Information Processing Letters 74, pp. 203-207, 2000.

Randomized Uniform Self-stabilizing Mutual Exclusion [⋆]

Jérôme Durand-Lose [1]

Laboratoire ISSS, Bât. ESSI, BP 145, 06903 Sophia Antipolis Cedex, FRANCE.

A system is *self-stabilizing* if when started in any configuration it reaches a legal configuration, all subsequent configurations are legal. We present a randomized self-stabilizing mutual exclusion that works on any uniform graphs. It is based on irregularities that have to be present in the graph. Irregularities make random walks and merge on meeting. The number of states is bounded by $o(\Delta \ln n)$ where Δ is the maximal degree and n is the number of vertices. The protocol is also proof against addition and removal of processors.

Key words: Self-stabilization, Mutual exclusion, Fault tolerance, Distributed computing, Random walks.

1 Introduction

A system of interconnected processors often needs some mutual exclusion (ME) scheme to handle resources. At any time one and only one processor is privileged and may enter a critical section –in order, e.g., to access some resource. Each processor should be privileged infinitely often. The abstract concept of *token* is used to indicate the privilege.

Sometimes systems get corrupted, e.g., no privileged processor exists –access to crucial resources is lost– or more that one processor are privileged –devices might get confused by dual access. Since it is not possible to prevent any failure, we would like the system to regain consistency without any external intervention, and as sound as possible.

[⋆] This work was done while the author was in the LaBRI, UMR CNRS 5 800, Université Bordeaux I, Talence, FRANCE.

[1] `Jerome.DURAND-LOSE@unice.fr, http://www.i3s.unice.fr/~jdurand`

A system is *self-stabilizing* (SS) for a predicate if when started in any possible configuration it eventually reaches a legal configuration (*convergence*) and once in a legal configuration, it remains in legal configurations (*closure*) and the predicate is verified (*correctness*). A self-stabilizing system does not need to be initialized. Moreover, it is tolerant to transient fault: it regains consistency without any external intervention when a processor halts and recovers in any arbitrary state or communications get corrupted.

It this article we provide an algorithm for mutual exclusion 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.

The concept of self-stabilization was first introduced the pioneering paper by Dijkstra [Dij74] which presents some SSME on semi-uniform 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. Our *uniform self-stabilizing mutual exclusion* (USSME) is randomized.

There already exists self-stabilizing mutual exclusion algorithms in the literature: on rings ([Dij74,BD94]) or any graphs ([IJ90]), randomized ([DIM90,Her90,BD94]) or deterministic ([Dij74]), for some daemon scheduler or only in the synchronous case ([Her90]) . . . Some work on any graph but with a lesser version of uniformity: one or two processors may be different ([Dij74,DIM90]). Our USSME works on any graph under any daemon.

Israeli and Jalfon [IJ90] provide a SSME construction based on two levels of abstraction: Token Manipulation Scheme (TMS) and Graph Traversal (GT). In their article they provide two TMS (one with an infinite number of states and one for rings) and use random walks for GT. They prove lower bounds on the number of states for USSME on rings and on general graphs. At the end of their article, they mention three problems for further research. The third one is to find a TMS with a finite number of states for any communication graph with a bounded number of processors. Our USSME provides such a TMS.

Beauquier and Delaët [BD94] give a randomized USSME on oriented rings. Their technique is to impose a gap between the values of neighbor vertices. The gap is 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.

We extend the idea of Beauquier and Delaët to undirected graphs. The gap is computed in the following way. Each communication edge register holds an integer between 0 and $m-1$, where m does not divide the number of vertices. 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 (of neighboring processors), plus 1 (modulo m). The definition of m ensures that all processors can not be balanced simultaneously.

We call *bias* the difference from balance. An unbalanced processor 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. This provides the TMS.

The GT is as in [IJ90]: tokens make random walks in the graph and merge (or disappear) on meeting. We show that any two tokens eventually meet with probability 1. Eventually, only one token remains. Upper bounds on the meeting time of tokens can be found in [BHWG99].

The network is uniform, this means that processors are anonymous and that any two processors with the same degree are identical. Each processor is a randomized finite state machine. Communications are made through registers. One register is attached to each end of any communication edge. Along any edge, any incident processor may read the registers on both side but it can only write on the register on its side.

Processors are activated by some adversary *daemon* scheduler. To simplify the proofs, we only consider here the *central daemon* (it can only activate one processor at a time); nevertheless, our protocol is also proof against distributed (it can activate any set of processors) and read/write daemons (during an activation, a processor can only perform one single read or write operation) since passing the token consists only in one action.

We conclude the paper with a short remark on how to use our TMS for dynamical self-stabilization (processors can be added or removed).

2 Definitions

The network is *uniform*: processors are anonymous and any two processors with the same degree are identical. A processor is fully defined by its degree. The network is modeled by its (undirected, finite and connected) *communication graph* $G = (V, E)$. Let n be the number of vertices in the graph ($n = |V|$). For any vertex x, we denote E_x the collection of edges incident to x. If (x, y) is an edge, then x and y are *neighbors.*

Communications are handled with registers. For each edge (x, y) there is 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}).

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 a processor(s) by $c_1 \vdash c_2$. The transitive closure of $\vdash$ is denoted $\vdash^*$. A protocol is *self-stabilizing* (SS) for a given predicate $\mathcal{P}$ if there exist some set of *legal configurations* $\mathcal{L}$ ($\mathcal{L} \subset \mathcal{C}$) such that:
– *convergence*: the system eventually enters a legal configuration with probability 1;
– *closure*: 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 $c_1 \vdash^* c_2$, $c_2 \in \mathcal{L}$). The set $\mathcal{L}$ is an attractor of the system;
– *correctness*: starting on a legal configuration, the execution of the system will verify $\mathcal{P}$ with probability 1.

