Randomized Uniform Self-stabilizing Mutual Exclusion

Jérôme O. DURAND-LOSE *†

Laboratoire I3S, CNRS UPREES-A 6070, 930 Route des Colles, BP 145,
06903 SOPHIA ANTIPOLIS Cedex, FRANCE.

Abstract. *The mutual exclusion protocol presented ensures that whatever perturbation the network undergo, it regains consistency in finite time: one and only one privilege token is present. Many such protocols are know for rings, ours is the first one for anonymous network with any topology in the shared memory communication model.*

It carries out the token random walk scheme presented by Israeli and Jalfon in PODC '90 and extends a ring protocol presented by Beauquier and Delaët in PODC '94. It uses $O(\ln n)$ states per register and the expected transient time is polynomial.

Key-words. *Self-stabilization, Mutual exclusion, Random walks.*

1 Introduction

A system of interconnected processors often needs some mutual exclusion (ME) scheme for processors to execute some critical part of their code. At any time one and only one processor is privileged and may enter a critical section – in order, *e.g.*, to write a data or access a device. Each processors should be privileged infinitely often. We use the abstract concept of *token* to indicate the privilege.

Sometimes systems get corrupted, *e.g.*, no privileged processor exists –access to crucial resources is lost– or more that one is privileged –data might get corrupted, devices confused by dual access ... This can happen for various reasons: processor/communication failure/addition/removal.

A system is said *self-stabilizing* (SS) if when started in any possible configuration it eventually reaches a legal configuration (*convergence*) and once

*jdurand@unice.fr, http://www.i3s.unice.fr/~jdurand.
†This work was done while the author was in LaBRI, Université Bordeaux I, FRANCE.

in a legal configuration, all following configurations are legal (*correctness* or *closure*).

A self-stabilizing system does not need to be initialized. Moreover, it is tolerant to fault: it regains consistency without any external intervention when a processor crashes and recovers in any arbitrary state. The time period between crashed is supposed to be long enough for stabilization and computation.

It this article we provide an algorithm that ensures that the system eventually enters a legal configuration –one and only one privileged processor– and that the system remains in legal configurations afterwards. Our *uniform self-stabilizing mutual exclusion* (USSME) is randomized.

The concept of self-stabilization was first introduced in the pioneering paper by Dijkstra [Dij74] which provides SSME for rings. He mentions that no deterministic protocol exists for uniform ring with a composite number of vertices. Angluin [Ang80] proves that it is impossible to deterministically distinguished a vertex in a graph that is a strict covering of another graph. There is no way to deterministically break existing symmetries in any uniform distributed system. Thus, no deterministic USSME exists for graph with symmetries and randomized approach has to be used.

Herman [Her90] provides a parallel and randomized USSME for rings of odd size. Each processor holds a bit. The privilege token corresponds to having the same value as the preceding processor. Tokens make random walk on the ring; disappearing on meeting.

Israeli and Jalfon [IJ90] define a SSME construction based on two levels of abstraction: token manipulation (lower level) and graph traversal. They prove lower bounds on the number of states for USSME on rings and general graphs. Our USSME uses a number of states of the order of their bounds as long as the degree is bounded by some constant.

Beauquier and Delaët [BD94] provide a random USSME on rings. Their technique is to impose a gap between the integer values of neighbor vertices, the gap and modulo base are such that there should be an irregular gap somewhere in the ring. This irregular gap yields the privilege. The correction of their algorithm comes from the non-increasing number of irregular gaps. This provides a generalization of SSME on rings based on alternating 0 and 1. Their gap depends on the number of vertices and a global orientation is needed. Orientation of ring can be constructed by SS algorithms. Israeli and Jalfon [IJ93] provide such an algorithm, as does Beauquier *et al.* [BDK96] (and one for torus [BDKR98]).

In a distributed system, there is no global control, processors work on their own with only the information provided by neighbor processors. Each processor is a randomized finite state machine. The processor network is uniform, this means that processors are anonymous and that any two processors with the same degree are identical.

Communications are made through registers. One register is attached to each end of any communication edge. Along any edge, any incident processor can read the register on both side but it can only write on the register on its

side. The communications are safe.

To ensure correctness of our USSME we consider that processors are activated by some adversary *deamon* scheduler. The deamon knows the actual configuration but ignores the result of the next coin toss. It can activate any set of vertices at a time. The deamon is fair: it must activate infinitely often each processor.

