Necessary and Sufficient Conditions for 1-adaptivity

Joffroy Beauquier [1], Sylvie Delaët [2], Sammy Haddad [3]

[1]Université Paris-Sud
PCRI, LRI (CNRS UMR 8623),
INRIA Futurs
Orsay, France
jb@lri.fr

[2]Université Paris-Sud
LRI (CNRS UMR 8623)
Orsay, France
delaet@lri.fr

[3]Université Paris-Sud
LRI (CNRS UMR 8623)
Orsay, France
haddad@lri.fr

Abstract

A 1-adaptive self-stabilizing system is a self-stabilizing system that can correct any memory corruption of a single process in one computation step. 1-adaptivity means that if in a legitimate state the memory of a single process is corrupted, then the next system transition will lead to a legitimate state and the system will recover a correct behavior. Thus 1-adaptive self-stabilizing algorithms guarantee the very strong property that a single fault is corrected immediately and consequently that it cannot be propagated. Our aim here is to study necessary and sufficient conditions to obtain that property in order to design such algorithms. In particular we show that this property can be obtained even under the distributed demon and that it can also be applied to probabilistic algorithms.

We provide two self-stabilizing 1-adaptive algorithms that demonstrate how the conditions we present here can be used to design and prove 1-adaptive algorithms.

1. Introduction

Self-stabilization was introduced by E. W. Dijkstra in [7]. In this article he presents the three first self-stabilizing algorithms for the problem of mutual exclusion on a ring. Since, this notion has been proven to be one of the most important in the field of fault tolerance in distributed systems.

Indeed self-stabilization guarantees that regardless of its initial state the system will eventually reach a legitimate state (a state from which the execution satisfies the desired specification) in a finite bounded time. In particular, this implies that no matter how bad the memory corruption that hit the system are, the system will always regain a correct behavior by itself, without any external intervention. The only assumption made in self-stabilization is that the code of the processors cannot be corrupted. That is why self-stabilization is so useful for systems where some memory spaces are safe (code executed from ROM memory) and where the frequency of faults that hit the non safe memory zone (RAM memory) occur regularly and with a frequency not greater than a reasonable time of stabilization.

Self-stabilization is the best known solution, in terms of fault tolerance, for large distributed systems, where failures are a normal part of the behavior and where non systematic corrections are not possible. But one drawback to the application of self-stabilizing algorithms is that most of these algorithms still have a stabilization time proportional to the dimension of the system and not related to the actual number of failures.

Thus a small number of faults can force a majority of processors in the system to participate in the stabilization phase. In particular, non-faulty processors can start to behave incorrectly, and this for quite a long time even if, originally, their state was not corrupted. One known solution to this problem is time adaptive stabilization. Time adaptive stabilization guarantees a stabilization time directly proportional to the number of memory corruption that hit the system. For example, if only one processor is corrupted, then the system will stabilize in a constant time, whatever the network size is. But, as it appears in the

1-4244-0054-6/06/$20.00

literature, this attractive notion is not obvious to obtain.

Related works. The first time adaptive algorithms, as well as the notion of fault locality, where both presented in [18] and [15]. They were introduced in the context of non reactive problems. These articles present algorithms for the task of the persistent bit. They both stabilize in a time proportional to the number of corrupted nodes in the initial state of the system if that number does not exceed a predefined value. A first asynchronous fault containing algorithm was introduced in [16]. This algorithm, based on the same principle as the those presented in [18] and [15], also solves the problem of the persistent bit.

General methods for transforming silent self-stabilizing algorithms into time adaptive algorithms started to be studied in [11], [15] and [13]. In [11], the authors present a transformation which has a stabilization time in $\theta(1)$ if $k = 1$ and in $\theta(ST.diam)$ for $k \geq 2$, where k is the number of faults, ST the stabilization time of the non transformed algorithm and $diam$ is the diameter of the network. In [15], the idea is to replicate data and to use a voting strategy to repair data corrupted by transient faults. This transformation has an output stabilization time in $\theta(k)$ for a number of corrupted nodes lower or equal to $n/2$ (where n is the number of nodes in the system). Otherwise it stabilizes in $\theta(diam)$. [13] extends the idea of [15] to any number of corrupted nodes.

In [6] appears the first time adaptive solution to a reactive problem. This article presents a k-adaptive algorithm for the problem of mutual exclusion. A k-adaptive algorithm is an algorithm that stabilizes in a time proportional to the number of faults that hit the system if that number is smaller than a fixed k. Otherwise the system may not converge to a legitimate state. Two algorithms for the problem of broadcast are presented in [17] and [4]. The algorithm of [17] is used to prove that any non silent algorithm in synchronous systems has an adaptive solution. In [4] the measure of agility which quantifies the strength of a reactive algorithm against state corrupting faults is also defined and the broadcast algorithm is proven to guarantee error confinement with optimal agility within a constant factor.

Other approaches of fault containment can be found in [2], [9], [14], [1] and [19]. In [2] and [14] the notion of SuperStabilization is presented. A superstabilizing algorithm is a self-stabilizing algorithm that satisfies a passage predicate during recovery and thus restrains the effect of the fault. In [2] algorithms for coloring and spanning tree are given, and in [14] algorithms for the problem of mutual exclusion. In [2] a local stabilizer transforming any algorithm into a self-stabilizing algorithm that stabilizes in $\theta(k)$ is presented. This transformation was the first to introduce the use of snapshots in order to locally detect and correct inconsistencies due to transient failures. The correction was performed by a system of votes based on the snapshots. In [1], the power supply technique is presented. It consists of regularly broadcasting and consuming informations. The corrupted information is then consumed whereas the correct information keeps circulating since its power is supplied by correct processors. The first algorithms for the problems of graph coloring and the dining philosophers resistant to Byzantine faults are presented in [19]. This algorithms are locally tolerant to faults. That is, any processor far enough from the faulty processors will behave correctly.

An interesting impossibility results can be found in [10]. In this article it is proven that a large class of reactive problems do not have an adaptive solution in asynchronous networks.

