Deterministic, silence and self-stabilizing leader election algorithm on id-based rings

Colette Johnen

L.R.I./C.N.R.S. Université de Paris-Sud, Bat. 490, Campus d'Orsay, F-91405 Orsay Cedex, France, phone : (+33) 1 69 15 67 02, fax : (+33) 1 69 15 65 86, colette@lri.fr,
Research Note : NO

Abstract. We present in this paper a deterministic, silence and self-stabilizing leader election algorithm on unidirectional, id-based rings which have bound on their id-values. The id-values of processors in a ring whose size is N, have to be inferior to $N + k$. The size of communication registers required by the algorithm is constant. The algorithm stabilizes in $(k + 2)N + 1$ time.
In [DGS96], Dolev, Gouda and Schneider have established that the memory requirement of a silent self-stabilizing leader election protocol is $O(\lg N)$ in the general case. Here, we present a silent algorithm requiring constant memory space: We conjecture that we have found the only case where a silent protocol requiring constant memory space can be designed: id-based rings with bound id-values.
Topic: 10 (Distributed Systems and Algorithms),
keywords: self-stabilization, leader election, silent algorithm, memory space.
Comments: other topic of interest : 20 (Fault Tolerant Computing).

1 Introduction

Robustness is one of the most important requirements of modern distributed systems. Various types of faults are likely to occur at various parts of the system. These systems go through the transient faults because they are exposed to constant change of their environment (memory corruptions, processors and communication-channels crashing and recovering -i.e. dynamic networks-). The concept of self-stabilization [Dij74] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. Such a property is very desirable for any distributed system, because after any unexpected perturbation modifying the memory state, the system eventually recovers and returns to a legitimate state, without any outside intervention. Furthermore, self-stabilizing systems do not require particular initial state when a processor recovers assuming that processor codes are not corrupted.

In this paper, we present a deterministic, silence and self-stabilizing algorithm that elects the processor having the minimal id-value on id-based unidirectional N-rings such that the id-values of its processors is inferior to $N + k$.

In an id-based system, as processor ids are not part of the local variables, whatever is the current state each processor as distinct identity. There is several deterministic self-stabilizing leader election protocols on id-based systems whose the memory space requirement is N states per processor. Some of these algorithms build a spanning tree - the leader will be the tree root - [AG93] (in that protocol, processors need to know a bound on the network size, the same protocol is presented in the read/write model [AG94]) in [AKY90] the knowledge of ring size is not required. the algorithm presented in [Dol93] assume only the read/write atomicity and processors does not need to know the network size. Ghosh and Gupta have designed a self-stabilizing algorithm on unidirectional rings that contained faults: after a fault, the ring will recovers in $O(1)$ time [GG96].
Mayer, Ostrovsky and Yung, in [MOY96] have proposed a compiler that transforms any self-stabilizing protocol on bidirectional uniform rings to a self-stabilizing protocol which run on unidirectional uniform rings (their compiler to break the symmetric flip coins). Afek and Bremler in [AB97] have proposed a general paradigm for the development of self-stabilizing algorithm on unidirectional general id-based processors that communicate by messages-passing. The size of the exchanged messages is $log(N)$.

There are two principal measures of efficiency for self-stabilizing algorithms: stabilization time, and memory requirements per processor. On huge, distributed networks (containing several millions of processors) managed by several organizations, the properly functioning of network management protocols should not depend on global properties (as network size) which can be modified at any time, by anybody. Therefore, we propose an algorithm whose space complexity is constant (thus it is independent of ring size). When, the ring grows, the local algorithm implementations do not need to be changed. There is several self-stabilizing protocols where the required memory space is constant [GH96], [JABD97], [Joh97], [Pet97], [PV97], and [Vil97]. On general uniform rings, there is no deterministic and self-stabilizing leader election protocol ([BJ98]). Only on bidirectional and prime size rings, one can designed a protocol requiring only constant memory space (such a protocol is presented in [ILS95]). To my knowledge, we present here the first self-stabilizing leader election algorithm on id-based rings where the required memory space is constant.

A self-stabilizing algorithm is *silent* if once the system is stabilized, processors do not change their state (processors only check that their neighbor states have not be corrupted). The silence property of self-stabilizing algorithms is a desirable property in terms of simplicity and communication overhead. Our algorithm completes, in some sense, the result by Dolev, Gouda and Schneider [DGS96]. They state that the memory requirement to a silent self-stabilizing leader election protocol is $O(\log N)$ in the general case (i.e. when there is no bound on the id-values).