We consider *central, distributed* and *read/write deamons*. With the last one, activated processors can only perform one atomic action during which only one (communication) register can be accessed (for either reading or writing).

The cited algorithms work on rings ([Dij74, BD94]) or any graphs ([IJ90]), directed or oriented or not, for some deamon or only in the synchronous case ([Her90]), are deterministic ([Dij74, BD94]) or randomized ([Her90, DIM90]) ... Some work on any graph but with a lesser version of uniformity: one or two processors may be different ([Dij74, DIM90]).

Our USSME works in almost all of the above cases. It follows the idea of Beauquier and Delaët: identify irregular integer gaps with tokens. We use random walks to ensure stabilization and fair sharing of the token afterwards in any graph. Each register holds an integer between 0 and $m-1$, where m does not divide the number of vertices n ($n=|V|$). We say that a processor is *balanced* if it verifies a modified Kirshoff's law: the sum of inside registers is equal to the sum of outside registers, plus 1 (modulo m). The number of processors ensures that not all processors can be balanced simultaneously.

An unbalanced processor first tosses a coin to decide whether he passes the token, this forbids deadlocks in the synchronous case. To pass it, he recovers balance by adding its *bias* to one of its registers randomly chosen. The corresponding processor gains the bias which is added to its own if any. The bias represents a privilege token which is transmitted to some randomly chosen neighboring processor.

Tokens make random walks in the graph and merge or disappear on meeting. Using random walk techniques, we show that any two tokens meet in finite time with probability one. Eventually, only one token remains.

Let a *round* be any minimal activation sequence such that all the processors are activated at least one time. The expected stabilization time, in rounds, is polynomial. It is bounded by $O(mn^2)$ in the case of a central random activation and $O(mn^3)$ in the read/write case.

The USSME and the privilege condition are presented in Sect. 3. We prove the correction and the convergence of our algorithm In Sect. 4 and give a polynomial bound on the expected stabilization time in Sect. 5.

2 Definitions

The processor network is modeled by its (finite and connected) *communication graph* $G = (V, E)$. Let n be the number of vertices in the graph ($n=|V|$), Δ be its diameter and D be its maximal degree. For any vertex x, we denote E_x the

collection of edges incident to x and d_x its degree. If (x, y) is an edge, then x and y are *neighbors*.

Communications are handled with registers. For each edge (x, y) there are one register attached to each incident vertex: R_{xy} and R_{yx}. Vertices x and y can read both registers. Only x (y) can modify R_{xy} (R_{yx}).

We suppose the network *uniform*: processors are anonymous and any two processors with the same degree are identical. Each processor is defined by its degree.

Let $\mathcal{C}$ be the set of all configurations. We denote that a configuration c_2 can be reached from configuration c_1 by a single activation of processors by $c_1 \vdash c_2$. The transitive closure of $\vdash$ is denoted $\vdash^*$.

A protocol is called *self-stabilizing* (SS) for a given set of *legal configurations* $\mathcal{L}$ ($\mathcal{L} \subset \mathcal{C}$) if it verifies the following assertions:
- *correctness*: once in a legal configuration c ($c \in \mathcal{L}$) the system remains in legal configurations (if $c_1 \in \mathcal{L}$ then for any c_2 such that $c_1 \vdash^* c_2$, $c_2 \in \mathcal{L}$);
- *convergence*: the system eventually enters a legal configuration with probability 1.

No deadlock should come from SS: any deadlock configuration must be a legal one.

A *mutual exclusion* (ME) is an algorithm/protocol/scheme such that in any configuration, one and only one processor is in a *privileged* state, all the other processors are in unprivileged states. In any execution of a ME, all processors are infinitely often privileged. The privilege is represented by some abstract *token* which is passed on.

When legal configurations are defined by containing exactly one token, one speaks of a *self-stabilizing mutual exclusion* (SSME).

Generally the termination, or the fact that a legal configuration is reached, is not locally detectable. This comes from the fact that the system can be started in any configuration, including ones that are locally correct but globally illegal, *e.g.*, two privilege tokens at sufficiently large distance.

The updates of the processors are done according to their activation by a scheduler. The scheduler chooses each time which processors are activated.

