1-adaptivity

Joffroy Beauquier, Sylvie Delaët and Sammy Haddad

Laboratoire de Recherche en Informatique, UMR CNRS 8623,
Université de Paris Sud, 91405 Orsay Cedex, France
email:{jb,delaet,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. Then, in one step the system recovers a correct behavior. Then 1-adaptive self-stabilizing algorithms guarantee that a single failure will not propagate and will be corrected immediately. This is not the case for most of the existing self-stabilizing algorithms. Our aim here is to study the possibility of designing such algorithms. More precisely we discuss necessary and sufficient conditions for a self-stabilizing algorithm to be 1-adaptive. Such conditions yield a simple way to verify whether or not a self-stabilizing algorithm is also 1-adaptive, as they only look at a small subset of the system states and as they can be verified locally. We also provide examples of self-stabilizing 1-adaptive algorithms.

Keywords: Distributed Algorithms, Fault Tolerance, Self-stabilization, Fault Containment, 1-Adaptivity.

Résumé

Un système auto-stabilisant 1-adaptatif a la capacité de corriger n'importe quelle corruption mémoire d'un seul de ses processeurs en un seul pas d'exécution et n'importe quelle corruption plus importante en un temps fini borné. Donc si le système se trouve dans une configuration légitime et qu'un seul de ses processeurs est corrompu de telle sorte que le système passe dans une configuration illégitme alors quelque soit la transition effectuée par le système à partir de cette configuration il vérifira à nouveau sa spécification. Les algorithmes auto-stabilisants 1-adaptatifs permettentdonc de garantir que si le système est frappé par une fauteunique alors celle-ci ne pourra pas se propager et sera corrigée immédiatement, ce que ne garantissent pas la plus part des algorithmes auto-stabilisants existant. Notre but dans cet article est d'étudier la faisabilité de tels algorithmes. Nous discutons pour cela des conditions nécessaires et suffisantes pour qu'un algorithme auto-stabilisant soit 1-adaptatif. Nous avons d'abord considéré les algorithmes auto-stabilisants strictement 1-locals avec une restriction sur la topologie des systèmes qui les exécutent. Puis nous avons étudié les algorithmes silencieux dont les configurations légitimes sont éloignées les unes des autres. Ces conditions nous donnent une manière simple de vérifier qu'un algorithme auto-stabilisant est 1-adaptatif, puisqu'elles ne portent que sur un petit nombre des configurations du système et qu'elles se vérifient de manière très locale. Nous présentons aussi des exemples.

Mots clefs: Algorithmes Distribués, Tolérance aux fautes, Auto-stabilisation, Confinement de fautes, 1-Adaptivité.

Chapter 1

Introduction

Self-stabilization was introduced by E. W. Dijkstra in [Dij74]. 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 notions 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 its specification) in a finite bounded time. In particular, this implies that no matter how bad the memory corruptions 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 efficient in 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) are not greater than the time of stabilization.

Self-stabilization is the best known solution for large distributed systems, where failures are a normal part of the behavior and where non systematic corrections are not possible. One difficulty in 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 are not related to the actual number of failures.

Thus a small number of faults can force most of the processors of the system to participate in the stabilization phase. In particular, non-faulty processors can start to behave incorrectly, and this for quiet a long time even if, originally, their state was not corrupted. One solution to this problem is time adaptive stabilization. Time adaptive stabilization guarantees a stabilization time directly proportional to the number of memorycorruptions that hit the system. For example, if only one processor is corrupted, then the system will stabilize in a constant time, whatever is the network size. But,as it appears in the literature, this attractive notion, in terms of its practical usefulness, is not obvious to obtain.

Time adaptivity was first introduced in the context of non-reactive problems, where specifications are on the system configurations and not on its executions. In [KP95] the notion of fault locality is introduced, as well as an algorithm for the simple task called the persistent bit. This algorithm has a $\theta(k.log(n))$ stabilization time, where k is the number of corrupted nodes in the initial state and n is the number of nodes of the system. In this article the number of faults to ensure time adaptivity mustnot exceed $f = \theta(n.log(n))$.In [KP97] another solution for the same problem is presented. Its stabilization time is $\theta(k)$ for $k \leq n/2$. An asynchronous version of this algorithm can be found in [KP98].

[KP97] and [Her97] give transformations of silent self-stabilizing algorithms into time adaptive algorithms. A silent algorithm is an algorithm that does not change any of its output values after reaching a legitimate configuration. This transformation has an output stabilization time in $\theta(k)$ for a number of faults

equal to k, $k \leq n/2$, and in $\theta(diam)$ otherwise, where $diam$ is the diameter of the network. In [KP97], the idea is to replicate data and to use a voting strategy to repair data corrupted by transient faults. In [GGHP96], the authors present another transformation which has a stabilization time in $\theta(1)$ if $k = 1$ and in $\theta(T.diam)$ for $k \geq 2$, where T is the stabilization time of the non transformed algorithm. The first k-adaptive reactive algorithm was presented in [BGK99]. This algorithm solves the problem of mutual exclusion and stabilizes in a time proportional to the number of faults that hit the system if that number is smaller than a fixed k. If the number of faults is greater than k then the system may not converge to a legitimate state. In [KP01], it is proven that any non silent algorithm in synchronous systems has an adaptive solution. The proof is based on an adaptive algorithm for broadcast, which is used to distribute all output variables' values in a time adaptive manner. In [AKP03] the measure of agility which quantifies the strength of a reactive algorithm against state corrupting faults is defined and another broadcast algorithm that guarantees error confinement with optimal agility within a constant factor, is given. In [GT02] it is stated that a large class of reactive problems do not have an adaptive solution in asynchronous networks.

Two similar approaches of fault containment can be found in [AD97] and [NA02]. In [AD97] a local stabilizer transforming any algorithm into a self-stabilizing algorithm that stabilizes in $\theta(k)$ is presented. Every processor manages a snapshot of the system. A faulty processor can detect inconsistencies with its neighbor correct view. It then regains the state that it had before the corruption thanks to a system of vote on its neighbors' snapshot. Algorithms that guarantee the tolerance of Byzantine behavior are presented in [NA02]. Byzantine processors are processors subject to faults that are not limited in time.

We introduce here 1-adaptivity which is a stronger notion of time adaptivity. An 1-adaptive self-stabilizing algorithm can correct in one step the memory corruption of a single processor in the network (and thus completely prevent the fault propagation). More precisely, we present here necessary and sufficient conditions for 1-adaptivity. These conditions concern only a subset of the system configurations. It will appear that they can easily be locally checked. These conditions ease the task of verifying whether a self-stabilizing algorithm is also 1-adaptive. They also show that there is no need to study every execution of the system as they just involve a few local properties of a small subset of configurations. These conditions are a simple tool for verification of the 1-adaptive property and they also highlight the need of some properties for an algorithm to be 1-adaptive.

First we restrict ourselves, for technical reasons, to some particular networks, in which there are no triangles. The scheduler is quite general : we assume the distributed demon (at each step an arbitrary subset of the enabled actions of the system are applied). Secondly we present those conditions for silent algorithms having legitimate configurations not too close one from the other (two different legitimate configurations differ by the states of at least three processors). Here there are no restrictions on the network topology. The assumption on the legitimate configurations allows us to treat other types of problems as, contrary to the first case, problems which are not strictly 1-local.