Section 2 describes our model. In section 3, we present the algorithm. Section 4 is devoted to the proof of convergence and section 5 to the proof of correctness. Finally, the paper ends with complexity proofs.

2 Model

Distributed Unidirectional Ring. A *distributed unidirectional ring* is a connected ring of processors. A processor may only obtain "information" from its left neighbor, and give "information" to its right neighbor. Communication among processors is carried out by communication registers. A processor can write and read in its register, and read the register of its left neighbor. Thus, processors have two types of variables: *local variables* and *field variables*. The field variables of a processors are in its register (which is readed by its right neighbor). The local variables are used strictly locally, meaning that they can be only accessed (writing/reading) by their owner. The path from p' to p is the processors sequence $(p_1, p_2, ..., p_m)$ such that (i) $p' = p_1$, (ii) $p = p_m$, (iii) $\forall j \in]1,m[$, the left neighbor of p_j is p_{j-1} and its right neighbor is p_{j+1}.
On bidirectional rings, a processor may give and obtain "information" from its both neighbors: it can read the communication register of its both neighbors. An algorithm on unidirectional rings may be performed on bidirectional rings; the converse is not true.
States and Configurations. The *register state* of processor is defined by the values of its field variables. The *state* of processor is defined by the values of its local variables and field variables. A *configuration* of an unidirectional ring is a product of the states of all processors of ring. The set of configurations of the ring is denoted as $\mathcal{C}$.
Id-based ring. In an *id-based ring*, processors have distinct identities: processor are distinct. An id-based ring whose size is N is *k-bounded* if and only if id-values of its processors is inferior or equal to $N + k$.
Actions. Each processor executes an algorithm. The algorithm consists of a set of variables and a finite set of actions. Each action is uniquely identified by a label; and is of the following form: $< label >:: < guard > \longrightarrow < statement >$

The guard of an action in the algorithm of p is a boolean expression involving the local, field variables of p, and the field variables of its left neighbor. The statement of an action of p updates some local variables and/or field variables of p. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed: the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.
Computations. During a *computation step*, one or more processors execute an atomic step, and a processor may take at most one atomic step; this is known as the *distributed daemon* [BGM89]. Since the system is asynchronous, we define a time unit, *round*, to compute the time complexity. A *round* is a maximal computation step where all processors that hold the guard of an action, execute the corresponding action during the step. A round is called by some authors a synchronous computation step. A *computation* e of an algorithm $\mathcal{A}$ is a *fair, maximal* sequence of configurations $c_1, c_2, \ldots$ such that for $i = 1, 2, \ldots$, the configuration c_{n+1} is reached from c_n by a computation step. c_1 is called the *initial configuration* of e. Along a *fair* sequence, if a processor may continuously perform an action then it will eventually perform an action. A *maximal* sequence is

either infinite, or it is finite; in that case no action is enabled in the final configuration. The set of computations of an algorithm $\mathcal{A}$ starting with a particular initial configuration $c \in \mathcal{C}$ is denoted by $\mathcal{E}_c$. The set of computations of $\mathcal{A}$ whose initial configurations are elements of $B \subset \mathcal{C}$ is denoted as $\mathcal{E}_B$. $\mathcal{E}$ is the set of all possible fair and maximal computations ($\mathcal{E} = \mathcal{E}_{\mathcal{C}}$).

Register space complexity. The register space complexity of a self-stabilizing algorithm is the number of register states of a processor performing the self-stabilizing algorithm. An algorithm requires only constant register space if the number of register states on each processor required by the algorithm is a constant.

Predicates. Let $\mathcal{X}$ be a set. $x \vdash P$ means that an element $x \in \mathcal{X}$ satisfies the predicate P defined on the set $\mathcal{X}$. We distinguish a special predicate: true (satisfied by each element of $\mathcal{X}$) formally defined as follows: $x \vdash \mathsf{true}$.

Self-Stabilization. We use the following term, *attractor* in the definition of self-stabilization.

Definition 1 Attractor. Let X and Y be two predicates defined on $\mathcal{C}$. In $\mathcal{C}$, Y is an attractor for X if and only if the following conditions are true:

- **convergence** $\forall e \in \mathcal{E}_X : \ (e = c_1, c_2, \ldots) :: \ \exists n \geq 1, c_n \vdash Y$
- **closure** $\forall c \vdash Y : \forall c'$ where (c,c') is a computation step:: $\ c' \vdash Y$

Let $\mathcal{LS}$ be a predicate defined on $\mathcal{C}$. The system self-stabilizes to $\mathcal{LS}$ if only if $\mathcal{LS}$ is attractor for true.