A *mutual exclusion* (ME) protocol ensures that in any (legal) 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).

Processors are activated by a scheduler(s). The scheduler chooses each time which processor(s) to activate. To ensure correctness, the classical model for scheduler is an adversary *daemon*. The daemon knows the whole configuration but ignores the result of the next coin toss. We only consider here the *central daemon* which can only activate one processor at a time.

3 USSME definition

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. The lowest integer which does not divide the number of vertices n is up bounded by $O(\log(n))$ as mentioned in [IJ90]. The number of states needed to implement the algorithm is bounded by $O(\Delta \log(n))$, where Δ is the maximal degree.

Definition 1 *A vertex x is* balanced *if it verifies the equation:*

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

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

Definition 2 *A processor is* privileged *when it is unbalanced. The difference from balance is called the* bias.

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

```
1   $bias := -1$ ;
2   for_each $(x,y)$ in $E_x$
3         $bias := ( bias + R_{xy} - R_{yx} ) \mod m$ ;
4   if ( $bias != 0$ ) then                 /* unbalanced */
5         /* begin critical section */
6         ...
7         /* end critical section */
8         if toss_coin () then
9               /* pass the privilege */
10              $(x,y)$ := randomly_chosen_incident_edge () ;
11              $R_{xy} := ( R_{xy} - bias ) \mod m$ ;       /* regain balance */
12        end
13  end
```

Fig. 1. Balancing algorithm for vertex x.

There is no copy of the contents of the registers inside the processor. This avoid duplication of data and risks of incoherence. 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 8) prevents malice actions of distributed and read/write daemons as in [DIM90,Her90].

An example of stabilization is given in Fig. 2 (only the activations of unbalanced processors that pass tokens are indicated). Balanced processors are indicated by '.' and each unbalanced one holds the value of its bias. The activated processors are circled, the chosen edge is in bolt. Only 4 states ($m = 4$) are needed on each register since there are 6 processors. Initially, the number of irregular gap is 4, it rapidly goes down to 1. Finally, the irregular gap makes a random walk inside the graph.

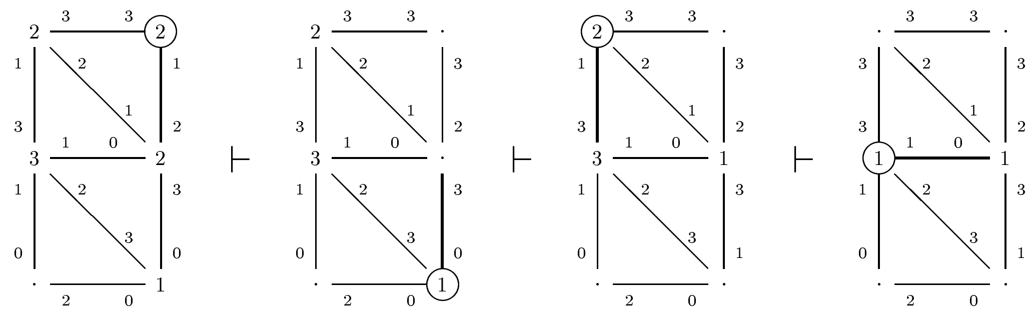
Fig. 2. Example of iterations, $n = 6$ and $m = 4$.

4 Proofs

Our USSME verifies the Reliability, No Deadlock and Self Stabilization requirements of [IJ90].

Lemma 3 *There is always at least one token, i.e., an 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:

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

It simplifies to $n \equiv 0 \mod m$ which contradicts the definition of m. □

Lemma 4 *The number of unbalanced processors is non-increasing.*

PROOF. Let us look what happens when a vertex is updated:
– if the vertex is balanced or is unbalanced but does not send the token then no register is modified, the number of unbalanced processors remains constant;
– if the vertex is unbalanced and sends the token then the algorithm 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. □

From the two previous lemmas, comes:

Lemma 5 *Once there is only one unbalanced processor, there remains only one forever.*

Lemma 6 *The system eventually reaches a configuration where only one processor is unbalanced with probability 1.*

PROOF. Since there must be at least one token in the graph, we only have to consider the case where there are more than one token. Let us consider any two tokens in the graph.

The dynamics given in the proof of Lem. 4 are the dynamics of random walk of token in the graph. When two tokens met, they merge (or disappear which accelerate the diminishing process and is thus not considered).

Let d be the distance between these two token and Δ the maximal degree of the graph. The first time that one of their processors is activated: the probability that the distance between token decreases by one is down bounded by $1/(2.\Delta)$ (it goes towards the other). Once the distance reach 0, they merge or both disappear, the number of token decreases.

If these processors are not activated, then other tokens moves. Since tokens make random walks, one of the moving tokens will enter one of the left idle processors and merge.

Two tokens will 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 1. $\square$

Finally, random walks ensure that the token eventually visits each processor infinitely often.

Although it is impossible for a processor to know that the whole configuration is correct, it can tell that the system is perturbed if it owns a bias different from $n \mod m$.

By taking $m = N+1$, one get a USSME which works even if a processor is added or removed (*dynamic* self-stabilization) as long as there are less than N processors. By removing the modulo part in (1) of Def. 1, on get a dynamic USSME with no bounds on the number of processors, but in this case, the number of state in infinite.

References

[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.

[BHWG99] N. Bshouty, L. Higham, and J. Warpechowska-Gruca. Meeting times of random walks on graphs. *Information Processing Letters*, 69:259–265, 1999.

[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.