The first article to introduce the problem of correcting a single failure in one computation step is [12]. The authors present a transformer of self-stabilizing algorithm into probabilistic 1-adaptive algorithm. To do that they enlarge the processors states space with states that can only be reached after a failure. Thus if a processor is corrupted and reached one of those state. The system can easily detect the error and correct it in just one computation step.

Our contribution. We start by formalizing our model and the definition of 1-adaptive self-stabilizing algorithm.

Then we study the necessary and sufficient conditions for a self-stabilizing algorithm to be 1-adaptive. We start by studying these conditions for strictly 1-local algorithms that works on networks without triangles. Those algorithms are such that in case of a single fault in the system then at least two processors have a view of the system that does not correspond to a legitimate configuration. A network without triangles is a network that does not contain two processors with a common neighbor.

In the same section we also study what these conditions become if we change the hypothesis to : the algorithm is silent and it has its legitimate configurations 3-distant one from the other. For this conditions there are no more assumption made on the system topology neither on the locality of the algorithm.

We finish by presenting two 1-adaptive self-stabilizing algorithms. The first one is an election algorithm that works under the weakly fair demon. Such a demon guarantees that if a processor is continuously enabled then it will eventually be activated. This algorithm works on hypercubes and illustrates how the first conditions we have stated can be interpreted to get 1-adaptive algorithms. In the same way we present a 1-adaptive self-stabilizing probabilistic algorithm for the naming problem that works on complete graphs.

Finally we discuss how those principle can be generalized to get a transformer that makes some self-stabilizing algorithms 1-adaptive.

2. Model

Our model of distributed systems refers to [3] and the definitions of transition systems to [20]. A distributed system is a set of processors connected by communication links. It is represented by a **communication graph** $G = (\mathcal{P}, \mathcal{E})$ such that $\mathcal{P}$ is a set of processors and $\mathcal{E}$ a set of edges, $l = (p_i, p_j)$, where $(p_i, p_j) \in \mathcal{P}^2$ and $p_i \neq p_j$. Two processors p_i and p_j of G are said to be **neighbors** and can communicate in G if and only if $(p_i, p_j) \in \mathcal{E}$. They communicate via **shared registers**. We note $\mathcal{N}_p$ the set of p's neighbors in G. We say that a communication graph G has no **triangle** if for any pair , (p_i, p_j), of processors in G,$p_i \neq p_j$ there is no processor p_k, $p_k \neq p_i$ and $p_k \neq p_j$ such that $(p_i, p_k) \in \mathcal{E}$ and $(p_j, p_k) \in \mathcal{E}$. A **path** of length k in G is a set of processors noted $ch = \{p_1...,p_k\}$ such that $\forall i \in \{1...k-1\}, (p_i, p_{i+1}) \in \mathcal{E}$. In a communication graph the **distance** between two processors p_i and p_j, noted $dist(p_i, p_j)$, is the length of the shortest path from p_i to p_j in $\mathcal{P}$.

A **processor** p is a state machine. Its state, noted e_p, is the vector of all the values of its variables. A processor has a set of **guarded rules**, noted $r = \{label_1...,label_n\}$, of the form $< label >::< guard > \rightarrow < action >$ where $label$ is the identifier of the rule, $guard$ is a boolean expression over p's and p's neighbors' variables and $action$ updates the values of p's variables. A **configuration** $C \in \mathcal{C}$ is a vector $C = (e_{p_1}...,e_{p_n})$ of the processors' states of S. We note $C_{|P}$ the restriction of the configuration C for to a set of processors $P = \{p_i...,p_j\}$, $P \in \mathcal{P}$ and $Dist(C, C')$, the **distance** between two configurations C and C'such that $Dist(C, C')$ is equal to the number of processors which have a different state in C and C'. We say that a guarded rule is **executable** in a configuration C if and only if p evaluates the guard of this rule to true in C. We also say that a processor is **enabled** in a configuration C if and only if it has at least one of its guarded rule executable in C.

A distributed system can be modeled as a **transition system**. A transition system is a pair $S = (\mathcal{C}, \mathcal{T})$, where $\mathcal{C}$ is the set of all the possible configurations of the system, $\mathcal{T}$ is the set of all the possible transitions of S. A **transition** of $\mathcal{T}$ is a triple (C, t, C'), such that $(C, C') \in \mathcal{C}^2$ and t is a set of guarded rules. The activation of the guarded rules in t, where t is a subset of the executable rules in C, brings the system in the configuration C'. We also note $C \xrightarrow{p_i,...,p_j} C'$ a transition where $p_i, ..., p_j$ are the processors that execute an action in C. An **execution** E_α of a system S is a maximum sequence of computation steps, $e = (C_0, C_1..., C_i...)$, where $C_0 = \alpha$ and such that for every $i \geq 0$, there is a t such that $(C_i, t, C_{i+1}) \in \mathcal{T}$, otherwise C_i is terminal (no processor is enabled in C_i). The set of the possible executions of the system is restricted by the **demon**. The **distributed demon** (Cf [8]) chooses, for each transition, any subset of the enabled processors in C to apply at least one of their executable rules. The **weakly fair** distributed demon, is a distributed demon that guarantee that if a process is continuously enable in a execution then it will eventually be activated. With the **synchronous demon**, in a transition $T \in \mathcal{T}$, all the enabled processors apply one of their executable rules. In this article we use the definitions of $Ball$ and $View$ presented in [5]. Let G be a communication graph, d an integer and p a processor of G, **Ball(p, d)** is the set B of nodes b of $\mathcal{P}$ such that $dist(p, b) \leq d$. The **view** $\mathcal{V}_p^d(C)$ at distance d of the processor p in a configuration C contains the state of the processors of $Ball(p, d)$ in C, $\mathcal{V}_p^d(C) = C_{|Ball(p,d)}$. We also use the term of $Ball$ in a configuration C instead of view $\mathcal{V}_p^1(C)$, also noted $C_{|p \cup \mathcal{N}(p)}$.