Finally we provide a 1-adaptive self-stabilizing algorithm for the problem of naming a complete network. This example shows how the second necessary and sufficient conditions we present here can also be interpreted to design 1-adaptive algorithms. This particular example proving also that even probabilistic algorithms can be 1-adaptive, this algorithm being the first known probabilistic fault containing algorithm.

Chapter 2

Model

2.1 Communication Graph

A **communication graph** $G = (\mathcal{P}, \mathcal{E})$ is a tuple 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 $p_i \neq p_j$.

Each edge is associated to a neighborhood relationship. Two processors p_i and p_j can communicate in G if and only if $(p_i, p_j) \in \mathcal{E}$. We note $\mathcal{N}_p$ the set of p's neighbors in G. Communications among neighboring processors are carried out by **shared registers**.

A communication graph G has no **triangle** if for any pair of processors in G, (p_i, p_j), there is no processor p_k 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 sequence of processors noted $ch = \{p_1 ..., p_k\}$ such that $\forall i \in \{1...\text{k-1}\}, (p_i, p_{i+1}) \in \mathcal{E}$.

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 $\mathcal{P}$.

2.2 Distributed System

A distributed system is a communication graph where each **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 all values of p's variables.

A distributed system is associated to a **transition system**. A transition system is a tuple $S = (\mathcal{C}, \mathcal{T})$, where $\mathcal{C}$ is the set of all the configurations of the system and $\mathcal{T}$ is the set of all the transitions of S.

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 to a set of processors $P = \{p_i, ..., p_j\}$, $P \subset \mathcal{P}$. We note $Dist(C, C')$, 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 deterministic **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'.

Thus in the deterministic systems transitions $T = (C, t, C')$ are completely defined by the starting configuration C and the set of actions t executed in C.

Figure 2.1: A deterministic transition $T = (C, t, C')$

In the probabilistic systems the configuration C' also depends on the random values updated by the actions of t. The set of values given to random variables during a transition T is noted $rand$. Thus in these systems a transition T becomes $T = (C, t, rand, C')$, also noted $T^{rand} = (C, t, rand, C^{rand})$, and it is associated to a probability noted, $Pr(T)$. This probability correspond to the probability to get the value $rand$ among all the possible set of values for the probabilistic variables updated by T. In a probabilistic system we have that for any transition T, $Pr(T) > 0$ and $\sum_{T=(C,t,\_,\_)} Pr(T) = 1$ for any configuration C.

Let $T = (C, t, C')$ be a deterministic transition (repectively $T' = (C, t, rand, C')$ be a probabilistic transition) we note $first(T)$ the configuration C and $last(T)$ the configuration C' (respectively $first(T')$ the configuration C and $last(T')$ the configuration C').

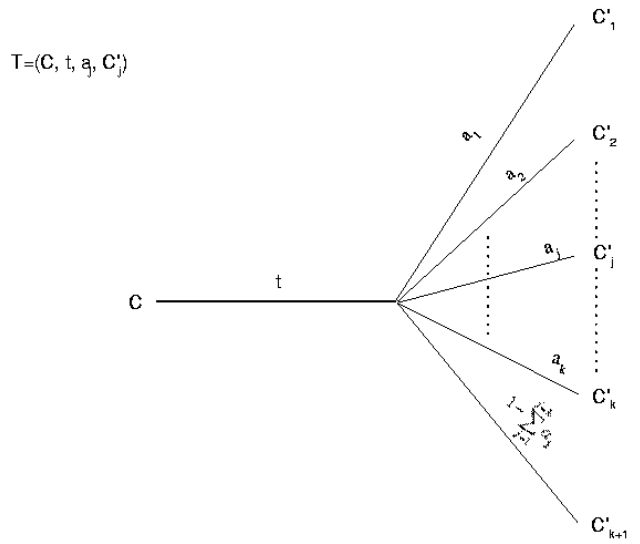

Figure 2.2: Probabilistic transitions $T_j = (C, t, a_j, C'_j)$ based on $T = (C, t, ,)$

We use the notation $C \xrightarrow{p_i...,p_l} C'$ for a transition where $p_i..., p_l$ are the processors that execute an action in C.

An **execution** e of a system S is a maximal sequence of transition, $e = (T_0, T_1..., T_i...)$, such that for every $i \geq 0$, $last(T_i) = first(T_{i+1})$ or $last(T_i)$ is final, that is, no processor is enabled in $last(T_i)$.

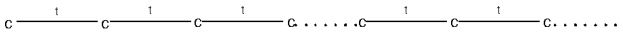

Figure 2.3: Execution in the deterministic model

2.3 Demons

The set of possible transitions of the system is restricted by the demon. A demon is a predicate over the executions of the system. It chooses for each transition $T = (C, t, C')$ the set t of executed actions among all the executable actions.

The **distributed** demon determines, for each transition, any subset of the enabled processors in C to apply at least one of their executable rules. With the **synchronous** demon, in a transition $T \in \mathcal{T}$, all the enabled processors apply one of their executable rules.

2.4 Trees of Executions and Strategies

The following definitions are taken from [Gra00].

An **execution tree rooted in C** for a system S, noted $\mathcal{T}ree(C)$, where C is a configuration of S, is the tree-representation of all the executions of S starting in C (see figure 2.4 and 2.5).

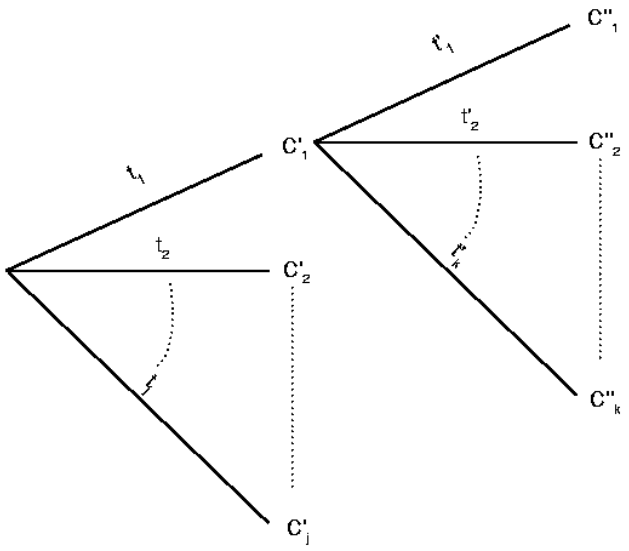

Figure 2.4: Beginning of the execution tree in the deterministic model

A **sub-tree of execution of degree** 1 **rooted in** C for a systemt S, is a restriction of $\mathcal{T}ree(C)$ such that for every node of $\mathcal{T}ree(C)$ either there is no outgoing branch or every possible transition from that node has the same label t (*i.e.* the same actions are executed).

The interaction between a demon and a distributed system is called a **strategie**.

Let S be a transition system, let D be a demon and let C be a configuration of S. We call a **demon strategy** rooted in C a sub-tree of degree 1 of $\mathcal{T}ree(C)$ such that any execution of the sub-tree verifies D.