Definition 2 trap. Let Pr be a predicate defined on processor state. Pr is a *trap* if only if Pr verifies the closure property: Let c be a configuration where $p \vdash Pr$, $\forall c'$ where (c,c') is a computation step:: $\ p \vdash Pr$ in c'.

To prove the correctness of our algorithm, we use the *convergent stair* [GM91] theorem.

Theorem 3. *Let Y and X be two predicates defined on $\mathcal{C}$. if X is an attractor for* true *and if Y is and for X then Y is an attractor for* true.

Time complexity The temporal activities of a self-stabilizing protocol can be divided into three phases: (i) The *fault* phase during which faults occur in the system (these faults corrupt the variables value of processors); (ii) the *stabilizing* phase during which the system does not exhibit the correct behavior (however, no new fault occurs during this phase); (iii) the *stabilized* phase during which every computation satisfies the specification predicate ($\mathcal{LS}$). The time complexity of a self-stabilizing algorithm is the maximal number of rounds needed to reach the stabilized phase after the faults cease to occur.

3 Algorithm

In this section, we present a deterministic, silence and self-stabilizing algorithm that elects the processor having the minimal id-value on id-based unidirectional rings with a k-bound on its id-values.

On a k-bounded ring, the value of the smallest id-value is inferior to $k+2$ (id-values are greater or equal to 0). Only processors whose id-value is inferior to $k+2$ can be a leader; other processors will never be elected. Thus, we define two processors sets. One set ($\mathcal{SI}$) contains the processors that may have the smallest id-value; only these processors compete to the leadership. The other set ($\mathcal{BI}$) contains processor that cannot be the leader; these processors do not compete to the leadership and are as "quiet" as possible. Formally $\mathcal{SI}$: the set of processors whose id-value is inferior or equal to to $k+1$; $\mathcal{BI}$: the set of processors whose id-value is greater than $k+1$.

The algorithm is designed such way that once the ring is stabilized processors know id-value of each processor of $\mathcal{SI}$ (these values are stored in an array called F). Thus, each processor knows the smallest id-value in the ring (the id-value of the leader). Nevertheless, a processor knows at most $k+2$ id-values (all of them inferior to $k+2$). That why, the register size required by the algorithm is constant whatever is the ring size.

Leader election algorithm on k-bounded rings

Field Variables:

F_p is an array of $k+2$ elements taking value in $[0, k+2[$.
Ld_p is a boolean.

Notation:

lp is the left neighbor of p.
F_{lp} is the value of F on lp.
id_p is the value of the p identifier.

Predicate:

$Next(p, p') \equiv (\forall i \in [0, k+1[: F_p[i] = F_{p'}[i+1]) \ \wedge \ (F_p[k+1] = id_p)$.
$Following(p) \equiv Next(p, lp)$.

Macro:

$Update(p) : \forall i \in [0, k+1[: F_p[i] := F_{lp}[i+1] \, ; F_p[k+1] := id_p$.

Action:

{*The two following actions are executed by the processors of* $\mathcal{BI}$}

$\mathcal{A}_1$: $(id_p > k+1) \wedge (F_p \neq F_{lp}) \longrightarrow F_p := F_{lp}$.

$\mathcal{A}_2$: $(id_p > k+1) \wedge (Ld_p = 1) \longrightarrow Ld_p := 0$.

{*The three following actions are executed by the processors of* $\mathcal{SI}$}

$\mathcal{B}_1$: $(id_p \leq k+1) \ \wedge \ \neg Following(p) \longrightarrow Update(p)$

$\mathcal{B}_2$: $(id_p \leq k+1) \ \wedge \ Following(p) \ \wedge \ id_p$ is not the smallest value in F_p
$\wedge \ Ld_p = 1 \longrightarrow Ld := 0$

$\mathcal{B}_3$: $(id_p \leq k+1) \ \wedge \ Following(p) \ \wedge \ id_p$ is the smallest value in F_p
$\wedge \ Ld_p = 0 \longrightarrow \ Ld_p := 1$

Definition 4.
A leader is a processor p such that $Ld_p = 1$.
A configuration is a deadlock if only if no processor may perform an action.
We call $\mathcal{LS}$, the set of deadlock configurations where one and only one processor is elected.

