Soyez efficace, rembobinez

Stéphane Devismes, Colette Johnen

Soyez efficace, rembobinez †

Stéphane Devismes[1] et Colette Johnen[2]

[1] *VERIMAG, Université Grenoble Alpes Grenoble, France - Stephane.Devismes@univ-grenoble-alpes.fr*
[2] *LaBRI, Université de Bordeaux, Bordeaux, France - Colette.Johnen@labri.fr*

Nous proposons un algorithme, appelé SDR, qui réinitialise de manière *autostabilisante* et totalement *distribuée* un *réseau anonyme*, lorsque c'est nécessaire, *i.e.*, chaque processus, détectant localement une anomalie, peut déclencher une réinitialisation. SDR est *coopératif* au sens où il coordonne les réinitialisations concurrentes afin de gagner en efficacité. Nous utilisons SDR pour rendre autostabilisant des algorithmes distribués. Notre approche permet de résoudre efficacement à la fois les problèmes statiques et dynamiques, comme le montrent les deux exemples d'algorithme utilisant SDR que nous proposons. Un rapport technique en ligne (arxiv.org/abs/1901.03587) présente de manière détaillée l'ensemble des résultats (corrections et complexités) de cet article.

Mots-clefs : autostabilisation, réinitialisation, compilateur autostabilisant, unisson, alliance.

1 Introduction

A *self-stabilizing* algorithm is able to recover a correct behavior in finite time, regardless of the *arbitrary* initial configuration of the system, and therefore also after a finite number of transient faults. For more than 40 years of researches, many self-stabilizing solutions have been proposed to solve various problems in various settings. Drawing on this experience, general algorithms, so-called *transformers* or *compilers*, that make distributed algorithms self-stabilizing have been proposed. Many transformers, *e.g.*, [6, 2], are based on *reset* algorithms. Such algorithms are initiated when an inconsistency is discovered in the network, and aim at reinitializing the system to a correct (pre-defined) configuration. A reset algorithm may be centralized at a leader process, or fully distributed, meaning multi-initiator (as our proposal here). In the fully distributed case, resets are locally initiated by processes detecting inconsistencies. This latter approach is considered as more efficient when the concurrent resets are coordinated. In other words, concurrent resets have to be *cooperative* to ensure the fast convergence of the system to a consistent global state.

Self-stabilization makes no hypotheses on the nature or extent of transient faults that could hit the system, and a self-stabilizing system recovers from the effects of those faults in a unified manner. Now, such versatility comes at a price, *e.g.*, after transient faults cease, there is a finite period of time, called the *stabilization phase*, during which the safety properties of the system are violated. Hence, self-stabilizing algorithms are mainly compared according to their *stabilization time*, the maximum duration of the stabilization phase.

General schemes and efficiency are usually understood as orthogonal issues. In this paper we tackle this problem by proposing an efficient self-stabilizing reset algorithm working in any anonymous connected network. Our algorithm is written in the atomic-state model, where executions proceed in atomic steps (in which a subset of enabled processes move, *i.e.*, update their local states) and the asynchrony is captured by the notion of *daemon*. The most general daemon is the *distributed unfair daemon*. Hence, solutions stabilizing under such an assumption are highly desirable, because they work under any other daemon assumption. The *stabilization time* is usually evaluated in terms of rounds, which capture the execution time according to the speed of the slowest processes. Another crucial issue is the number of local state updates, *i.e.* the number of *moves*. Indeed, the stabilization time in moves captures the amount of computations an algorithm needs to recover a correct behavior. There are many self-stabilizing algorithms proven under the

† This study has been supported by the ANR projects DESCARTES (ANR-16-CE40-0023) and ESTATE (ANR-16 CE25-0009-03).

distributed unfair daemon. However, analyzes of the stabilization time in moves is rather unusual and this is an important issue. Indeed, recently, several self-stabilizing algorithms which work under a distributed unfair daemon have been shown to have an exponential stabilization time in moves in the worst case, *e.g.*, the silent leader election algorithms from [4] (see [1]).