Let st be a strategy. A **cone** of st corresponding to a non maximal execution h of st, noted $Cone_h$, is the set of all possible executions of st with the same prefix h. **last(h)** denotes the last configuration of h.

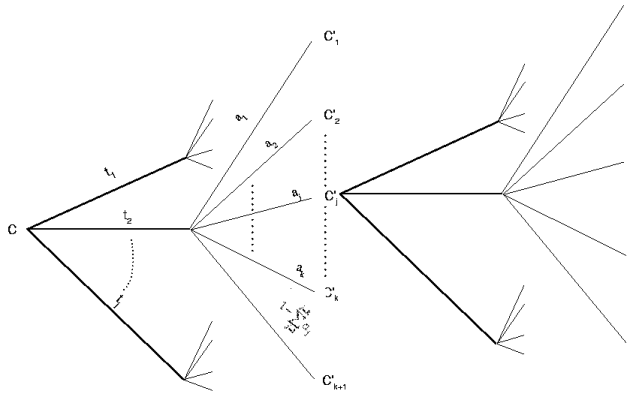

Figure 2.5: Beginning of the execution tree in the probabilistic model

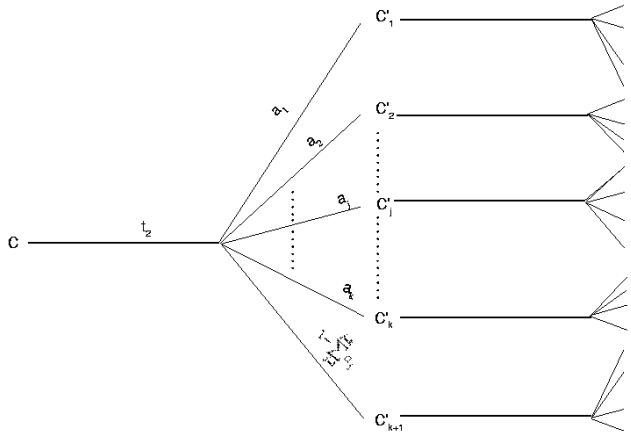

Figure 2.6: Beginning of a sub-tree of degree 1

The number of transitions T in the history h is denoted by $|h|$.

In deterministic systems a strategy st is reduced to an execution (cf 2.3).

The **probability measure of a cone** $\mathcal{C}_h$ is the probability measure of the prefix h, that is the product of the probability of every transition occurring in h. Thus for any strategy st and any cone $Cone_h$ of st we have $Pr(Cone_h) > 0$.

A cone $Cone_{h'}$ is called a **sub-cone** of $Cone_h$ if and only if $h' = [hf]$, where f is an execution factor.

We can note that, since that in a probabilistic system we have $\sum_{T=(C,t,\_,\_)} Pr(T) = 1$ for any configuration C, we also have for any cone of any strategy $\sum_{|f|=1} Pr(Cone_{hf}) = Pr(Cone_h)$.

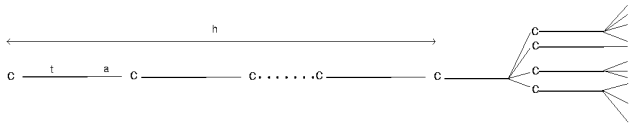

Figure 2.7: Beginning of a cone

2.5 Deterministic Self-Stabilization

The **specification of a problem** is a predicate over the system executions. The specification of a **static problem** is a predicate over the system configuration. 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 $\mathcal{C}$, named **legitimate configurations** of S such that:

- Every execution that starts in a configuration of $\mathcal{L}$ verifies Spe.
- Every execution reaches a configuration of $\mathcal{L}$.

A **silent** self-stabilizing algorithm A is a self-stabilizing algorithm such that all the configurations reached by an execution of A starting from a legitimate configuration have the same projection on the output variables of A. The legitimate configurations of a silent self-stabilizing algorithm are said to be **silent configurations**.

2.6 Probabilistic Self-Stabilization

The notation $x \vdash Pred$ means that the element x of $\mathcal{X}$ satisfies the predicate $Pred$ defined on the set $\mathcal{X}$.

We use the notation $\mathcal{E}_{Pred}$ to represent the set of executions of a strategy st under a demon D that reaches a configuration that satisfies the predicate $Pred$. The probability associated to $\mathcal{E}_{Pred}$ is the sum of the probability of the cones $Cone_h$ of st such that $h = [h'f], |f| = 1, \neg(last(h') \vdash Pred)$ and $last(f) \vdash Pred$.

A predicate $Pred$ is **closed** for the executions of a distributed system if and only if when $Pred$ holds in a configuration C, $Pred$ also holds in any configuration reachable from C.

Let S be a system, D be a demon and st be a strategy satisfying D. Let $\mathcal{C}_{|Pred}$ be the set of all system configurations satisfying a closed predicate $Pred$. The set of executions of st that reach configurations $C \in \mathcal{C}_{|Pred}$ is denoted by $\mathcal{E}_{Pred}$ and its probability by $Pr(\mathcal{E}_{Pred})$.

In this paper we study a silent probabilistic algorithm. The definition of self-stabilization for this particular type of algorithms is restricted to the probabilistic convergence definition.

A system S is **self-stabilizing** under a demon D for a closed legitimacy predicate L on the system configurations, if and only if in any strategy st of S under D the following condition holds:

- (Probabilistic Convergence) The probability of the set of executions of st, starting from C, reaching a configuration C', such that C' satisfies L is 1 . Formally, $\forall st, Pr(\mathcal{E}_L) = 1$.

2.7 Convergence of Probabilistic Stabilizing Systems

We recall here the theorem of local convergence presented in [Gra00]. This theorem can be used for proving self-stabilization via probabilistic attractors.

Let L_1 and L_2 be two predicates defined on configurations. L_2 is a **probabilistic attractor** for L_1 in a system S under the demon D, noted $L_1 \rhd_{prob} L_2$, if and only if for every strategie st of S under D such that $Pr(\mathcal{E}_{L_1}) = 1$, we have $Pr(\mathcal{E}_{L_2}) = 1$.

Let st be a strategy, PR_1 and PR_2 be two predicates on configurations, where PR_1 is a closed predicate. Let δ be a positive probability and N a positive integer. Let $Cone_h$ be a cone of st with $last(h) \vdash PR1$

and let $\mathcal{F}$ denote the set of sub-cones $Cone_{h'}$ of the cone $Cone_h$ such that the following is true for every sub-cone $Cone_{h'}$: $last(h') \vdash PR2$ and $|h'| - |h| \leq N$. The cone $Cone_h$ satisfies the property of **local convergence** from $PR1$ to $PR2$, noted $LC\ (PR_1, PR_2, \delta, N)$, if and only if $Pr(\bigcup_{Cone_{h'} \in \mathcal{F}} Cone_{h'}) \geq \delta$.

Theorem 2.7.1 *Let st be a strategy of the system S under the demon D. Let PR_1 be a closed predicate on configurations such that $Pr(\mathcal{E}_1) = 1$. Let PR_2 be another closed predicate on configurations and $PR_{12} = PR_1 \wedge PR_2$. If $\exists \delta_{st} > 0$ and $\exists n_{st} > 1$ such that st verifies $LC(PR_1, PR_2, \delta_{st}, n_{st})$ then $Pr(\mathcal{E}_{PR_2}) = 1$.*

2.8 1-adaptivity

In this paper we introduce the new notion of 1-adaptivity, as follow.