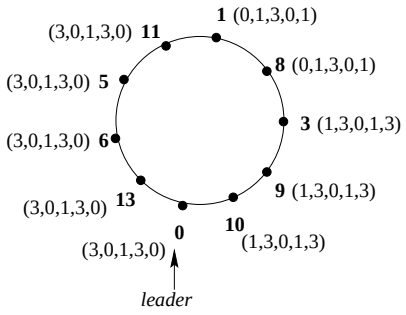

Fig. 1. the deadlock configuration in a 10-Ring that is 3-bounded

The processors of $\mathcal{BI}$ "transmit" the array ($\mathcal{A}_1$); and they also set up their Ld variable to 0 ($\mathcal{A}_2$). The processors of $\mathcal{SI}$ left-shift the F array elements (the first value is withdrawn); and then they add their own id-value at the end of the array[1] ($\mathcal{B}_1$). This simple action will ensure that (once the ring is stabilized) the array of each processor contains all the id-values of $\mathcal{SI}$ processors that are in the ring and only these id-values; though processors of $\mathcal{SI}$ have different arrays (the id-values are not in the same order), Thus once the ring is stabilized, a processor of $\mathcal{SI}$ may decide if it is the leader or not: it compares its id-value with the smallest id-value of its F variable ($\mathcal{B}_2$ or $\mathcal{B}_3$). In the section 4 we prove that from any configuration, whatever the computation performs, a deadlock configuration is reached. In the section 5, we will prove that the reached deadlock configuration belongs to $\mathcal{LS}$: one and only one processor is elected. We will also proof that the elected processor has the smallest id-value of the ring.

4 Convergence

Definition 5.
• $\mathcal{REG}_0 = \{\forall p \in \mathcal{SI}, F_p[k+1] = id_p\}$. Let p be a processor of $\mathcal{SI}$. Thus, in $\mathcal{REG}_0$, $F_p[k+1]$ is the id-value of p.

[1] this action is similar to an action on a FIFO list where the processors gets the first value of the list and adds it id-value at the end of the list.

• $i \in [0, k+1]$, $\mathcal{REG}'_i = \mathcal{REG}_i \cap \{\forall p \in \mathcal{BI}, F_p[k+1-i]$ is the id-value of the i+1-th previous processor of p in the ring that belongs to $\mathcal{SI}\}$.
• $i \in [1, k+1]$, $\mathcal{REG}_i = \mathcal{REG}'_{i-1} \cap \{\forall p \in \mathcal{SI}, F_p[k+1-i]$ is the id-value of the i-th previous processor of p in the ring that belongs to $\mathcal{SI}\}$.

In the ring of the figure 1, $\mathcal{SI}$ contains 3 processors (processors whose id-value are $0, 1, or\ 3$). Call p the processor whose id-value is 0. 3 is the id-value of the first previous processor of p that belongs to $\mathcal{SI}$; 1 is the id-value of the second one; 0 is the id-value of the third one; 3 is the id-value of the forth one. Notice that third previous processor of p in $\mathcal{SI}$ is p itself.

The figure 2 illustrates how a ring stabilizes. From any configuration, whatever the computation performs $\mathcal{REG}_0$ is reached in at most one round. After that, $\mathcal{REG}'_0$ is reached in at most $N-1$ rounds; then $\mathcal{REG}_1$ in one round, $\mathcal{REG}'_1$ in at most $N-1$ rounds ... . We will prove that $\mathcal{REG}_0$ is an attractor. Then, we will finish the convergence verification by proving that (i) $\mathcal{REG}'_i$ is an attractor if $\mathcal{REG}_i$ is an attractor; (ii) $\mathcal{REG}_{i+1}$ is an attractor if $\mathcal{REG}'_i$ is an attractor; (iii) from any configuration of $\mathcal{REG}_i$, $\mathcal{REG}'_i$ is reached in at most $N-1$ rounds; (iv) from any configuration of $\mathcal{REG}'_i$, $\mathcal{REG}_{i+1}$ is reached in 1 round. Thus, we will prove that $\mathcal{REG}'_{k+1}$ is an attractor that is reached in at most $(k+1)N$.

Lemma 6. *Let p be a processor of $\mathcal{SI}$. Whatever the computation performs, $F_p[k+1]$ will eventually get the value id_p. Then, $F_p[k+1]$ will keep this value forever. Whatever the current configuration, after a round, each processor p of $\mathcal{SI}$ verify the equality: $F_p[k+1] = id_p$.*

Proof. **Closure** After an action, p verifies the *Following* predicate according to the definition of guard actions that p may perform ($\mathcal{B}_1$, $\mathcal{B}_2$, or $\mathcal{B}_3$). thus no action may change the value of $F_p[k+1]$, once $F_p[k+1] = id_p$.
Convergence if $F_p[k+1] \neq id_p$, then p holds the $\mathcal{B}_1$ guard till $F_p[k+1] \neq id_p$. Thus, p will eventually perform this action.
Stabilization time all processors of $\mathcal{SI}$ such that $F_p[k+1] \neq id_p$ hold the $\mathcal{B}_1$ guard. During the first round, they perform this action; thus after the round all processors of $\mathcal{SI}$ verify the equality $F_p[k+1] = id_p$. □

The following lemma is a direct consequence of lemma 6 and its proof.