The **specification** of a problem is a predicate over the system executions. The specification of a **static problem** is a predicate over the system configurations. We call **output variables** of a system the set of variables which have to verify the specification. Let $S = (\mathcal{C}, \mathcal{T})$ be a transition system and Spe be a specification of a problem. Then S is **self-stabilizing** for Spe if and only if there is a subset $\mathcal{L}$ of configurations of $\mathcal{C}$, called **legitimate configurations** of S such that: *(i)* Every execution that starts in a configuration of $\mathcal{L}$ satisfies Spe; *(ii)* Every execution reaches a configuration of $\mathcal{L}$. A **silent** self-stabilizing algorithm A is a self-stabilizing algorithm such that all the legitimate configurations are terminal.

A self-stabilizing algorithm A is **1-adaptive** if and only if for any pair of configurations (C, C') such that $C \in \mathcal{L}$, $C' \notin \mathcal{L}$ and $Dist(C, C') = 1$, we have, for every C'' such that there exists $T = (C', t, C'') \in \mathcal{T}, C'' \in \mathcal{L}$.

A **correct view** $\mathcal{V}_{p_i}^d(C)$ for a self-stabilizing algorithm A, in a transition system S, is a view such that there exists a legitimate configuration $l \in \mathcal{L}$ of S in which there is a processor p_j such that $\mathcal{V}_{p_i}^d(C) = \mathcal{V}_{p_j}^d(C)$. Let B be the $Ball(p, 1)$, we say that **B is enabled** in a configuration C if and only if p is enabled in C. We will also say that $\mathcal{V}_p^1(C)$ is enabled. A self-stabilizing algorithm is **1-local** if and only if every configuration C that only contains correct views at distance 1 is legitimate.A self-stabilizing algorithm is **strictly 1-local** if it is 1-local and if for any configuration C containing a processor p_i whose view at distance 1 is not correct, then there is a processor p_j, neighbor of p_i, whose view is also incorrect.

3. Necessary and Sufficient Conditions for 1-adaptivity

The definition of 1-adaptivity states that 1-adaptive algorithms guarantee the correction of a single transient fault hitting a legitimate state in just one transition. Thus it makes 1-adaptive self-stabilizing algorithms being optimal in terms of fault containment for the distributed systems fac-

ing local faults. Indeed after the system reaches a legitimate state, a single fault cannot be propagated because the system regains immediately a legitimate state. We study in this section the feasibility of such algorithms. For that, we start by establishing some necessary and sufficient conditions for a self-stabilizing algorithm to be 1-adaptive.

First we assume the distributed demon (contrary to the large majority of the studies on fault containment which are made under the synchronous demon). Our goal is to demonstrate that even under a demon that seems hardly compatible with the property of 1-adaptivity we still can get some positive results.

We first consider strictly 1-local self-stabilizing algorithms, assuming the network is without triangle for which the correction task will be easier. Then we consider self-stabilizing algorithms whose legitimate configuration are not too close one from the other.

3.1. Networks without triangle

The networks we consider in this section have no triangles. We can notice that several common topologies have this property (rings, trees, grid, hypercubes, etc.). This hypothesis is made here for technical reason but it helps to highlight the impact of the system topology on the property of fault containment.

Definition 3.1 *Let A be a strictly 1-local self-stabilizing algorithm executed on a communication graph G without triangle, with an associated transition system S, under the distributed demon. For all $p_i \in \mathcal{P}$, $\forall C \in \mathcal{L}$, $C' \notin \mathcal{L}$ such that $C_{|\{p_i\}} \neq C'_{|\{p_i\}}$ and $C_{|\mathcal{P}\setminus\{p_i\}} = C'_{|\mathcal{P}\setminus\{p_i\}}$. Let:*

- *Condition 1 (Locality) No processor p_j with a correct view $\mathcal{V}^1_{p_j}(C')$ is enabled in C'.*
- *Condition 2 (Correction) Any transition of S from C' implying only processors p_j located in $B = Ball(p_i, 1)$ brings the system into a configuration C'' such that all the views $\mathcal{V}^1_{p_j}(C'')$ are correct.*

Proposition 3.1 *(Condition 1 $\wedge$ Condition 2) is a necessary and sufficient condition for A to be 1-adaptive.*

Proof . First, let us prove $Condition1 \wedge Condition2 \Rightarrow$ *1-adaptive*.

Let C and C' be two configurations of S such that $C \in \mathcal{L}$, $C' \notin \mathcal{L}$, $Dist(C, C') = 1$ and $C_{|p_i} \neq C'_{|p_i}$.

The view at distance 1 in C and in C' which are not centered on a processor of $Ball(p_i, 1)$ are identical and thus correct since C is legitimate. From $Condition1$, only the balls centered on a processor of $Ball(p_i, 1)$ with an incorrect associated view in C' are possibly enabled. However as A is self-stabilizing, then at least one processor is enabled in C'.

Then $Condition2$ implies that any step taken by the algorithm makes correct all the view centered in a processor of $Ball(p_i, 1)$ and thus that any transition of the algorithm starting from C' brings the system in a configuration C'' where for all $p \in \mathcal{P}, \mathcal{V}^1_p(C'')$ is correct. C'' is thus legitimate since A is strictly 1-local. We can conclude that $Condition1 \wedge Condition2 \Rightarrow$ *1-adaptive*.

Second, we prove that the reciprocal is also true and that *1-adaptive* $\Rightarrow Condition1 \wedge Condition2$. To do that, we prove the following proposition $\neg Condition1 \vee \neg Condition2 \Rightarrow \neg$*1-adaptive*.