A *deterministic self-stabilizing algorithm A* is **1-adaptive** if and only if there is a subset of legitimate configurations called stable configurations, noted $\mathcal{SC}$, such that:

- (Correction) For any pair of configurations (C, C') such that $C \in \mathcal{SC}$, $C' \notin \mathcal{L}$ and $Dist(C, C') = 1$, we have, for every C'' such that $C' \rightarrow C''$, $C'' \in \mathcal{L}$.
- (Convergence) Every execution of A reaches a configuration of $\mathcal{SC}$.
- (Closure) Every execution that starts in a configuration of $\mathcal{SC}$ reaches only configurations of $\mathcal{SC}$

A *probabilistic self-stabilizing algorithm A* executed by a system S under a demon D is **1-adaptive** if and only if there is a subset of legitimate configurations called stable configurations, noted $\mathcal{SC}$, such that:

- (Correction) For any pair of configurations (C, C') such that $C \in \mathcal{SC}$, $C' \notin \mathcal{L}$ and $Dist(C, C') = 1$, we have, for every C'' such that $C' \rightarrow C''$, $C'' \in \mathcal{L}$.
- (Probabilistic Convergence) Every strategy st of S under D starting in a configuration C verifies that the set of executions in st that reach a configuration of $\mathcal{SC}$, noted $\mathcal{ECS}$ has a probability 1. Formally, $\forall st, Pr_{st}(\mathcal{ECS}_{st}) = 1$.
- (Closure) Every execution that starts in a configuration of $\mathcal{SC}$ reaches only configurations of $\mathcal{SC}$.

2.9 Balls and Views

In this article we use the concepts of $Ball$ and $View$ inspired by those presented in [BDDT98].

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 use sometimes the term of $Ball$ in a configuration C instead of view $\mathcal{V}_p^1(c)$, also noted $C_{|p \cup \mathcal{N}(p)}$.

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.

Chapter 3

Necessary and Sufficient Conditions for 1-adaptivity

We consider here 1-adaptive algorithms such that $\mathcal{SC} = \mathcal{L}$. As soon as the system reaches a legitimate configuration, it gains the ability to correct a memory corruption in just one transition. The 1-adaptive algorithms have, under this assumption, the capacity to reach a legitimate configuration in only one transition from an illegitimate configuration at a distance 1 from a legitimate configuration.

Thus, once stabilization is reached and if the faults are separated in space, the only enabled processors are processors that can correct the system state and there will not be inopportune propagation of erroneous values and incorrect behaviors due to faults. This property makes the self-stabilizing systems much safer.

We study here the feasibility of such algorithms. For that, we establish some necessary and sufficient conditions for a self-stabilizing algorithm to be 1-adaptive.

For the sake of generality, we choose as the execution model the distributed demon [Dol00].

First we will consider strictly 1-local algorithms designed for specific network topologies. The hypothesis of 1-locality is easy to understand: if the algorithm is not 1-local, from an illegitimate configuration such that every view at distance 1 is correct, it is almost not possible to reach a legitimate configuration in one step. In a second step we will consider any topology but we will restrict our study to algorithms whose legitimate configurations are not too close one to the other, in order to be able to determine with certainty which is the nearest legitimate configuration.

3.1 Networks without triangle

The networks we consider in this section have no triangles. We make this assumptions for technical reasons, but it should be noticed that several common topologies have this property (rings, trees, grid, hypercubes, etc.). The corruption of a single processor in a stable legitimate configuration of a self-stabilizing algorithm may put the system into an illegitimate configuration where potentially several processors can be activated. Then for being an 1-adaptive algorithm any combination of rules must bring the system back into a legitimate state.

Definition 3.1.1 *Let A be a strictly 1-local self-stabilizing algorithm executed on a communication graph G without triangle, with 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 3.1.1 *Condition 1 $\wedge$ Condition 2 is a necessary and sufficient condition for an algorithm 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 $C_{|p} \neq C'_{|p}$.

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*.

Let us prove now *1-adaptive* $\Rightarrow Condition1 \wedge Condition2$ and thus $\neg Condition1 \vee \neg Condition2 \Rightarrow \neg$*1-adaptive*. First, if $Condition1$ is false, then there is at least one $B = Ball(p, 1)$ enabled in C' with a correct associated view $\mathcal{V}^1_p(C')$.

Moreover as C' is illegitimate and the algorithm is strictly 1-local then there are two neighbors processors 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 $p_k \in \mathcal{N}(p)$) then $p_k \notin \mathcal{N}(p)$ (respectively $p_j \notin \mathcal{N}(p)$).

Thus the system may only activate B because we are under the distributed demon. It comes that in a configuration C'' where at least $\mathcal{V}^1_{p_j}(C'')$ or $\mathcal{V}^1_{p_k}(C'')$ is not correct. Because if only B is activated in C' then from what precedes either $p_j \notin \mathcal{N}(p)$ or $p_k \notin \mathcal{N}(p)$ and thus 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 because A is strictly 1-local C'' is illegitimate and then the algorithm is not 1-adaptive. Thus, we have that $\neg Condition1 \Rightarrow \neg$*1-adaptive*.

Now if $Condition1$ is true and $Condition2$ is false, then there is 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 show that to study the property of 1-adaptivity of strictly 1-local self-stabilizing algorithms, it is unnecessary to study all the executions of the system (but just the possible transitions of the faulty processor and its neighbors in the configurations at distance 1 from a legitimate configuration, becauseonly these processors can correct the error).

We can note that the hypothesis of the distributed demon makes the $Condition2$ difficult to obtain. Indeed the condition implies that all the enabled processors of $Ball(p_i, 1)$ must, if they are activated, correct the error. Then, if several processors are enabled at the same time, their activations must be compatible.

Thus for an algorithm being 1-adaptive under the distributed demon, it is necessary for a processor neighbor of p_i to be enabled if and only if :

- Its activation corrects the error.

- This activation is compatible with the activation of any of its enabled neighbors.

We can derive a useful property from proposition 3.1.1.

Proposition 3.1.2 *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 exist 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 3.1.1, $\mathcal{V}^1_p(C^i)$ 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 which verifies proposition 3.1.2 for all its legitimate configurations is silent.*

Proof : It is obvious that if all the legitimate configurations of an algorithm verify 3.1.2 then all of its legitimate configurations are silent and by definition the algorithm itself is silent. □

For a large majority of self-stabilizing algorithms, the corruption of any of its processors brings the system in an illegitimate configuration. Then for the strictly 1-local algorithms, 1-adaptivity is strongly related to the property to be silent.

3.2 Example : 1-adaptive algorithm for a network without triangles

We present in this section a very simple 1-adaptive self-stabilizing coloration algorithm. A distributed system is in a configuration that satisfies the coloration problem if and only if all the processors are colored and every processor has a color different from its neighbors. This algorithm colors a chain of three processors. These three processors are named and thus distinguishable. We name them p_1, p_2 and p_3 where p_1 and p_3 are both neighbors of p_2.

The system is in a legitimate configuration if and only if it satisfies the coloration specification.