Lemma 7. *$\mathcal{REG}_0$ is attractor for* true. *Whatever the initial configuration, $\mathcal{REG}_0$ is reached in one round.*

Definition 8. Let p be a processor of $\mathcal{BI}$. Let us call p' the first previous processor of p belonging to $\mathcal{SI}$. p verifies the predicate $stable_i$ if only if (i) $F_p[k+1-i] = F_{p'}[k+1-i]$; and if (ii) lp[2] verifies $stable_i$ predicate. or lp belonging to $\mathcal{SI}$.

Remark. Let c be a configuration of $\mathcal{REG}_i$. c is a configuration of $\mathcal{REG}'_i$ if and only if all processors of $\mathcal{BI}$ verify the $stable_i$ predicate.

[2] lp is the left neighbor of p.

Lemma 9. *If $\mathcal{REG}_i$ is closed then $stable_i$ predicate is a trap in $\mathcal{REG}_i$.*

Proof. Let p be a processor of $\mathcal{BI}$ that verifies the $stable_i$ predicate in $\mathcal{REG}_i$. Call p' the first previous processor of p belonging to $\mathcal{SI}$. Let $\mu_{p'p}$ be the path from p' to p. If p verifies the $stable_i$ predicate then each processor in the path $\mu_{p'p}$ verifies the $stable_i$ predicate. Assume that p stops to verify the $stable_i$ predicate. Let us call ps the first processor in the path $\mu_{p'p}$ that has stopped to verify the $stable_i$ predicate (such a processor exists by assumption). In order to stop to verify the $stable_i$ predicate in $\mathcal{REG}_i$, ps has to change its $F_{ps}[k+1-i]$ value although $F_{ps}[k+1-i]$ was equal to $F_{lps}[k+1-i]$ value and lps[3] has not modified the $F_{lps}[k+1-i]$ value. Such a change cannot be done. □

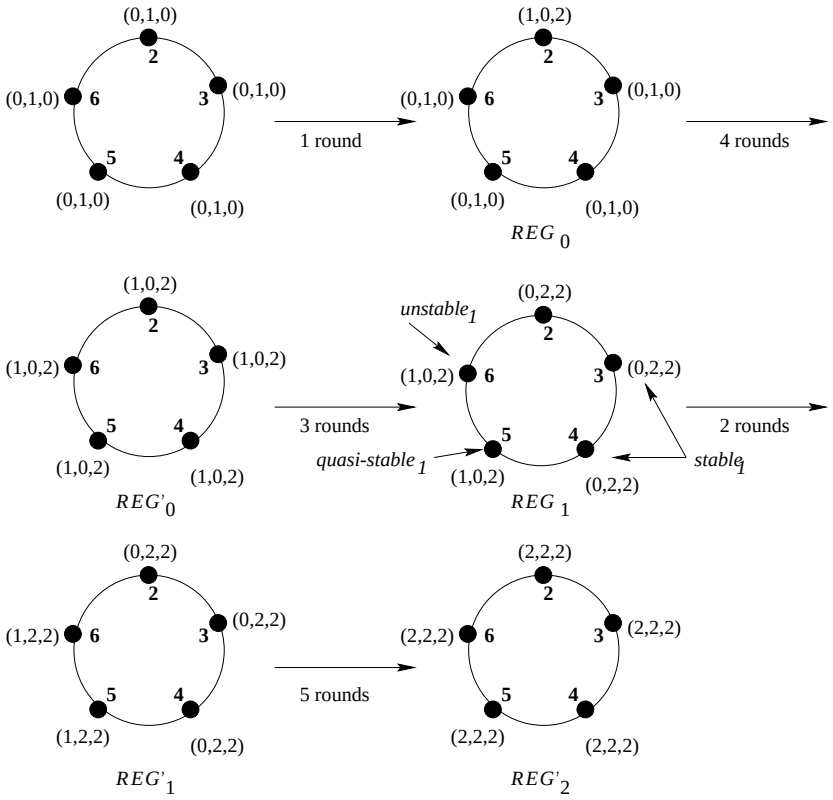

Fig. 2. stabilization in a 5-Ring that is 1-bounded

[3] lps is the left neighbor of ps.