To ensure correctness, the classical model for scheduler is an adversary *deamon*. The deamon knows the whole configuration but ignores the result of the next coin toss. It is *fair*: any processor is activated infinitely often in any infinite activation sequence.

We consider the following kinds of deamons:
- *central deamon*: it can only activate one processor at a time;
- *distributed deamon*: it can activate any set of processors at a time;
- *read/write deamon*: it can activate any set of processors, but processors only performs one atomic action that contains at most one access (reading or writing) to one (communication) register.

The central deamon is the weakest adversary. The read/write deamon is the strongest one because it may have all the processor read registers and then have some modify registers so that many processors works with outdated data.

3 USSME definition

Definition 1 Let m be the smallest integer such that m does not divide the number of vertices n.

Each edge register can hold any value between 0 and $m-1$. A configuration is defined by the values of all the registers.

Definition 2 A vertex x is *balanced* if it verifies the following equation:

$$\sum_{(x,y)\in E_x} R_{xy} \equiv \sum_{(x,y)\in E_x} R_{yx} + 1 \mod m \ .$$

It is a modified Kirshoff's law: the outgoing (inside) is the incoming (outside) plus 1 (modulo m).

Definition 3 A processor is *privileged* when it is unbalanced. The difference to balance is called the *bias*.

When an unbalanced processor is activated, if a coin toss succeeds, it tries to recover balance by adding the computed bias to a randomly chosen register. The algorithm is given in Fig. 1.

```
1   diff := 0 ;
2   for_each (x, y) in E_x
3       diff := ( diff + R_xy ) mod m ;
4       diff := ( diff - R_yx ) mod m ;
5   if ( diff == 1 ) then                /* unbalanced */
6       /* begin critical section */
7       ...
8       /* end critical section */
9       if toss_coin_with_probability ( degree / (degree + 1) ) then
10          /* pass the privilege */
11          (x, y) := randomly_chosen_incident_edge () ;
12          diff := ( R_xy + 1 - diff ) mod m ;
13          R_xy := diff ;       /* regain balance */
14      end
15  end
```

Figure 1: Balancing algorithm for vertex x.

The algorithm is written in detail in order to show the atomic activation parts of the read/write deamon. The atomic actions are the reading of all

registers (inside and outside, *i.e.*, lines 3 and 4) then the eventual passing of the privilege (lines 12 and 13).

Let us note that no copy of the contents of the registers are inside the processor. This avoid duplication of data and the risk of outdated values. If there would have been copies, the processor should read anyway its registers because they can mask the presence of bias and produce a deadlock.

The random test to let go the privilege (line 9) prevents malice actions of distributed and read/write deamons as in [DIM90]. If the toss coin fails, the processor has to read again all the registers. If the value of the held data $diff$ is wrong, on the second reading, the other bias is discovered and both are added (leading even to their destruction).

In the next section, we prove that the algorithm ensures that the system eventually reaches a configuration where one and only one processor is unbalanced and afterward, one and only one processor is unbalanced.

An example of stabilizing iterations is given in Fig. 2. We only consider the case where activated processors are unbalanced and pass the privilege. The privileged processors are indicated by '$*$' and the activated processors by circles. Only two states are needed on each register since the number of vertices is odd. Initially, the number of irregular gap is 3, it rapidly goes down to 1. Finally, the irregular gap makes a random walk inside the graph.

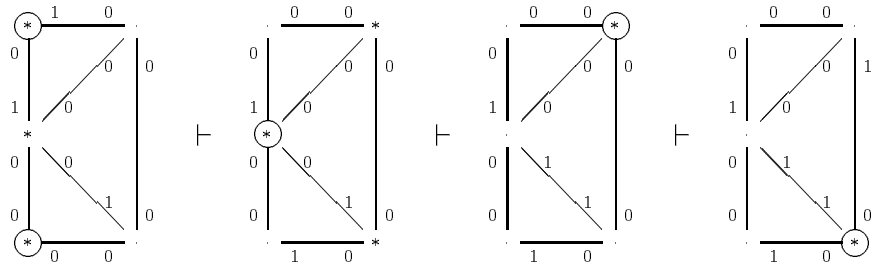

Figure 2: Example of iterations, $n = 5$ and $m = 2$.

4 Correction of the algorithm

In this section, we prove that our algorithm is a USSME against any deamon. The proofs for read/write deamon are only sketched. The atomic action are:

- all the readings to test for balancing and to compute the bias;
- reading the value of the register to be modified;
- writing a new value (with the known information) on the chosen register.