Proposition 3.2.1 *Algorithm 3.2 is self-stabilizing for the coloration problem.*

Proof :

Algorithm 1 Coloration algorithm for a chain of three processors

p_1**'s variable :**	p_2**'s variable :**	p_3**'s variable :**
Color : integer, Color = 0 ;	**Color** : integer, Color$\in$ { 1,2};	**Color** : integer, Color$\in$ { 2,3};
p_1**'s action :**	p_2**'s action :**	p_3**'s action :**
TRUE $\rightarrow \emptyset$;	Color = p_3.Color $\rightarrow$ Color := 1 ;	Color = p_2.Color $\rightarrow$ Color := 3 ;

Closure Let us suppose that the system is in a legitimate configuration , then all the processors have a color different from each other. Thus p_2 evaluates $Color = p_3.Color$ to false and p_3 evaluates $Color = p_2.Color$ also to false. As p_1 cannot change its color, we obtain that no processor can execute an action and change its color. The system cannot make any transition and thus it remains in a legitimate configuration. The set of the legitimate configurations is closed.

Convergence The only illegitimate configuration of the system is the configuration where p_2 and p_3 have the same color. In fact p_1 can only have the color 1 which is necessarily different from that of its neighbors p_2 since p_2 can not get the color 1.

The only color that p_2 and p_3 can have in common is the color 2. Thus the only illegitimate configuration is $C_i = \{p_1.Color = 0, p_2.Color = 2, p_3.Color = 2\}$. In this case both p_2 and p_3 can execute their action. Since the execution of the action of p_2 or p_3 gives to this processor a color that the other cannot have the system necessarily get into a legitimate configuration in any transition starting from C_i. □

Proposition 3.2.2 *This algorithm is 1-adaptive.*

Proof : Algorithm 3.2 is strictly 1-local and is executed on a chain which by definition contains no triangle. Thus it suffices to check $Conditions 1 and 2$ from proposition 3.1.1.

$Condition 1$ is actually verified, since if a processor has a correct view then its neighbors have a different color and thus if p_2 has a correct view at distance 1 then it evaluates $Color = p_3.Color$ to false. If p_3 has a correct view at distance 1 then it evaluates $Color = p_2.Color$ to false. As p_1 cannot change its color, we obtain that this algorithm verifies $Condition 1$.

Let us check now that $Condition 2$ is also satisfied. There is only one illegitimate configuration, the configuration C_i. This configuration is 1 distant of at least one legitimate configuration since we just have to change either the color of p_2 or the color of p_3 to reach a legitimate configuration.

In this configuration, two balls are enabled, $B_1 = Ball(p_2, 1)$ and $B_2 = Ball(p_3, 1)$. There are three possible transitions. In the first only p_2 executes its action. In this case p_2 takes color 1 and the system gets into the configuration $C_1 = \{p_1.Color = 0, p_2.Color = 1, p_3.Color = 2\}$, where all the balls are correct and thus $Condition 2$ is satisfied. With the transition where only p_3 executes its action, the system reaches $C_2 = \{p_1.Color = 0, p_2.Color = 2, p_3.Color = 3\}$. Finally if p_2 and p_3 are activated at the same time then the system reaches the configuration $C_3 = \{p_1.Color = 0, p_2.Color = 1, p_3.Color = 3\}$ and this transition also satisfies $Condition 2$. Thus algorithm 3.2 is 1-adaptive. □

3.3 Restriction on the legitimate configuration density

Corollary 1 confirms the impossibility result of [GT02] and points out the connexion between 1-adaptivity and the property to be silent. Let us remind that in [GT02] they prove that any dynamic global algortihm that tolerates at least one memory corruption requires at least $\Omega(D)$ transitions to converge, where D is

the diameter of the system. We study now the conditions for silent algorithms to be 1-adaptive. We assume that self-stabilizing algorithms we consider here have legitimate configurations at distance at least 3 from each other. This hypothesis is verified by many algorithm solving problems with only one legitimate configuration, such that the computation of the network size, the topology learning, the leader election, etc.

Definition 3.3.1 *Let A be a silent self-stabilizing algorithm and $\mathcal{L}$ its legitimate configurations, such that for all $(l, l') \in \mathcal{L}^2 Dist(l, l') \geq 3$.*

Then 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 3.3.1 *Condition 1 $\wedge$ Condition 2 is a necessary and sufficient conditions for A to be 1-adaptive.*

Proof: First we prove the implication $Condition1 \wedge Condition2 \Rightarrow 1 - adaptive$.

In C' all the views $\mathcal{V}_p^1(C')$ such that $p \in \mathcal{P} \setminus p_i \cup \mathcal{N}(p_i)$ are correct since $\mathcal{V}_p^1(C') = \mathcal{V}_p^1(C)$ and C is legitimate. Because A is silent any processor with 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 $Ball(p_i, 1)$ are potentially enabled. According to $Condition1$ the only enabled ball in C' is $Ball(p_i, 1)$. Moreover according to $Condition2$, the activation of this ball puts p_i in the same state as in C. We can conclude that 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: $1 - adaptive \Rightarrow Condition1 \wedge Condition2$. For that, as for the preceding conditions, we will prove that $\neg Condition1 \vee \neg Condition2 \Rightarrow \neg 1 - adaptive$. Let us suppose that $Condition1$ is false. As previously, in C' all the views $\mathcal{V}_p^1(C')$ such that $p \in \mathcal{P} \setminus p_i \cup \mathcal{N}(p_i)$ are correct since $\mathcal{V}_p^1(C') = \mathcal{V}_p^1(C)$ and C is legitimate. Thus $\neg Condition1$ implies that $\exists p_j \neq p_i$, $p_j \in \mathcal{N}(p_i)$ such that $\mathcal{V}_{p_j}^1(C')$ 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 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 $\neg Condition1 \Rightarrow \neg 1 - adaptive$.

Let us consider now that $Condition1$ is true and $Condition2$ 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 $Condition1 \wedge Condition2\neg \Rightarrow \neg 1 - adaptive$. We conclude that $\neg Condition1 \vee \neg Condition2 \Rightarrow \neg 1 - adaptive$ and that by consequence $1 - adaptive \Rightarrow Condition1 \wedge Condition2$. □

The results of this section point out the fact that, if the legitimate configurations (of a silent self-stabilizing algorithm) are at distance at least 3, then the only way to be 1-adaptive is to return after a single corruption to the previous legitimate configuration. That yields a general methodology for designing 1-adaptive algorithms.

- Increase the memory space of the processor for moving away the legitimate configurations.
- Manage for the corrupted processor to be the only one enabled.

3.4 Example : 1-adaptive naming algorithm with N-Distant legitimate configurations

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 its legitimate configurations at distance N from each other. It is well known that naming can not be achieved in a deterministic way. Thus the only known solutions are probabilistic. That is the case for the algorithm we present, which is also 1-adaptive.

This algorithm works on complete graphs. We assume that each processor has previously numbered its registers 1,2,..,$N-1$. We note $Reg[i]$ the value of the i^{th} register. If p has a neighbor q, corresponding to its i^{th} register , then p can read the number that q has for the register corresponding to p with the function $GetOrder(Reg[i])$. This function returns j if p corresponds to the j^{th} register of q. The numbering of registers can not be corrupted.

3.4.1 Algorithm description

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 configuration.