Definition 10. Let p be a processor of $\mathcal{BI}$. p verifies the predicate $quasi\_stable_i$ if only if (i) p does not verify the $stable_i$ predicate, and (ii) lp verifies the $stable_i$ predicate or $lp \in \mathcal{SI}$.

Lemma 11. *If it exists a processor p of $\mathcal{BI}$ that does not verify the $stable_i$, then there is a processor ps that verifies the $quasi\_stable_i$ predicate.*

Proof. Call p' the first previous processor of p belonging to $\mathcal{SI}$. Let $\mu_{p'p}$ be the path from p' to p. Call p_j the first processus on the path $\mu_{p'p}$ that p_j does not verified the $stable_i$ predicate. If $j = 1$ then p_1 does not satisfy the $stable_i$ predicate and its left neighbor belong to $\mathcal{SI}$; thus p_1 verifies the $quasi\_stable_i$ predicate. If $j > 1$, p_j does not satisfy the $stable_i$ predicate but its left neighbor satisfies this predicate. In both cases, p_j verifies the $quasi\_stable_i$ predicate. □

Lemma 12. *$\forall i \in [0, k+1]$, if $\mathcal{REG}_i$ is closed then $\mathcal{REG}'_i$ is an attractor for $\mathcal{REG}_i$ From any configuration of $\mathcal{REG}_i$, $\mathcal{REG}'_i$ is reached in at most $N-1$ rounds.*

Proof. Let us call nbr the number of processors of $\mathcal{BI}$ that does not verify the $stable_i$ predicate.
Closure nbr cannot increase in $\mathcal{REG}_i$ (lemma 9)
Convergence if $\mathcal{REG}'_i$ is not reached, then there is a processor p of $\mathcal{BI}$ that does not verify the $stable_i$ predicate (remark 4). Thus, there is a processor ps that does verify the $quasi\_stable_i$ predicate (lemma 11). Let us call ps' the first previous processor of ps that belongs to $\mathcal{SI}$. $F_{ps}[k+1-i] \neq F_{ps'}[k+1-i]$ (ps does not verify the $stable_i$ predicate) and $F_{lps}[k+1-i] = F_{ps'}[k+1-i]$ (lps verifies the $stable_i$ predicate or belongs to $\mathcal{SI}$). Thus, ps holds the $\mathcal{A}_1$ guard ($F_{ps}[k+1-i] \neq F_{lps}[k+1-i]$). As $stable_i$ predicate is a trap in $\mathcal{REG}_i$ (lemma 9); ps verifies the $quasi\_stable_i$ predicate till it has not executed $\mathcal{A}_1$ action. And, ps holds the $\mathcal{A}_1$ guard till it verifies the $quasi\_stable_i$ predicate. By fairness scheduling, ps will eventually execute $\mathcal{A}_1$ action. After this action, ps verifies the $stable_i$ predicate ($F_{ps}[k+1-i] = F_{lps}[k+1-i] = F_{ps'}[k+1-i]$). nbr has decreased. When $nbr = 0$, $\mathcal{REG}'_i$ is reached (remark 4).
Stabilization time according to the round definition, at each round nbr decreases. As nbr is bounded by $N-1$ ($\mathcal{BI}$ has at most $N-1$ processors); after at most $N-1$ rounds, $\mathcal{REG}'_i$ is reached. □

Lemma 13. *$\forall i \in [1, k+1]$, if $\mathcal{REG}'_{i-1}$ is closed then $\mathcal{REG}_i$ is attractor for $\mathcal{REG}'_{i-1}$. Whatever the current configuration of $\mathcal{REG}'_{i-1}$, $\mathcal{REG}_i$ is reached in one round.*

Proof. Let p be a processor belonging to $\mathcal{SI}$. Call p' the first previous processor of p belonging to $\mathcal{SI}$.
Let (1) be the equation defined as follows: $F_{lp}[k+2-i] = F_{p'}[k+2-i]$
If lp belongs to $\mathcal{SI}$, (1) is always verified. If lp belongs to $\mathcal{BI}$, $F_{lp}[k+2-i]$ is the id-value of the (i)-th previous processor of lp; and also the id-value of $i-1$-th processor previous of p'. Therefore, the equation (1) is verified in $\mathcal{REG}'_{i-1}$.

