. 667–678, 1992.

[21] L. Rosaz, Self-stabilizing token circulation on an asynchronous unidirectional ring, *in* "PODC'00, Proceedings of the 19th Annual ACM Symp. on Principles of Distributed Computing," pp. 249–258, 2000.

[22] R. Segala, Modeling and verification of randomized distributed real-time Systems, Ph.D. thesis, MIT, Department of Electrical Engineering and Computer Science, 1995.

[23] R. Segala and N. Lynch, Probabilistic simulations for probabilistic processes, *in* "CONCUR'94, Concurrency Theory, 5th International Conference," Lecture Notes in Computer Science, Vol 836, Springer-Verlag, 1994.

[24] S. H. Wu, S. A. Smolka, and E. W. Stark, Composition and behaviors of probabilistic I/O automata, *in* "CONCUR'94, Concurrency Theory, 5th International Conference," Lecture Notes in Computer Science, Vol 836, Springer-Verlag, 1994.

JOFFROY BEAUQUIER is a professor at Paris-Sud university, where he teaches Operating System and Distributed Algorithms. Former student of Maurice Nivat, he first published in formal language theory. For the last ten years, he has been studying distributed algorithms, particularly self-stabilization. Every spring, he runs Paris marathon.

JÉRÔME DURAND-LOSE received its Master's degree from the École Normale Supérieure de Lyon, France, in 1992 and presented his Ph.D. at the University of Bordeaux in 1996, and is, from 1998 on, assistant professor at the University of Nice-Sophia Antipolis. His researches deal with distributed computing and theoretical models of parallel and distributed computing.

MARIA GRADINARIU received her Master's degree in computer science from the "Al. I Cuza" University, Iasi, Romania in 1997 and her Ph.D. from the University Paris-Sud, France in 2000. She is currently assistant professor of computer science at University of Rennes, France. Her research focuses on the deterministic and probabilistic distributed protocols.

COLETTE JOHNEN is an assistant professor in computer science at Paris-Sud University, France since 1987. She received her PH.D. in computer science from the Paris-Sud University in 1987, her M.S. in applied mathematics from the Paris-Sud University in 1984. Until 1990, her main research areas were analysis of Petri Net, and protocol verification. Currently, her research interests include theoretical models of distributed system, distributed algorithms, and routing in communication network.

JOFFROY BEAUQUIER is a professor at Paris-Sud university, where he teaches Operating System and Distributed Algorithms. Former student of Maurice Nivat, he first published in formal language theory. For the last ten years, he has been studying distributed algorithms, particularly self-stabilization. Every spring, he runs Paris marathon.