Finally Reg is an array and it contains at the index i the shared variable values of the neighbor corresponding to the i^{th} register. Reg is completely updated before the evaluation of the guarded actions.

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 [AD97]. 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 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 $Name$ variable 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 $Names$ 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 with the $Names$ and $Snapshot$ variables of the correct processors.

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 $random(\{0, ..., N\text{-}1\} \setminus \{Reg[i].Name | i \in \{1...N\text{-}1\} \wedge \forall k \in \{1...N\text{-}1\}, Reg[i].Name \neq Reg[k].Name\})$;
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 consistent with each other and these $Snapshots$ represent a legitimate configuration where $(p.Name, p.Names) \neq Reg[i].Snapshot[GetOrder(Reg[i])]$ for all i $\in \{0...N\text{-}1\}$ then **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 : $\forall (i, j) \in \{1...N\text{-}1\}^2$, $Reg[i].Name \neq Reg[j].Name \wedge \exists k \in \{1...N\text{-}1\}$, $Name = Reg[k].Name$
$\wedge \neg (N\text{-}1) ConsistentSnapshots \wedge Consensus = 0$
$\rightarrow GetNewName$;
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 : $\forall (i, j) \in \{1...N\text{-}1\}^2$, $Reg[i].Name \neq Reg[j].Name \wedge \exists k \in \{1...N\text{-}1\}$, $Name = Reg[k].Name$
$\wedge Consensus \neq 0$
$\rightarrow Name = Consensus.Name$, $Names = Consensus.Names$, $TakeSnapshot$;

3.5 Algorithm Analysis

Definition 3.5.1 *A legitimate configuration for algorithm 3.4.1 is such that for every pair of processors (p, q) where q corresponds to the i^{th} register of p, $p.Name \neq q.Name$, $p.Names[i] = q.Name$ and $p.Snapshots[i] = (q.Name, q.Names)$.*

Proposition 3.5.1 *The legitimate configurations of algorithm 3.4.1 are N-Distant.*

Proof : In any two different legitimate configurations, noted L_1 and L_2, there are at least two different processors, noted p and q, such that $L_{1|\{p\}}.Name \neq L_{2|\{p\}}.Name$ and $L_{1|\{q\}}.Name \neq L_{2|\{q\}}.Name$. In fact if every processor has the same $Name$ then $L_1 = L_2$, and if one processor p has a different $Name$ in L_1 and L_2 then has there are only N possible $Name$ for the N different processor there is at least the processor q that had the $Name$, $L_{2|\{p\}}.Name$ in L_1 which necessary has a new $Name$ in L_2.

Then every processor r different from p and q, for which p is the i^{th} neighbor and q the j^{th} neighbor, is such that $L_{1|\{r\}}.Names[i] \neq L_{2|\{r\}}.Names[i]$ since $L_{1|\{r\}}.Names[i] = L_{1|\{p\}}.Name$, $L_{2|\{r\}}.Names[i] = L_{2|\{p\}}.Name$ and $L_{1|\{p\}}.Name \neq L_{2|\{p\}}$.Name. Thus we have $L_{1|\{r\}} \neq L_{2|\{r\}}$.

Since $L_{1|\{p\}}.Name \neq L_{2|\{p\}}.Name$ and $L_{1|q}.Name \neq L_{2|q}.Name$, we have $L_{1|\{p\}} \neq L_{2|\{p\}}$ and $L_{1|q} \neq L_{2|q}$. Then we can conclude that every processor has a different state in L_1 and L_2. Thus L_1 and L_2 are at distance N from each other.

The legitimate configurations of this algorithm are N-Distant.

□

Definition 3.5.2 *We say that a processors p in a configuration C of a distributed system S that executes algorithm 3.4.1 is **misnamed** (or wrongly named) if and only if there is a processor q in S such that $C_{|p}.Name = C_{|q}.Name$. We also say that p and q are homonyms.*

Definition 3.5.3 *Let $P(k)$ be a predicate over the configurations of a system S executing algorithm 3.4.1. A configuration C of S satisfies $P(k)$ if and only if there is exactely k different misnamed processors in C.*

First we prove that there is no possible way to misnammed a processor. Thus the number of misnammed processor in the system can only decrease.

Proposition 3.5.2 *Let S be a system that executes algorithm 3.4.1. There is no transition $T = (C, t, \text{rand}, C')$ of S such that $C \vdash P(k_1), C' \vdash P(k_2)$ and $k_2 > k_1$.*

Proof : In any configuration C of the system, the processors that can change their $Name$ are the processors that can execute A1, A2 or A5. Those processors necessarily have at least one neighbor with the same $Name$, otherwise they evaluate the guards of these rules to false. However none of these actions can give a $Name$ that a correctly named processor has before the transition. In fact A1 and A2 use the method GetNewName that picks up uniformly a random $Name$ in the set $(\{0, ..., N\text{-}1\} \setminus \{Reg[i].Name | i \in \{1...N\text{-}1\} \wedge \forall k \in \{1...N\text{-}1\}, Reg[i].Name \neq Reg[k].Name\})$. Thus those processors cannot be misnamed after the execution of $A1$ or $A2$.

Moreover A5 can not give the $Name$ that a processor had before the transition otherwise the method consensus would return 0. Thus it can have the same $Name$ as a processor that has been activated during the same transition. In this case this processor was already wrongly named. □

Proposition 3.5.3 *Every cone $Cone_h$ of every strategy st of a system S that executes algorithm 3.4.1 under the distributed demon, such that $last(h)$ satisfies $P(k)$, for a $k \geq 3$, satisfies $LC(P(k), P(k-1), \delta_k, 1)$, where $\delta_k > 0$.*

Proof : Let st be a strategy of algorithm 3.4.1 under the distributed demon starting in a configuration C and $Cone_h$ be a cone of st, where $C' = last(h)$ satisfies $P(k)$ for a $k \geq 3$. We have by definition that $Pr(Cone_h) > 0$ since $Cone_h$ is a cone of st.

From proposition 3.5.2 we know that every configuration reachable from C' in st verifies $P(k')$ for $k' \leq k$. Moreover the only action that can be executed in C' is action A1. Thus in every possible transition of st starting in C' there is a non empty subset of misnamed processors of the system, noted $\mathcal{WN}$, that are activated to execute their action A1.

Then the probability, for the system to reach a configuration C'' where there is at least 1 less misnammed processor, is equal to the probability of a transition where one of the misnamed processors get a correct $Name$. This means that this processor is activated and it picks up a $Name$ different from all the $Name$ of the not activated processors. This $Name$ also has to be different from the new $Name$ chosen by the other activated processor.

Let $Cone_{h'}$ be a cone of st where $h' = [hf], |h'| = |h| + 1, last(h') = C''$ and $last(h')$ satisfies $P(k-1)$ and $T = (C', t, \text{rand}, C'')$ be a transition in h' as described above. Let m be the number of different $Name$ of not correctly named processors that are not activated in T, $(k/2) \geq m \geq 0$, since every misnammed processor has at least one homonym.