In $\mathcal{REG}'_{i-1}$, $F_{p'}[k+2-i]$ is the id-value of the (i)-th previous processor of p. Thus $\mathcal{REG}_i$ is not reached if only if there is a processor p of $\mathcal{SI}$ verifying the following condition: (2) $F_p[k+1-i] \neq F_{lp}[k+2-i]$.
Let us call nbr the number of processors verifying (2).
Closure nbr cannot increase in $\mathcal{REG}'_{i-1}$, because lp cannot change the value of $F_{lp}[k+2-i]$ ($\mathcal{REG}'_{i-1}$ is closed, see the hypothesis) and once the following equality is verified $F_p[k+1-i] = F_{lp}[k+2-i]$, it stays verify forever (definition of guard and statement actions).
Convergence a processor p verifying (2) holds the $\mathcal{B}_1$ guard. By fairness, p will perform the action $\mathcal{B}_1$. After that action: $F_p[k+1-i] = F_{p'}[k+2-i]$; nbr has decreased.
Stabilization time all processors verifying (2) hold the $\mathcal{B}_1$ guard; during the first round in $\mathcal{REG}'_{i-1}$, all of them perform this action. After one round, no processor verifies (2): $\mathcal{REG}_i$ is reached. □

Theorem 14. *$\forall i \in [0, k+1]$, $\mathcal{REG}_i$ is an attractor for* true *and $\mathcal{REG}'_i$ is an attractor for* true.

Proof. $\mathcal{REG}_0$ is attractor for true (lemma 7). $\forall i \in [0, k+1]$, if $\mathcal{REG}_i$ is closed then $\mathcal{REG}'_i$ is attractor for $\mathcal{REG}_i$ (lemma 12). $\forall i \in]0, k+1]$, if $\mathcal{REG}'_{i-1}$ is closed then $\mathcal{REG}_i$ is attractor for $\mathcal{REG}'_{i-1}$ (lemma 13). According to the theorem 3, $\mathcal{REG}'_0$, $\mathcal{REG}_1$, $\mathcal{REG}'_1$, $\mathcal{REG}_2$, $\mathcal{REG}'_2$, ... $\mathcal{REG}_{k+1}$, $\mathcal{REG}'_{k+1}$ are the attractors for true. □

Lemma 15. *From $\mathcal{REG}'_{k+1}$, any computation will eventually reach a deadlock configuration. The deadlock configuration is reached in one round.*

Proof. In $\mathcal{REG}'_{k+1}$, the array F of a processor never change its value: the action $\mathcal{A}_1$ and $\mathcal{B}_1$ cannot be performed. A processor can only update its Ld variable according to its F variable: The action that updates its variable ($\mathcal{A}_2$, $\mathcal{B}_2$, or $\mathcal{B}_3$) will be performed at most one time.

In a no-deadlock configuration of $\mathcal{REG}'_{k+1}$, processors that need to update their Ld variable verify an action guard. After a round, no processor need to update their Ld variable: a deadlock configuration is reached. □

5 Correctness

In the previous section, we have proven that whatever the computation performs, a deadlock configuration is reached that belong to $\mathcal{REG}'_{k+1}$. In this section, we will prove that in a deadlock configuration of $\mathcal{REG}'_{k+1}$, one only one processor is elected: the processor having the smallest id-value.

Remark. Let c be a deadlock configuration of $\mathcal{REG}'_{k+1}$. In c, the array F of any processor of $\mathcal{SI}$ contains only id-values of processors in the ring (processors that belongs to $\mathcal{SI}$). The processor having the smallest id-value in the ring (processor that belongs to $\mathcal{SI}$) is a leader in c.

Lemma 16. *Let c be a deadlock configuration of ${\mathcal{REG}'}_{k+1}$. In c, the array F of any processor of $\mathcal{SI}$ contains all id-values of processors in the ring that belongs to $\mathcal{SI}$.*

Proof. Let $\mathcal{R}$ be a k-bounded ring. Let p be a processor of $\mathcal{SI}$. Call S the size of $\mathcal{SI}$. $S \leq k+2$ ($\mathcal{SI}$ contains at most $k+2$ processors).
$(F_p[k+1], ...F_p[k+2-S])$ is the list of id-values of S previous processors of $\mathcal{SI}$ in the ring.Thus, F_p contains all id-values of processors in the ring that belongs to $\mathcal{SI}$. □

Theorem 17. *Let c be a deadlock configuration of ${\mathcal{REG}'}_{k+1}$. $c \in \mathcal{LS}$.*

Proof. In c, there is at least one leader (see remark 5).
In c, the array F of any processor of $\mathcal{SI}$ contains all id-values of processors in the ring that belongs to $\mathcal{SI}$ (lemma 16). Only the processor of $\mathcal{SI}$ having the smallest id-value is a leader. □

6 Conclusion

Theorem 18. *The algorithm 3 requires only constant register space.*

Proof. In the algorithm 3, the field variables of a processor p are Ld_p and F_p. Thus, the register space required by the algorithm 3 is $2.(k+2)^{k+2}$ states. □

We have proven that the algorithm 3 is a a self-stabilizing leader election algorithm on unidirectional k-bounded rings. The algorithm 3 is a silent one: the legitimate configuration is deadlock one (notice that there is only one legitimate configuration on a ring).