If $Condition1$ is false, then there is at least one ball, $B = Ball(p_j, 1)$, with a correct associated view $\mathcal{V}^1_{p_j}(C')$ that is enabled in C'. As C' is illegitimate and the algorithm is strictly 1-local then by definition we know that there are two processors of the system p_j and p_k such that $\mathcal{V}^1_{p_j}(C')$ and $\mathcal{V}^1_{p_k}(C')$ are not correct. Since the system has a topology without triangle then if $p_j \in \mathcal{N}(p)$ (respectively $p_k \in \mathcal{N}(p_i)$) then $p_k \notin \mathcal{N}(p_i)$ (respectively $p_j \notin \mathcal{N}(p_i)$). The system can thus activate only B since we are under the distributed demon. It then gets in a configuration C'' where at least $\mathcal{V}^1_{p_j}(C'')$ or $\mathcal{V}^1_{p_k}(C'')$ is not correct since from what precedes we have either $\mathcal{V}^1_{p_j}(C') = \mathcal{V}^1_{p_j}(C'')$ or $\mathcal{V}^1_{p_k}(C') = \mathcal{V}^1_{p_k}(C'')$. Moreover as A is strictly 1-local, C'' is illegitimate and the algorithm is not 1-adaptive. So we have that $\neg Condition1 \Rightarrow \neg$*1-adaptive*.

Now if $Condition1$ is true and $Condition2$ is false, then there is again a transition of the system which preserves an incorrect view and thus leads the system to an illegitimate configuration. In this case the algorithm is not 1-adaptive and we obtain the following implication $Condition1 \wedge \neg Condition2 \Rightarrow \neg$*1-adaptive*.

We can now conclude that $\neg Condition1 \vee \neg Condition2 \Rightarrow \neg$*1-adaptive*.□

These conditions put forward the fact that the correction of a single fault must only imply processors in the neighborhood of the faulty processor. It shows that in case of a single fault, a processor must be able to recognize whether it is in the neighborhood of a corrupted process. It also highlights the necessity for the algorithm to be silent if the network contains no triangle. In particular we can derive from this conditions the following propositions.

Proposition 3.2 *Let A be a 1-adaptive self-stabilizing algorithm, strictly 1-local, executed on a network without triangle, under the distributed demon. If in a given legitimate configuration, there exists two processors p_1 and p_2 such that $dist(p_1, p_2) \geq 3$ and whose corruption can lead to an illegitimate configuration, then this configuration is silent.*

Proof. Let $C \in \mathcal{L}$, p_1 and p_2 be two processors, $dist(p_1, p_2) \geq 3$, such that there exist C^1 and C^2 such that for $i \in \{1, 2\}$, $C^i \notin \mathcal{L}$, $C_{|\mathcal{P}\setminus\{p_i\}} = C^i_{|\mathcal{P}\setminus\{p_i\}}$ and $C_{|\{p_i\}} \neq C^i_{|\{p_i\}}$. By definition of C^i, for all p such that $dist(p, p_i) \geq 2$, we have $\mathcal{V}^1_p(C^i) = \mathcal{V}^1_p(C)$. Since C is legitimate $\mathcal{V}^1_p(C^i)$ and $\mathcal{V}^1_p(C)$ are correct. Since the algorithm verifies condition 1 of proposition 3.1, $\mathcal{V}^1_p(C^i)$ is correct and enabled and thus $\mathcal{V}^1_p(C)$ is enabled. Then for every p and every $i \in \{1, 2\}$such that $dist(p, p_i) \geq 2, \mathcal{V}^1_p(C)$ is enabled.

By the triangular inequality, and by the choice of p_1 and p_2 we have, for all $p \in \mathcal{P}, dist(p, p_1) + dist(p, p_2) \geq dist(p_1, p_2) > 2$. As the distances are positive integers and by virtue of the inequality we obtain that, for all $p \in \mathcal{P}$ there exists $i \in \{1, 2\}$ such that $dist(p, p_i) \geq 2$. But for every p there exist $i \in \{1, 2\}$ such that $dist(p, p_i) \geq 2$. Then $\mathcal{V}^1_p(C)$ is not enabled and we obtain that no rule is applicable in C. Thus C is a silent configuration. □

We get the following corollary :

Corollary 1 *An algorithm that verifies proposition 3.2 for all its legitimate configurations is silent.*

As the corruption of any single processor usually suffices to put a distributed system in an illegitimate configuration. We get from corollary 1 that a self-stabilizing algorithm under the distributed demon has to be silent to be 1-adaptive.

3.2. Restriction on the legitimate configuration density

Corollary 1 connects 1-adaptivity with the fact that no correct processor can be activated. In particular conditions 3.1 proves that only a processor in the neighborhood of the corrupted one can take a step. That is why we study in this section the conditions for a silent self-stabilizing algorithm to be 1-adaptive.

Moreover our aim here is to study if the same principle as in [2] can be applied to get 1-adaptivity. What we want to do is to see how the replication of every processor state in its neighbors can help for 1-adaptivity.

That is why we assume that the algorithms we consider here have legitimate configurations at distance at least 3 from each other. In fact if every processor has its state replicated in its neighbors snapshots then if one processor changes its state, at least two processors (itself and it(s) neighbor(s)) have a different state.

Moreover we can note that this assumption is verified by all algorithms solving problems with only one legitimate configuration, such that the computation of the network size, the topology learning, some leader election, etc.

Definition 3.2 *Let A be a silent self-stabilizing algorithm and $\mathcal{L}$ its legitimate configurations, such that for all $(l, l') \in \mathcal{L}^2, Dist(l, l') \geq 3$. For all $p_i \in \mathcal{P}$, $\forall C \in \mathcal{L}$, $C' \notin \mathcal{L}$ such that $C_{|\{p_i\}} \neq C'_{|\{p_i\}}$ and $C_{|\mathcal{P}\setminus\{p_i\}} = C'_{|\mathcal{P}\setminus\{p_i\}}$. Let:*

- *Condition 1 (Locality) The only enabled processor in $Ball(p_i, 1)$ is the processor p_i.*
- *Condition 2 (Correction) The activation of $Ball(p_i, 1)$ in C' brings p_i back in the state $C_{|p_i}$.*

Proposition 3.3 *(Condition 1 ∧ Condition 2) is a necessary and sufficient conditions for A to be 1-adaptive.*

Proof. First we prove the implication $Condition\ 1 \wedge Condition\ 2 \Rightarrow 1 - adaptive$.