The probability associated to T is :

$$\begin{aligned} Pr_1 &= \text{probability to pick up a } Name \text{ that none of the not activated processor has} \\ &= (k-m)/k \\ Pr_2 &= \text{probability that every other processors picks up a } Name \text{ different from the processor} \\ &\quad \text{that get a correct Name} \\ &= ((k-(m+1))/k)^{|\mathcal{WN}-1|} \\ Pr_3 &= Pr_1 * Pr_2 \\ Pr_3 &= (k-m)/(k) * ((k-(m+1))/k)^{|\mathcal{WN}-1|} \end{aligned}$$

We get that $Pr_3 > 0$ since $k \geq 2m$. Thus we have that from any configuration C of the system, where there is at more than two misnamed processors. There is at least one transition from C that leads in a configuration where there is one less misnamed processor that is associated positive probability. Then we obtaine that the union of the sub-cones of $Cone_h$ that reache a configuration with one less misnamed processor is superior or equal to the probability calculated above. Formaly $Pr(\bigcup_{Cone_{h'}, \in \mathcal{F}, |h'|=|h|+1} Cone_{h'}) \geq Pr(Cone_{h'}) = Pr(Cone_h) * (k-m)/(k) * ((k-(m+1))/k)^{|\mathcal{WN}|} = \delta_k > 0$. □

Proposition 3.5.4 *Every strategy st of a system S that executes algorithm 3.4.1 under the distributed demon verifies $Pr_{st}(\mathcal{E}_{P(2)}) = 1$.*

Proof : Let st be a strategy under the distributed demon rooted in C. If $C \vdash P(k), k \leq 2$ then the property is trivially verified.

If $C \vdash P(k), k \geq 2$ then we know from proposition 3.5.3 and 3.5.3, and theorem 2.7.1 that $P_{st}(\mathcal{E}_{P(k-1)}) = 1$. By induction on k we get that $P_{st}(\mathcal{E}_{P(2)}) = 1$. □

Proposition 3.5.5 *In any strategy st of a system S that executes algorithm 3.4.1 under the distributed demon, every cone $Cone_h$ such that $last(h) = C_i, C_i \notin \mathcal{L}$ and $C_i \vdash P(2)$, and for every legitimate configuration L, $Dist(L, C_i) \geq 2$. Then either the system reaches in one transition a configuration where every processor has an unique $Name$ or $Cone_h$ verifies $LC(P(2), P(0), \delta, 1)$ for $\delta > 0$.*

Proof : Let $Cone_h$ be a cone of a strategy st of a system executing algorithm 3.4.1 under the distributed demon. Let $C_i = last(h)$ be an illegitimate configuration such that $C_i \vdash P(2)$, so there exists two processors in C_i with the same $Name$, namely p and q, and C_i is also such that for every legitimate configuration $L \in \mathcal{L}$, $Dist(L, C_i) \geq 2$.

In C_i no processor different from p and q can execute an action. In fact they cannot execute A2 or A5 as they are correctly named. They can not execute A1, A3 or A4 as there is exactly two processors with the same $Name$ in the network. Thus the only enabled processors in C_i are p and q.

For the same reason, p and q can not execute A1,A3 or A4. Then they can just execute A2 or A5. They can not both evaluate the guard of A5 to true since that would mean that they both evaluate Consensus to true. And so they have a different state from the one in the $Snapshot$ of their neighbors at the index that correspond to them (which are different by definition). But Consensus is evaluated to 0 for a processor if there is a $Snapshot$ of one of its neighbor which contains more than one state that does not match the current state of a processor. Here this is the case for every processor different from p and q. Thus there is at most one processor in C_i that evaluates the guard of A5 to true.

Suppose that p evaluates A5 to true. The first possibility is that in the transition starting from C_i, just p is activated. Then the system reaches in one transition a configuration where every processor has a unique $Name$, since Consensus can not return a value with a $Name$ already attributed.

Otherwise the system takes a step where q is activated. If q is activated, then there is two possibilities : p is also activated or just q is activated.

Let $Cone_{h'}$ be a sub-cone of $Cone_h$ in st, such that $h' = [hf], last(h') = C'', |h| + 1 = |h'|$ and for every pair (p_i, p_j) of $\mathcal{P}^2$, $C''_{|\{p_i\}}.Name \neq C''_{|\{p_j\}}.Name$.

Let $Pr(Cone_h)$ be the probability associated to $Cone_h$.

If just q is activated in $last(h)$, then $Cone_{h'}$ exists since it corresponds to the transition of st from $last(h)$ where q picks up the last $Name$ that no processor has. Since q uniformly randomly chose a $Name$ in the set of $Name$ corresponding to its $Name$ in $last(h)$ and the $Name \in \{0...N-1\}$ that no processor has in $last(h)$ (by definition of GetNewName). We get that :

$Pr(Cone_{h'}) = 1/2 * (Pr(Cone_h))$

If p and q are both activated in $last(h)$. Then the cone $Cone_{h'}$ exists since it corresponds to the transition of st from $last(h)$ where p executes A5 and changes its $Name$ and q picks up the $Name$ it already had in $last(h)$. Since q uniformly randomly chose a $Name$ in a set of two values as described above. We get that :

$Pr(Cone_{h'}) = 1/2 * (Pr(Cone_h))$

Suppose now that no processor can execute A5 in C_i. Then p and q are enabled as they evaluate the guard of A2 to true.

If just one processor is activated then $Cone'_h$ exists and the probability associated to it in st is :

$Pr(Cone_{h'}) = 1/2 * (Pr(Cone_h))$.

As it has just been proven.

Otherwise p and q are activated in C_i. Then there are four possible toss. They both get a new $Name$, they both keep their former $Name$ or one of the processor changes its $Name$ and the other don't. Thus the probability associated to $Cone'_h$ is :

$Pr(Cone_{h'}) = 1/4 * (Pr(Cone_h))$

However as there is two such $Cone'_h$ the probability associated to all the sub-cones of $Cone_h$ that satisfies $P(0)$ is $1/2 * (Pr(Cone_h))$.

We can conclude that either the system reaches in one transition from C_i a configuration where every processor has a unique $Name$ or $Cone_h$ verifies $LC(P(2), P(0), \delta, 1)$ for $\delta = 1/4 * (Pr(Cone_h)) > 0$.
□

Proposition 3.5.6 *Any transition of a system executing algorithm 3.4.1 under the distributed demon that starts in an illegitimate configuration C' such that $C' \vdash P(2)$ and there exists $C \in \mathcal{L}$, $Dist(C, C') = 1$ brings the system into a legitimate state, namely C.*

Proof : Suppose two configurations $C \in \mathcal{L}$ and $C' \notin \mathcal{L}$ such that $C_{|\{p_i\}} \neq C'_{|\{p_i\}}$, $C_{|\{p_i\}}.Name \neq C'_{|\{p_i\}}.Name$ and $C'_{|\{p_i\}}.Name = C'_{|\{p_j\}}.Name$. Then for the same reasons as the one mentioned above. No processor of the system different from p_i and p_j is enabled in C'.

Since C' is at distance 1 from C, all the $Snapshot$ of the processors different from p_i are coherent and all contains the current state of the processors different from p_i in the correct order (in regard to their local register order). Thus every processor different from p_i evaluates (N-1)ConsistentSnapshots to true. Then p_j cannot execute A2. It can not execute A1, A3 or A4 as there are exactly two processors with the same $Name$ in C'. It can not execute A5 since it evaluates (N-1)ConsistentSnapshots to true, which means that all its neighbors different from p_i have its current state in their $Snapshot$ at the right place and thus Consensus returns 0.