Lemma 4 *There is always at least one unbalanced processor.*

Proof. By contradiction, let's assume that all the processors are balanced. Then the following formula is verified:

$$\forall x \in V, \qquad \sum_{(x,y)\in E_x} R_{xy} \equiv \sum_{(x,y)\in E_x} R_{yx} + 1 \mod m \ .$$

Let us sum it over all vertices:

$$\sum_{(x,y)\in E} R_{xy} \equiv \sum_{(x,y)\in E} R_{yx} + n \mod m \ .$$

Is simplifies to $n \equiv 0 \mod m$ which contradicts Definition 1 (m does not divide n).

When considering the read/write deamon, unbalance can be temporally hidden. If no processor finds itself unbalanced because of outdated reading then no register can be modified. Since the deamon is fair, all processors will read again the registers and then any unbalanced one realizes that it has a token.

□

There is always at least one privileged processor. This ensures that no deadlock is possible.

Lemma 5 *The number of unbalanced processors is non-increasing.*

Proof. Let us look what happens when a vertex is updated:
– if the vertex is balanced then no register is modified, the number of unbalanced processors remains as is;
– if the vertex is not balanced then the algorithm eventually balances it – the number of unbalanced processors is diminished by one– and the register of one edge is changed. This change can only unbalanced one processor: the one at the other end of the edge. If it was unbalanced, then the number of unbalanced processors is diminished by one –or two if it balances the end processor– otherwise it remains constant;
– if more than processor are activated synchronously. Let us only consider the unbalanced ones. Each one can unbalance at most one processor. If it remains unbalanced, it means that it was unbalanced by another one, but the other one does not act on any other processor.

□

For read/write deamon, we have to consider:
– processors that "perceived themselves unbalanced" once they have read a register that has changed since last read (even if they have no memory of previous value and have not yet computed the total bias);
– "delayed bias" sent by a processor but the corresponding register has already been read. The delayed bias is considered in the next cycle of the processor.

There are less token than perceived themselves unbalanced processors and delayed bias. It is possible to prove that this sum is non increasing.

From the two previous lemmas, it comes:

Lemma 6 *Once there is only one unbalanced processor, there remains only one forever.*

Finally we have to ensure that a legal state is eventually reached.

Lemma 7 *The system eventually reaches a configuration where only one processor is unbalanced with probability 1.*

Proof. There must be at least one token in the graph, let us consider that there are more than one. Let us consider any two tokens in the graph.

The dynamic given in the proof of Lem. 5 is the dynamic of random walk of bias in the graph. When two biases met, they merge (or disappear which accelerate the diminishing process and is thus not considered).

Let d be their distance and D the maximal degree of the graph. The first time that one of their processors is activated (the last part of the algorithm):
– if the other processor is not activated: the probability that the distance between bias decreases by one is down bounded by $1/(D+1) \leq \min(1/(d_x+1), 1/(d_y+1))$ (it goes towards the other);
– if both are activated: this probability is down bounded by $1/(D+1) \leq (d_x/(d_x+1)+d_y/(d_y+1)).1/(D+1)$ (one goes toward the other and the other remains in the same vertex). The case where they both go one toward the other is not counted because they pass each other at the distance one.

The probability that the distance decreases is down bounded by $4/(D+2)^2$. Once the distance reach 0, they merge or both disappear, the number of token decreases. Since the deamon is fair, the processors holding the two tokens have to be activated. They eventually merge or disappear with probability 1.

There are at most n tokens and while there are at least two, their number eventually decreases with probability one.

□

Let a round be any minimal sequence of activations where each processor is activated at least one. Above computation gives a lower bound of the probability that a token disappear in Δ rounds. Let $S_t = r$ mean that the systems, starting with t tokens is stable after r rounds, *i.e.*, there is only one token left. Since two tokens are at most at distance Δ (the diameter of the graph):

$$\left(\frac{1}{(D+1)}\right)^{\Delta} \leq \mathcal{P}\left(S_2 = \Delta\right) \quad .$$

For the read/write deamon, one must only consider activations which corresponds to lines 9 to 14 of Fig. 1, *i.e.*, the reading part is not considered. This corresponds to a factor of $O(D)$.

5 Expected stabilization time

Let us consider the central assumption –only one processor activated at a time– and that processors are activated with a probability proportional to their degrees plus one to simplify calculus.