In C' all the view $\mathcal{V}^1_p(C')$ such that $p \in \mathcal{P} \setminus \{p_i\} \cup \mathcal{N}(p_i)$ are correct since $\mathcal{V}^1_p(C') = \mathcal{V}^1_p(C)$ and C is legitimate. Since A is silent any processor with a correct view at distance 1 is not enabled. We obtain that in C' only the processors with an incorrect view are potentially enabled. Thus only the processors of $Ball(p_i, 1)$ may be enabled. According to $Condition\ 1$ the only enabled ball in C' is $Ball(p_i, 1)$. Moreover according to $Condition 2$ the activation of this ball puts p_i in the same state as in C. So the only possible transition from C' is the transition $C' \xrightarrow{p_i} C$ where by assumption $C \in \mathcal{L}$ and thus A is 1-adaptive.

Let us prove now the reciprocal. As for the preceding conditions, we will prove that $\neg Condition\ 1 \vee \neg Condition\ 2 \Rightarrow \neg 1 - adaptive$. Let us suppose that $Condition\ 1$ is false. We get that, in C' all the views $\mathcal{V}^1_p(C')$ such that $p \in \mathcal{P} \setminus \{p_i\} \cup \mathcal{N}(p_i)$ are correct since $\mathcal{V}^1_p(C') = \mathcal{V}^1_p(C)$ and C is legitimate. Thus $\neg Condition\ 1$ implies that there exists a processor p_j different from p_i such that $\mathcal{V}^1_{p_j}(C')$ is not correct and $Ball(p_j, 1)$ is enabled in C'. Since we are under the distributed demon the system may perform the transition $C' \xrightarrow{p_j} C''$, where $Dist(C', C'') = 1$ and $C'_{|p_j} \neq C''_{|p_j}$. We thus obtain that $Dist(C, C'') = 2$. However by assumption C is legitimate and all the legitimate configurations are separated by a distance of at least 3. Thus C'' is not legitimate and we have a transition from C' which brings the system into a legitimate configuration. Finally we have $\neg Condition\ 1 \Rightarrow \neg 1 - adaptive$.

Let us consider now that $Condition 1$ is true and $Condition\ 2$ is false. Then we know that the only enabled ball is $Ball(p_i, 1)$ and that $C' \xrightarrow{p_i} C''$ with C'' such that $C''_{|p_i} \neq C_{|p_i}$ and $C''_{|\mathcal{P}\setminus p_i} = C'_{|\mathcal{P}\setminus p_i} = C_{|\mathcal{P}\setminus p_i}$. Thus $Dist(C, C'') = 1$, and by assumption C is legitimate. Because the legitimate configurations are at least at distance 3 from each other, C'' is not legitimate and then $Condition\ 1 \wedge Condition\ 2\neg \Rightarrow \neg 1 - adaptive$. We conclude that $\neg Condition\ 1 \vee \neg Condition\ 2 \Rightarrow \neg 1 - adaptive$ and that by consequence $1 - adaptive \Rightarrow Condition\ 1 \wedge Condition\ 2$. □

The results of this section point out the fact that, if the legitimate configurations (of a silent self-stabilizing algorithm) are all at distance at least 3 from each other, then the only way to be 1-adaptive is to be able to go back to the previous legitimate configuration.

4. Examples

4.1. Deterministic 1-adaptive election algorithm for hypercubes

The first algorithm is an election algorithm for hypercubes with identifiers, under the weakly fair demon. This algorithm uses the specificity of hypercubes in two ways. First a processor knows the diameter of the of the system by the degree. Then an upper bound for the distance to the leader is known. This information is used for avoiding an infinite circulation of fake Ids.

Second, the network has no triangle. Moreover, for any pair of nodes (p_i, p_j) there exists at least two processor p_k and p_l such that $p_k \in \mathcal{N}_{p_i}, p_l \in \mathcal{N}_{p_i}$ and $dist(p_k, p_j) = dist(p_l, p_j) = dist(p_i, p_j) - 1$. Then in case of a single corruption there is locally a majority of processors holding a correct information.

4.2. Algorithm description

The algorithm consists in maintaining in each node a variable $Leader$ that contains the smallest Id in the network and a variable $DistanceToTheLeader$, that contains the corresponding distance. The basic idea is that a new $Leader$ is chosen when a neighbor proposes a smaller identifier together with a distance inferior to the diameter (action A3). When a processor detects an inconsistency (no neighbors has the same $Leader$ or the distances are inconsistent) it becomes its own $Leader$ (action A2). Then a fake $Leader$ (resulting from the initial corruption) is eliminated.

The property of 1-adaptivity is obtained by adding an action that is executed if in a configuration a processor can see that all its neighbors have the same $Leader$ and that their distance to this $Leader$ are mutually consistent with each other. Then this processor chooses this $Leader$ and the smallest associated distance incremented by one (action A1). The neighbors of the apparently faulty processor are blocked and cannot execute any action. This is obtained thanks to the function $Freeze$ that freezes processors if they agree with each of their neighbors but one. As a matter of fact this function freezes every correct neighbor of a faulty processors in the case of a single fault and does not block the stabilization of the system otherwise. Note that, for every fake $Leader$, the processors with the smallest distance to that $Leader$ always correct themselves. Because they do not have any neighbor with the same $Leader$ and a smaller distance to it. So they evaluate the guard of action $A3$ to true and since the demon is weakly fair they will eventually execute this action. By induction on the distance to the fake $Leader$, we prove that every processor with that fake Id will become its own $Leader$.

Moreover if every processor has a $Leader$ that is a real Id of a processor, then the smallest Id will be propagated normally. That is proven again by induction on the distance to the true $Leader$.