Theorem 19. *Whatever is the initial configuration, the system stabilizes in at most $(k+2)N+1$ rounds.*

Proof. Whatever the initial configuration, $\mathcal{REG}_0$ is reached in one round (lemma 7). From any configuration of $\mathcal{REG}_i$, ${\mathcal{REG}'}_i$ is reached in at most $N-1$ rounds (lemma 12). Whatever the current configuration of ${\mathcal{REG}'}_{i-1}$, $\mathcal{REG}_i$ is reached in one round (lemma 13).

According to lemma 7 and 12, ${\mathcal{REG}'}_0$ is reached in at most N rounds. Whatever the current configuration of ${\mathcal{REG}'}_{i-1}$, ${\mathcal{REG}'}_i$ is reached in N rounds (lemma 13 and 12). Thus ${\mathcal{REG}'}_{k+1}$ is reached in at most $(k+2)N$ rounds. Whatever the current configuration of ${\mathcal{REG}'}_{k+1}$, the deadlock configuration is reached in at most one round. According to the theorem 17, this configuration is legitimate. □

References

[AB97] Y Afek and A Bremler. Self-stabilizing unidirectional network algorithms by power-supply. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA97)*, pages 111–120, 1997.

[AG93] A Arora and MG Gouda. Distributed reset. In *FSTTCS90 Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag LNCS:472*, pages 316–329, 1993.

[AG94] A Arora and MG Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[AKY90] Y Afek, S Kutten, and M Yung. Memory-efficient self-stabilization on general networks. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486*, pages 15–28, 1990.

[BGM89] JE Burns, MG Gouda, and RE Miller. On relaxing interleaving assumptions. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.

[BJ98] J. Beauquier and C. Johnen. Deterministic and self-stabilizing leader election protocol. Technical report, Laboratoire de Recherche en Informatique, Université Paris-Sud, 1998.

[DGS96] S Dolev, MG Gouda, and M Schneider. Memory requirements for silent stabilization. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

[Dij74] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[Dol93] S Dolev. Optimal time self-stabilization in dynamic systems. In *WDAG93 Distributed Algorithms 7th International Workshop Proceedings, Springer-Verlag LNCS:725*, pages 160–173, 1993.

[GG96] S Ghosh and A Gupta. An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, 59:281–288, 1996.

[GH96] MG Gouda and FF Haddix. The stabilizing token ring in three bits. *Journal of Parallel and Distributed Computing*, 35:43–48, 1996.

[GM91] MG Gouda and N Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[ILS95] G Itkis, C Lin, and J Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972*, pages 288–302, 1995.

[JABD97] C Johnen, G Alari, J Beauquier, and AK Datta. Self-stabilizing depth-first token passing on rooted networks. In *WDAG97 Distributed Algorithms 11th International Workshop Proceedings, Springer-Verlag LNCS:1320*, pages 260–274, 1997.

[Joh97] C Johnen. Memory-efficient self-stabilizing algorithm to construct BFS spanning trees. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 125–140. Carleton University Press, 1997.

[MOY96] A Mayer, R Ostrovsky, and M Yung. Self-stabilizing algorithms for synchronous unidirectional rings. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA96)*, pages 564–573, 1996.

[Pet97] F. Petit. Highly space efficient self-stabilizing depth-first token circulation for trees. In V. Villain A. Bui, M. Bui, editor, *On Principles Of DIstributed Systems, OPODIS'97*, pages 221–236. Hermes, 1997.

[PV97] F Petit and V Villain. Color optimal self-stabilizing depth-first token circulation. In *I-SPAN'97, Third International Symposium on Parallel Architectures, Algorithms and Networks Proceedings, IEEE Computer Society Press*. IEEE Computer Society Press, 1997. To appear.

[Vil97] V. Villain. A new lower bound for self-stabilizing mutual exclusion algorithms. Technical Report RR97-17, LaRIA, University of Picardie Jules Verne, France, 1997.

This article was processed using the LaTeX macro package with LLNCS style