Contribution. We propose an efficient self-stabilizing reset algorithm, called SDR, working in any anonymous connected network. SDR is written in the atomic-state model, assuming a distributed unfair daemon. It is based on local checking and is multi-initiator. Concurrent resets are locally initiated by processes detecting inconsistencies, these latter being cooperative to gain efficient. SDR makes an input algorithm recovering a consistent global state within at most $3n$ rounds, where n is the number of processes. During a recovering, any process executes at most $3n+3$ moves. Our reset algorithm allows to build efficient self-stabilizing solutions for various problems and settings. In particular, it applies to both static and dynamic specifications. In the static case, the self-stabilizing solution we obtain is also *silent*: a silent algorithm converges within finite time to a configuration from which the values of the communication registers used by the algorithm remain fixed. The silent property usually implies more simplicity in the algorithm design. Moreover, a silent algorithm usually utilize less communication operations and communication bandwidth. To illustrate the efficiency of our method, we propose two efficient reset-based self-stabilizing algorithms, respectively solving the (asynchronous) unison problem in anonymous networks and the 1-minimal (f,g)-alliance in identified networks; see Section 3.

2 Self-Stabilizing Distributed Reset

The code of SDR is given in Algorithm 1. SDR aims at reinitializing an input algorithm I when necessary. SDR is self-stabilizing in the sense that the composition of I and SDR, noted I $\circ$ SDR, is self-stabilizing for the specification of I. Actually, I $\circ$ SDR is the distributed algorithm where the local program (I $\circ$ SDR)(u), for every process u, simply consists of all variables and rules of both I(u) and SDR (u).

Algorithm SDR works in anonymous connected networks and is multi-initiator: a process u can initiate a reset whenever it locally detects an inconsistency in I, *i.e.*, whenever the predicate $\neg$**P_ICorrect**(u) holds (*i.e.*, I is locally checkable). So, several resets may be executed concurrently. In this case, they are coordinated: a reset may be partial since we try to prevent resets from overlapping. To that goal, each process u maintains two variables in Algorithm SDR: $st_u \in \{C, RB, RC\}$, the *status* of u with respect to the reset, and $d_u \in \mathbb{N}$, the *distance* of u in a reset.

Variable st_u. If u is not currently involved into a reset, then it has status C, which stands for *correct*. Otherwise, u has status either *RB* or *RF*, which respectively mean *reset broadcast* and *reset feedback*. Indeed, a reset is based on a (maybe partial) *Propagation of Information with Feedback (PIF)* where processes reset their local state in I (using the macro *reset*) during the broadcast phase. When a reset locally terminates at process u (*i.e.*, when u goes back to status C by **rule_C**(u)), each member v of its closed neighborhood satisfies **P_reset**(v) (*n.b.*, we denote by $N(u)$ the set of u's neighbors, so $N(u) \cup \{u\}$ is the closed neighborhood of u), meaning that they are all in a pre-defined initial state of I. At the global termination of a reset, every process u involved into that reset has a state in I which is consistent *w.r.t.* that of its neighbors, *i.e.*, **P_ICorrect**(u) holds. Notice that, to ensure that **P_ICorrect**(u) holds at the end of a reset and for liveness issues, we enforce each process u stops executing I whenever a member of its closed neighborhood (in particular, u itself) is involved into a reset: whenever $\neg$**P_Clean**(u) holds, u is not allowed to execute I.

Variable d_u. This variable is meaningless when u is not involved into a reset (*i.e.*, $st_u = C$). Otherwise, the distance values are used to arrange processes involved into resets as a *Directed Acyclic Graph (DAG)*. This distributed structure allows to prevent both livelock and deadlock. Any process u initiating a reset (using rule **rule_R**(u)) takes distance 0. Otherwise, when a reset is propagated to u (*i.e.*, when **rule_RB**(u) is executed), d_u is set to the minimum distance of a neighbor involved in a broadcast phase plus 1.

Sample Execution. Assume the system starts from a configuration where $st_u = C$, for every process u. A process u detecting an inconsistency in I (*i.e.*, when $\neg$**P_ICorrect**(u) holds) stops executing I and initiates a reset using **rule_R**(u), unless one of its neighbors v is already broadcasting a reset, in which case it joins

Algorithm 1 Algorithm SDR, code for every process u

Inputs:

- $\mathbf{P\_ICorrect}(u)$, $\mathbf{P\_reset}(u)$: predicates from the input algorithm I
- $reset(u)$: macro from the input algorithm I

Variables: $st_u \in \{C, RB, RF\}$, $d_u \in \mathbb{N}$

Predicates:

- $\mathbf{P\_Correct}(u) \equiv st_u = C \Rightarrow \mathbf{P\_ICorrect}(u)$
- $\mathbf{P\_Clean}(u) \equiv \forall v \in N(u) \cup \{u\},\ st_u = C$
- $\mathbf{P\_R1}(u) \equiv st_u = C \wedge \neg\mathbf{P\_reset}(u) \wedge (\exists v \in N(u) \mid st_v = RF)$
- $\mathbf{P\_RB}(u) \equiv st_u = C \wedge (\exists v \in N(u) \mid st_v = RB)$
- $\mathbf{P\_RF}(u) \equiv st_u = RB \wedge \mathbf{P\_reset}(u) \wedge [\forall v \in N(u),\ (st_v = RB \wedge d_v \leq d_u) \vee (st_v = RF \wedge \mathbf{P\_reset}(v))]$
- $\mathbf{P\_C}(u) \equiv st_u = RF \wedge [\forall v \in N(u) \cup \{u\},\ \mathbf{P\_reset}(v) \wedge ((st_v = RF \wedge d_v \geq d_u) \vee (st_v = C))]$
- $\mathbf{P\_R2}(u) \equiv st_u \neq C \wedge \neg\mathbf{P\_reset}(u)$
- $\mathbf{P\_Up}(u) \equiv \neg\mathbf{P\_RB}(u) \wedge (\mathbf{P\_R1}(u) \vee \mathbf{P\_R2}(u) \vee \neg\mathbf{P\_Correct}(u))$

Rules:

- $\mathbf{rule\_RB}(u)$: $\mathbf{P\_RB}(u) \rightarrow st_u := RB;\ d_u := \text{argmin}_{(v \in N(u)\ \wedge\ st_v = RB)}(d_v) + 1;\ reset(u);$
- $\mathbf{rule\_RF}(u)$: $\mathbf{P\_RF}(u) \rightarrow st_u := RF;$
- $\mathbf{rule\_C}(u)$: $\mathbf{P\_C}(u) \rightarrow st_u := C;$
- $\mathbf{rule\_R}(u)$: $\mathbf{P\_Up}(u) \rightarrow st_u := RB;\ d_u := 0;\ reset(u);$

the broadcast of some neighbor by $\mathbf{rule\_RB}(u)$. To initiate a reset, u sets (st_u, d_u) to $(RB, 0)$ meaning that u is the root of a reset, and resets its I's variables to an pre-defined state of I, which satisfies $\mathbf{P\_reset}(u)$, by executing the macro $reset(u)$. Whenever a process v has a neighbor involved in a broadcast phase of a reset (status *RB*), it stops executing I and joins an existing reset using $\mathbf{rule\_RB}(v)$, even if its state in I is correct, (*i.e.*, even if $\mathbf{P\_ICorrect}(v)$ holds). To join a reset, v also switches its status to *RB* and resets its I's variables ($reset(v)$), yet it sets d_v to the minimum distance of its neighbors involved in a broadcast phase plus 1. Hence, if the configuration of I is not legitimate, then within at most n rounds, each process receives the broadcast of some reset. Meanwhile, processes (temporarily) stop executing I until the reset terminates in their closed neighborhood thanks to the predicate $\mathbf{P\_Clean}$.

When a process u involved in the broadcast phase of some reset realizes that all its neighbors are involved into a reset (*i.e.*, have status *RB* or *RF*), it initiates the feedback phase by switching to status *RF*, using $\mathbf{rule\_RF}(u)$. The feedback phase is then propagated up in the DAG described by the distance value: a broadcasting process u switches to the feedback phase if each of its neighbors v has not status *C* and if $d_v > d_u$, then v has status *RF*. This way the feedback phase is propagated up into the DAG within at most n additional rounds. Once a root of some reset has status *RF*, it can initiate the last phase of the reset: all processes involves into the reset has to switch to status *C*, using $\mathbf{rule\_C}$, meaning that the reset is done. The values *C* are propagated down into the reset DAG within at most n additional rounds. A process u can executing I again when all members of its closed neighborhood (that is, including u itself) have status *C*, *i.e.*, when it satisfies $\mathbf{P\_Clean}(u)$.

Hence, overall in this execution, the system reaches a configuration γ where all resets are done within at most $3n$ rounds. In γ, all processes have status *C*. However, process has not necessarily kept a state satisfying $\mathbf{P\_reset}$ (*i.e.*, the initial pre-defined state of I) in this configuration. Indeed, some process may have started executing I again before γ. However, the predicate $\mathbf{P\_Clean}$ ensures that no resetting process has been involved in these latter (partial) executions of I. Hence, SDR ensures that all processes are in I's states that are coherent with each other from γ. That is, γ is a so-called *normal configuration*, where $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds for every process u.