4.3. Probabilistic 1-adaptive naming algorithm

In this section we describe a probabilistic 1-adaptive self-stabilizing algorithm for the naming problem on a complete network of size N. This algorithm has legitimate configurations at distance N from each other. It is well known that naming cannot be achieved in a deterministic way. Thus the only known solutions are probabilistic. That is the case for the algorithm we present here, which is also 1-adaptive.

This algorithm works on complete networks. We assume that each processor has previously numbered its registers 1,2,..,$N-1$. We note $Reg_p[i]$ (or $Reg[i]$ if we are clearly talking about processor p) the value of the i^{th} register of the processor p. Then for two processors p and q of the system, if we have $Reg_p[i]$ corresponding to q and $Reg_q[j]$ corresponding to p then the function $GetOrder$ returns j for $GetOrder(Reg_p[i])$ and i for $GetOrder(Reg_q[j])$. The numbering of registers cannot be corrupted.

It can be noted that there exists 1-adaptive deterministic algorithms for other problems (such that the renaming problem). But we chose here to present a probabilistic algorithm to illustrate that conditions 3.1 also works for probabilistic algorithms and how they can be used to design 1-adaptive algorithms.

4.4. Algorithm description

This algorithm solves the naming problem in anonymous complete graph. For a complete graph the naming problem is equivalent to the coloration problem.

Each processor executing this algorithm has four variables, $Name$, $Names$, $Snapshot$ and Reg. The variable $Name$ represents the name of the processor, the variable $Names$ is an array used to collect the $Name$ of the neighbors of the processor. The i^{th} entry in $Names$ corresponds to the register numbered i. The variable $Snapshot$ is an array containing a view of the system. Finally Reg is an array and it contains at the index i the variable values of the neighbor corresponding to the i^{th} register. Reg is completely updated before the evaluation of the guarded actions.

Algorithm 1 Election Algorithm for hypercubes

Variables :
Id : positive integer, id of the processor p_i ;
Leader : integer, id of the current leader for p_i;
DTTL : integer, distance to the leader (i.e, length of the shortest from path p_i to the leader);

Fonctions :
MinLeader: returns $(p_j.Leader \mid p_j \in \mathcal{N}_{p_i}, \forall p_k \in \mathcal{N}_{p_i} p_k.Leader \leq p_j.Leader)$;
MinDistance(leader): returns $(p_j.DTTL \mid p_j \in \mathcal{N}_{p_i}, p_j.Leader = leader, \forall p_k \in \mathcal{N}_{p_i}, p_k.Leader = leader, p_k.DTTL \leq p_j.DTTL)$;

Freeze :
returns(

$$
\begin{array}{ll}
[& (Leader, DTTL) = (Id, 0) \\
& \wedge \ \forall p_j \in \mathcal{N}_{p_i}, Leader > p_j.Id, \\
& \wedge \ \exists p_{bad} \in \mathcal{N}_{p_i}, (p_{bad}.Leader, p_{bad}.DTTL) \neq (Id, 1) \\
& \quad \wedge \ \forall p_{ok} \in \mathcal{N}_{p_i} \setminus \{p_{bad}\}, (p_{ok}.Leader, p_{ok}.DTTL) = (Id, 1) \quad] \\
\vee & \\
[& \forall p_j \in \mathcal{N}_{p_i} \cup \{p_i\}, Leader \leq p_j.Id \\
& \wedge \ \exists p_l \in \mathcal{N}_{p_i}, (p_l.Leader, p_l.DTTL) = (Leader, DTTL - 1) \\
& \wedge \ \exists p_{bad} \in \mathcal{N}_{p_i}, (p_{bad}.Leader \neq Leader \vee |DTTL - p_{bad}.DTTL| > 1) \\
& \quad \wedge \ \forall p_{ok} \in \mathcal{N}_{p_i} \setminus \{p_{bad}\}, Leader = p_{ok}.Leader \wedge |DTTL - p_{ok}.DTTL| = 1) \quad] \\
\vee & \\
[& \forall p_j \in \mathcal{N}_{p_i} \cup \{p_i\}, Leader \leq p_j.Id \\
& \wedge \ \exists p_{bad} \in \mathcal{N}_{p_i}, \\
& \quad (\quad Leader = p_{bad}.Id \wedge (p_{bad}.Leader, p_{bad}.DTTL) \neq (Leader, 0) \\
& \quad \wedge \forall p_{ok} \in \mathcal{N}_{p_i} \setminus \{p_{bad}\}, Leader = p_{ok}.Leader \wedge p_{ok}.DTTL = 2 \quad) \quad]
\end{array}
$$

);

Actions :

$$
\begin{array}{ll}
\textbf{A1 :} & \neg \ (\quad \exists p_j \in \mathcal{N}_{p_i}, p_j.Id = Leader, DTTL = 1 \\
& \qquad \wedge \quad \forall p_k \in \mathcal{N}_{p_i} \setminus \{p_j\}, p_k.Leader = Leader, p_k.DTTL = 2 \\
& \qquad \wedge \quad (p_j.Leader, p_j.DTTL) \neq (p_j.Id, 0) \quad) \\
& \wedge \ MinLeader \neq Id \\
& \wedge \ (Leader, DTTL) \neq (MinLeader, MinDistance(MinLeader) + 1) \\
& \wedge \ \forall p_j \in \mathcal{N}_{p_i}, \forall p_k \in \mathcal{N}_{p_i} \\
& \qquad p_j.Leader = p_k.Leader \wedge p_j.DTTL \leq \delta \\
& \qquad \wedge \quad |p_j.DTTL - p_k.DTTL| \leq 2 \\
& \rightarrow \ Leader := MinLeader, DTTL := MinDistance(MinLeader) + 1;
\end{array}
$$

$$
\begin{array}{ll}
\textbf{A2 :} & \neg(A1 \vee Freeze) \\
& \wedge \ \neg Freeze \\
& \wedge \ (Leader, DTTL) \neq (Id, 0) \\
& \wedge \ (\quad \forall p_j \in \mathcal{N}_{p_i}, Leader.p_j \geq Id \\
& \qquad \vee \ \forall p_j \in \mathcal{N}_{p_i}, (p_j.Leader, p_j.DTTL) \neq (Leader, DTTL - 1) \quad) \\
& \rightarrow \ Leader := Id, DTTL := 0;
\end{array}
$$

$$
\begin{array}{ll}
\textbf{A3 :} & \neg(A1 \vee A2 \vee Freeze) \\
& \wedge \ \exists p_j \in \mathcal{N}_{p_i}, (Leader.p_j, p_j.DTTL) < (Leader, DTTL - 1), \\
& \qquad Leader.p_j < Id, p_j.DTTL < \delta \\
& \rightarrow \ Leader := MinLeader, DTTL := MinDistance(MinLeader) + 1;
\end{array}
$$