Theorem 8 *Under the central deamon assumption, the expected number of rounds for stabilization is up bounded by* $12mn^2\Delta$.

Proof. We consider the expected time for two tokens to meet. If one encounters another one before, either they merge and we consider the resulting token or they disappear which terminate directly as only one token remains.

Let us consider the graph $\tilde{G}$ build as follows: $\tilde{V} = V \times V$ and for any x, y, t and u in V, the following edges are in $\tilde{E}$: $([x,t],[x,t])$; $([x,t],[y,t])$ if $(x,y) \in E$; and $([x,t],[x,u])$ if $(t,u) \in E$. Any position in $\tilde{G}$ represents a position of two tokens. When a token is activated, any kind of edge can be crossed with the same probability (from the probabilities of processor activation and edge selection in our algorithm). By counting double the loop edge, we get a random walk in $\tilde{G}$.

Using Lemma 3 from [AKL+79], we get an upper bound on the expected number of activation before the two tokens go from any vertex to some $[x,x]$ vertex: $T \leq 2\tilde{m}\tilde{\Delta}$ where $\tilde{m} = n+2mn$ is the number of edges of $\tilde{G}$ and $\tilde{\Delta} = 2\Delta$ is the diameter of $\tilde{G}$.

Since at maximum $n-1$ token encounters are needed to reduce to one token, the expected number of rounds for stabilization is bounded by: $(n-1)2\tilde{m}\tilde{\Delta} = (n-1)2(n+2mn)(2\Delta) \leq 12mn^2\Delta$.

□

It is also possible to prove polynomial upper bounds in $O(mn^3)$ for distributed and read/write deamon with similar techniques and probabilities of activation.

6 Conclusion

The algorithm works for any graph. One stabilization is achieved, random walks ensure a fair sharing of the privilege token.

The lowest integer which does not divide the number of vertices n is up bounded by $O(\log(n))$ because $e^{1.8k} < lcm(1,2,..k)$ [RS62] (as mentioned in [IJ90]). The number of states needed to implement the algorithm is bounded by $O(D\log(n))$. There are at most $n-1$ incident edges, a more general upper bounded is $O(n\log(n))$.

To bound the expected stabilization time, we consider the time to arrive to some particular vertex and not in the diagonal set of $\tilde{G}$ and that the first token merge with the second, then the third, ... then the nth. The possibility that they merge before and in any order is omitted. We believe that all our bounds can be lowered by a power n or $\log(n)$.

The algorithm is partially dynamic: if processors/communication links are added or deleted, new bias are generated. As long as m does not divide n, the system regains consistency. For any N, it is sufficient to take $m = N+1$ to make the algorithm work on any network with at most N processors.

References

[AKL+79] R. Aleliunas, R. M. Karp, R. Lipton, L. Lovász, and C. Rackoff. Random walks, univeral traversal sequences and the complexity of the maze problem. In *Proc. Symp. on Foundation of Computer Science (FOCS'79)*, pages 218–223, 1979.

[Ang80] D. Angluin. Local and global properties in networks of processors. In *Proc. Symp. on Theory of Computing (STOC'80)*, pages 82–93, 1980.

[BD94] J. Beauquier and S. Delaët. Probabilistic self-stabilizing mutual exclusion in uniform rings. In *Principles of Distributed Computing (PODC'94)*, page 378, 1994.

[BDK96] J. Beauquier, O. Debas, and S. Kekkonen. Fault-tolerant and self-stabilizing ring orientation. In 3^{st} *International Colloquium on Structural Information & Communication Complexity (SIROCCO)*. Carleton University Press, 1996.

[BDKR98] J. Beauquier, O. Debas, S. Kekkonen, and B. Rozoy. Self-stabilizing torus orientation. private communication, 1998.

[Dij74] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Journal of the ACM*, 17(11):643–644, 1974.

[DIM90] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Principles of Distributed Computing (PODC'90)*, pages 103–117, 1990.

[Her90] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35:63–67, 1990.

[IJ90] A. Israeli and M. Jalfon. Token management schemes and random walks yields self-stabilizing mutual exclusion. In *Principles of Distributed Computing (PODC'90)*, pages 119–130, 1990.

[IJ93] A. Israeli and M. Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104:175–196, 1993.

[RS62] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.