Stabilization of SDR. If a process u is in an incorrect state of Algorithm SDR (*i.e.*, if $\mathbf{P\_R1}(u) \vee \mathbf{P\_R2}(u)$ holds), we proceed as for inconsistencies in Algorithm I. Either it joins an existing reset (using $\mathbf{rule\_RB}(u)$) because at least one of its neighbors is in a broadcast phase, or it initiates its own reset using $\mathbf{rule\_R}(u)$. Notice also that starting from an arbitrary configuration, the system may contain some reset in progress. However, similarly to the typical execution, the system stabilizes within at most $3n$ rounds to a normal configuration. Finally, Algorithm SDR is also efficient in moves. Such an efficiency is mainly due to the coordination of the resets which, in particular, guarantee that if a process u is enabled to initiate a reset

(**P_Up**(u)) or the root of a reset with status *RB*, then it satisfies this disjunction since the initial configuration.

Requirements on `I`**.** According to the previous explanation, `I` should satisfy the following prerequisites:

1. `I` should not write into the variables of `SDR`, *i.e.*, variables st_u and d_u, for every process u.
2. For each process u, `I` should provide the two input predicates **P_ICorrect**(u) and **P_reset**(u) to `SDR`, and the macro *reset*(u). Those inputs should satisfy:
   (a) **P_ICorrect**(u) does not involve any variable of `SDR` and is closed by `I`.
   (b) **P_reset**(u) involves neither a variable of `SDR` nor a variable of a neighbor of u.
   (c) If $\neg$**P_ICorrect**$(u) \vee \neg$**P_Clean**(u) holds (*n.b.* **P_Clean**(u) is defined in `SDR`), then no rule of `I` is enabled at u.
   (d) If **P_reset**(v) holds, for every process in its closed neighborhood, then **P_ICorrect**(u) holds.
   (e) If u performs a move of `SDR` where it in particular modifies its variables in `I` only by executing *reset*(u) (only), then **P_reset**(u) holds in the resulting configuration.

3 `SDR`-Based Applications

To show the efficiency of our method, we have proposed two `SDR`-based self-stabilizing algorithms.

(f,g)-alliance. Given a graph $G = (V,E)$, and two non-negative integer-valued functions on processes f and g, a subset of processes $A \subseteq V$ is an *(f,g)-alliance* of G if and only if every process $u \notin A$ has at least $f(u)$ neighbors in A, and every node $v \in A$ has at least $g(v)$ neighbors in A. The (f,g)-alliance problem is the problem of finding a subset of processes forming an *(f,g)-alliance* of the network. The *(f,g)-alliance* problem is a generalization of several problems, *e.g.*, the k-domination problem is a $(k,0)$-alliance. The *minimum* (f,g)-alliance problem is $\mathcal{NP}$-hard. Here, we consider the problem of finding a 1-*minimal* (f,g)-*alliance*. The set A is a 1-*minimal* (f,g)-*alliance* if deletion of just one member of A causes A to be no more an (f,g)-*alliance*. Our self-stabilizing 1-minimal (f,g)-alliance algorithm is silent. Its stabilization time is in $O(n)$ rounds and $O(\Delta.n.m)$ moves, where D is the network diameter and m is the number of edges in the network. Until now there was no self-stabilizing algorithm solving that problem without any restriction on f and g.

Unison. We have then considered the problem of *(asynchronous) unison*. This problem is a clock synchronization problem: each process u holds a variable called *clock*. Each process should increment its clock infinitely often (liveness). Moreover, the difference between clocks of every two neighbors should be at most one increment at each instant (safety). We consider here periodic clocks, *i.e.*, the clock incrementation is modulo a so-called *period* $K > n$. Our self-stabilizing unison algorithm has a stabilization time in $O(n)$ rounds and $O(\Delta.n^2)$ moves. Actually, its stabilization times in round matches the one of the previous best existing solution [3]. However, it achieves a better stabilization time in moves, since the algorithm in [3] stabilizes in $O(D.n^3 + \alpha.n^2)$ moves (as shown in [5]), where α is greater than the length of the longest chordless cycle in the network.

References

[1] K. Altisen, A. Cournier, S. Devismes, A. Durand, and F. Petit. Self-stabilizing leader election in polynomial steps. *Inf. Comput.*, 254:330–366, 2017.

[2] A. Arora and M. G. Gouda. Distributed reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994.

[3] C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. In *PODC'04*, pages 150–159, 2004.

[4] A. K. Datta, L. L. Larmore, and P. Vemula. An $O(N)$-time self-stabilizing leader election algorithm. *JPDC*, 71(11):1532–1544, 2011.

[5] S. Devismes and F. Petit. On efficiency of unison. In *4th Workshop on Theoretical Aspects of Dynamic Distributed Systems, (TADDS '12)*, pages 20–25, 2012.

[6] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.