Algorithm 2 Naming Algorithm for complete networks

Variables
Name : integer $\in \{0...N\text{-}1\}$;
Snapshot : array of $(Name, Names)$ of size N-1;
Names: integer array of size N-1;

Functions
TakeSnapshot : $\forall i \in \{0...N\text{-}1\}$, $Snapshot[i] := (Reg[i].Name, Reg[i].Names)$;
GetNewName : returns $(random(\{0...N\text{-}1\} \setminus \{Reg[i].Name | i \in \{1...N\text{-}1\}, \forall k \in \{1...N\text{-}1\},$
$Reg[i].Name \neq Reg[k].Name\})$;)
ConsistentSnapshot : returns $(\forall i \in \{0...N\text{-}1\}$, $Snapshot[i] = (Reg[i].Name, Reg[i].Names))$;
(N-1)ConsistentSnapshots :
if the $Snapshot$ of p is consistent with the $Snapshot$ of $(N\text{-}2)$ of its neighbors and these $Snapshot$ represent a legitimate configuration **returns** true **else** false;
Consensus :
if the $Snapshot$ of the $(N\text{-}1)$ neighbors of p are mutually consistent and these $Snapshots$ represent a legitimate configuration where $(p.Name, p.Names) \neq Reg[i].Snapshot[GetOrder(Reg[i])]$ for all i $\in \{0...N\text{-}1\}$ **returns** $Snapshot[GetOrder(Reg[1])]$ **else** 0;

Actions

A1 : $\exists (i,j,k) \in \{1...N\text{-}1\}^3, j \neq k, (Name = Reg[i].Name \wedge Reg[j].Name = Reg[k].Name)$
$\rightarrow$ $GetNewName$

A2 : $\forall (i,j) \in \{1...N\text{-}1\}^2, Reg[i].Name \neq Reg[j].Name$
$\wedge$ $\exists k \in \{1...N\text{-}1\}, Name = Reg[k].Name$
$\wedge$ $\neg(N\text{-}1)ConsistentSnapshots$
$\rightarrow$ $GetNewName$;

A3 : $\forall (i,j) \in \{1...N\text{-}1\}^2, i \neq j, Reg[i].Name \neq Name \wedge Reg[j].Name \neq Reg[i].Name$
$\wedge$ $\exists k \in \{1...N\text{-}1\}, Names[k] \neq Reg[k].Name$
$\rightarrow$ $\forall i \in \{1...N\text{-}1\}, Names[i] := Reg[i].Name, TakeSnapshot$;

A4 : $\forall (i,j) \in \{1...N\text{-}1\}^2, i \neq j, Reg[i].Name \neq Name \wedge Reg[j].Name \neq Reg[i].Name$
$\wedge$ $\forall k \in \{1...N\text{-}1\}, Names[k] = Reg[k].Name \wedge \neg ConsistentSnapshot$
$\wedge$ $\neg(N\text{-}1)ConsistentSnapshots$
$\rightarrow$ $TakeSnapshot$;

A5 : $\forall (i,j) \in \{1...N\text{-}1\}^2, Reg[i].Name \neq Reg[j].Name$
$\wedge$ $\exists k \in \{1...N\text{-}1\}, Name = Reg[k].Name$
$\wedge$ $Consensus \neq 0$
$\wedge$ $Consensus \neq (Name, Names)$
$\rightarrow$ $Name = Consensus.Name, Names = Consensus.Names, TakeSnapshot$;

A legitimate state is defined as follows. Every pair of processors (p, q) where q corresponds to the i^{th} register of p, is such that $p.Name \neq q.Name$, $p.Names[i] = q.Name$ and $p.Snapshots[i] = (q.Name, q.Names)$.

The principles used for this algorithm are inspired by the local stabilizer of [2]. Our goal is to enlarge the system state with copies of each processor state on each node of the network thanks to the variables $Names$ and $Snapshot$, in order to get the property of N-distant legitimate configurations. Then, after the corruption of a single processor, the nearest legitimate state is uniquely determined. This example illustrates how the necessary and sufficient conditions we gave can be used for designing 1-adaptive algorithms.

Stabilization is in two phases. The first one is probabilistic, and put the system either into a correctly named configuration (the $Name$ variables are all different but the $Names$ and $Snapshots$ variables are not necessary consistent) or into an illegitimate configuration at distance 1 from a legitimate one. To do this, every processor that has at least one neighbor with the same $Name$ picks up a random $Name$ (action A1 or A2), among the set of names not yet attributed. These probabilistic actions lead with probability 1 to a configuration where only deterministic actions are possible.

The second phase is deterministic. It either corrects in one action the state of a single faulty processor (action A5), or it updates the variables $Names$ and $Snapshot$ (action A3 and A4).

The 1-adaptive property is obtained by ensuring that in the case of a single corruption there is only one enabled action (only the faulty processor is enabled). A single corruption leads the system into a configuration where only deterministic actions are possible. If the single fault does not corrupt the $Name$ variable of the faulty processor only actions A3 or A4 are possible and the faulty process returns in the correct configuration by updating the $Names$ and $Snapshot$ variables. If the single fault corrupts the $Name$ variable of the faulty processor, the functions $(N\text{-}1)ConsistentSnapshots$ and $Consensus$ are used for fault confinement. Indeed in this case, as the $N-1$ correct processors evaluate $(N\text{-}1)ConsistentSnapshots$ to true, they are not enabled. The faulty processor evaluates $(N\text{-}1)ConsistentSnapshots$ to false and applies A5. The faulty processor updates its $Name$ variable with value computed by the consensus function and updates its $Names$ and $Snapshot$ values. These new values are consistent which the $Names$ and $Snapshots$ variables of the correct processors.

5. Conclusion

We have introduced in this paper a new kind of fault containing self-stabilizing algorithms. The concept of containing a single fault in one transition was introduced in [12]. In [12] only restricted types of corruptions are considered and many types of corruptions of a single processor can lead several other processes to participate to the stabilization, delaying the time of recovery.

We presented here two necessary and sufficient conditions for a self-stabilizing algorithm to be able to correct any possible memory corruption of a single process in just one computation step. With the help of these conditions, we gave two 1-adaptive self-stabilizing algorithms.

The first condition shows two things. First, if the algorithm we want to be 1-adaptive has a local specification (a specification that implies that every local view of every processor of the system has to be correct) then 1-adaptivity implies almost systematically the fact that the algorithm has to be silent. We can also state that the correction has to be local and that only the faulty processor or one of its neighbor can participate to the correction. Second the system topology has an important impact on the property of 1-adaptivity. In particular in a network with no triangle the correction is possible if and only a processor with a correct view cannot be activated.

The second condition points out the fact that if the legitimate configurations are far from each other, then the only way to be 1-adaptive is to return in the legitimate state that precedes the corruption. In this case the neighbors of the corrupted node have to be frozen and the corrupted node must return in its previous state.

We used these conditions for designing two algorithms. The first algorithm works on hypercubes. The stabilization is obtained thanks to the knowledge of the diameter. The property of 1-adaptivity is obtained by the fact that the correct information is forwarded to each processor in the system from the correct source by at least two neighbors. Thus even if one processor is corrupted every processor has at least one neighbor that continues to forward the correct information. Then we guarantee that the stabilization will not be blocked by the freezing mechanism because the sources of fake information eventually disappear.

The second algorithm uses a well known concept presented in [2]. Every processor holds a local snapshot of its neighbors. The processor detecting an inconsistency can then be frozen. An appropriate system of vote on the snapshots allows to restore the corrupted processor to its state before the corruption. This algorithm gives an example of a probabilistic 1-adaptive algorithm.

Note that our results have in fact a larger application domain and there are two simple extensions. First the assumption of a single corruption can be slightly relaxed. Our results also apply when an arbitrary number of memory corruptions hit the system, provided that no processor has two neighbors that are simultaneously corrupted. If it is not the case, the general stabilizing mechanisms allows anyway the

system to recover (unfortunately in more than one step).

The necessary and sufficient conditions we have presented can also be seen as a simple way to prove that a self-stabilizing algorithm is 1-adaptive. In fact those conditions points out the fact that the property of 1-adaptivity is very local. Thus to prove the 1-adaptivity there is no need to study every possible execution of the system but just local conditions.

Our future works will consist in generalizing the method we used to design the algorithms presented in this article to get general transformers of silent self-stabilizing algorithms to 1-adaptive self-stabilizing algorithms. In fact we strongly believe that several different problems can be solved on specific topologies in exactly the same way as for the election algorithm and in particular on hypercubes. We will also extend the snapshot technique to larger classes of networks.

References

[1] Y. Afek and A. Bremler. Self-stabilizing unidirectional network algorithms by power-supply. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 111–120, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[2] Y. Afek and S. Dolev. Local stabilizer. In *Israel Symposium on Theory of Computing Systems*, pages 74–84, 1997.

[3] H. Attiya and J. L. Welch. *Distributed computing: fundamentals, simulations and advanced topics*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1998.

[4] Y. Azar, S. Kutten, and B. Patt-Shamir. Distributed error confinement. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 33–42, Boston, Massachusetts, July 2003.

[5] J. Beauquier, S. Delaët, S. Dolev, and S. Tixeuil. Transient fault detectors. In *Proceedings of the 12th International Symposium on DIStributed Computing (DISC'98)*, number 1499, pages 62–74, Andros, Greece, 1998. Springer-Verlag.

[6] J. Beauquier, C. Genolini, and S. Kutten. Optimal reactive k-stabilisation: the case of mutual exclusion. In *18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, May 1999.

[7] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, 1974.

[8] S. Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, 2000. Ben-Gurion University of the Negev, Israel.

[9] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, page 255, New York, NY, USA, 1995. ACM Press.

[10] C. Genolini and S. Tixeuil. A lower bound of dynamic k-stabilization in asynchronous systems. In *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 212–222, Osaka University, Suita, Japan, Octobre 2002.

[11] S. Ghosh, A. Gupta, T. Herman, and S. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Symposium on Principles of Distributed Computing*, pages 45–54, 1996.

[12] Herman and Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *IPL: Information Processing Letters*, 73, 2000.

[13] T. Herman. Observations on time-adaptive self-stabilization, Oct. 15 1997.

[14] T. Herman. Superstabilizing mutual exclusion. *Distributed Computing*, 13(1):1–17, 2000.

[15] S. Kutten and B. Patt-Shamir. Time-adaptive self stabilization. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'97)*, pages 149–158, 1997.

[16] S. Kutten and B. Patt-Shamir. Asynchronous time-adaptive self stabilization. In *PODC*, page 319, 1998.

[17] S. Kutten and B. Patt-Shamir. Adaptive stabilization of reactive protocols. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 24, 2004.

[18] S. Kutten and D. Peleg. Fault-local distributed mending. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*, pages 20–27, August 1995.

[19] M. Nesterenko and A. Arora. Tolerance to unbounded byzantine faults. In *The 21th IEEE Symposium on Reliable Distributed Systems, (SRDS '02)*, pages 22–31, Washington - Brussels - Tokyo, Oct. 2002. IEEE.

[20] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, New York, NY, USA, 1994.