The only enabled processor in C' is p_i. p_i evaluates Consensus to $(C_{|\{p_i\}}.Name, C_{|\{p_i\}}.Names)$ since every $Snapshot$ of its neighbors contains this value at the index concerning p_i in C and thus in C'. Then it can not execute any other actions and the only possible transition of the system starting from C is $C' \xrightarrow{p_i} C$.
□

Proposition 3.5.7 *Let st be a strategy of a system executing algorithm 3.4.1 under the distributed demon. Then every $Cone_h$ of st such that $last(h) \vdash P(0)$ only contains executions that reach a legitimate configuration.*

Proof : Let $Cone_h$ be a cone of a strategy st of a system executing algorithm 3.4.1 under the distributed demon such that $last(h) = C$ and $C \vdash P(0)$.

Then by hypothesis and by proposition 3.5.2 we have that, in any configuration reachable from C, $P(0)$ is satisfied and no processor can execute A1, A2 or A5 in C. Since there are no two processors with the same $Name$. Thus the only enabled actions are A3 and A4 and no processor can change its variable $Name$ any more. Then the processors just have to fill in correctly their variable $Names$ and $Snapshot$ in order to reach a legitimate configuration. Let us prove now that evry processor will first get a correct value for its variable $Names$. And then a correct value for its variable $Snapshot$ and thus the system necessarily will get in a legitimate configuration.

Let $\mathcal{WN}$ be the set of processors that have a wrong variable $Names$. That is, a variable $Names$ that does not contain the $Name$ of each of its neighbors in the order of its registers. Let wn be the number of processors in $\mathcal{WN}$ in C. Let $\mathcal{WS}$ be the set of processors that have a wrong variable $Snapshot$. That is, a variable $Snapshot$ that does not contain the tuple $(Name, Names)$ of each of its neighbors in the order of its register in the current state of the system.

Every processor in $\mathcal{WN}$ evaluates the guard of A3 to true. They evaluate the guard of A4 to false since they have a wrong variable $Names$. Every processors in $\mathcal{WN} \cap \mathcal{WS}$ only have one action executable, namely A3. Since the processors of $\mathcal{WN}$ can not execute A4.

Every processor of $\mathcal{WN}$ chosen by the demon to execute A3 will get a correct variable $Names$ for the remaining of the execution. Every processor from $\mathcal{WS}$ activeted by the demon is not enabled until a processor of $\mathcal{WN}$ is activeted, since after the execution of A4 a processor evaluates $ConsistentSnapshot$ to true until a processor changes either its variable $Name$ or its variable $Names$.

Then every $|\mathcal{WS}| \leq N$ transitions at least one processor of $\mathcal{WN}$ is activated and we have $|\mathcal{WN}| = wn - 1$. In fact even if in every transition of the execution the demon choses only processors in $\mathcal{WS}$ it will reach in a number of transition lower or equal to $|\mathcal{WS}|$ a configuration where, $\mathcal{WS} = \emptyset$, and where the only enabled processors are the processors of $\mathcal{WN}$. It will then activate one of those processors and the set $\mathcal{WS}$ will contain all the processors of the system.

But then again in at least $|\mathcal{WS}| \leq N$ transitions we will have $|\mathcal{WN}| = wn - 2$. By induction on wn we get that the system reaches from C a configuration where $\mathcal{WN} = \emptyset$ in a number of transitions lower or equal to $wn * N$.

Then the system only reaches configurations where every processor is correctly named and has a correct variable $Names$. In fact the only enabled processors are the processors of $\mathcal{WS}$. Since in every transition of the system there is at least one processor activated then they will all be activated in a number of transition lower or equal to N. Then every $Snapshot$ will be updated correctly and the system will reach a legitimate configuration.

Finally we have that every sub-cone of $Cone_h$ reaches a legitimate configuration from C in at most $N * (wn + 1)$ transitions. □

Proposition 3.5.8 *Algorithm 3.4.1 is 1-adaptive.*

Proof : Let C and C' be two configurations of a system S executing algorithm 3.4.1 such that $C \in \mathcal{L}$, $C' \notin \mathcal{L}$, $Dist(C, C') = 1$,$C_{|p_i} \neq C'_{|p_i}$.

Then if there is a processor $p_j \neq p_i$ in S such that $C'_{|p_j}.Name = C'_{|p_i}.Name$ then by proposition 3.5.6 we know that the only possible transition from C' is $C' \xrightarrow{p_i} C$.

Let suppose now that $C_{|p_i}.Name = C'_{|p_i}.Name$ and $C_{|p_i}.Names \neq C'_{|p_i}.Names$. No processor p_j different from p_i is enabled. In fact as there are no two processors with the same $Name$, no processor can execute A1, A2 or A5. Since for every $p_j \in \mathcal{P}, p_j \neq p_i$, $C_{|p_j} = C'_{|p_j}$ and $C_{|p_i}.Name = C'_{|p_i}.Name$, they all evaluate the guard of A3 to false since they have $C'_{|p_j}.Names[i] = C_{|p_i}.Name \neq C'_{|p_i}.Name$ for the index i corresponding to p_i and (N-1)ConsistentSnapshots to true, since every processor but one is in the same state as in C which is a legitimate configuration. Then they can not execute A4 neither.

Thus the only enabled processor in C' is p_i. Since $C_{|p_i}.Names \neq C'_{|p_i}.Names$ and every processor has the same $Name$ in C and C', p_i evaluates the guard of A4 to true. By the application of this rule p_i regains a correct $Names$ and a correct $Snapshot$ that contains the current $Name$ and state of the processors which are the same as in C, since every processor verifies $Name.C' = Name.C$. Thus the only possible transition starting from C' is $C' \xrightarrow{p_i} C$.

Finally we suppose that just the $Snapshot$ of p_i is corrupted. Then as previously, the only enabled processor is p_i and the only executable action for p_i is A4. It can not execute A3 any more since its variable $Names$ is correct. Thus the only executable action for p_i is A4. Since all the other processors have the same state as in C, p_i regains with A4 the state that it had in C. Then again the only possible transition from C' is $C' \xrightarrow{p_i} C$.

We can conclude that in any configuration C' as described above the only enabled processor is p_i and the only possible transition of the system is $C' \xrightarrow{p_i} C$. Thus by proposition 3.3.1 we have that algorithm 3.4.1 is 1-adaptive. □

Proposition 3.5.9 *Algorithm 3.4.1 is silent.*

Proof : Suppose that a system that executes algorithm 3.4.1 is in a legitimate configuration $C \in \mathcal{L}$. Then no processor can execute actions A1, A2 nor A5 as there is no two processors $(p, q) \in \mathcal{P}^2$ such that $C_p.Name = C_q.Name$. No processor can execute A3 or A5, since by the definition 3.5.1 we know that in C, $p.Names[i] = q.Name$ and $p.Snapshots[i] = (q.Name, q.Names)$. Thus when a processor evaluates the guard of A3 and A4 its evaluates $\wedge\exists k \in \{1...N\text{-}1\}, Names[k] \neq Reg[k].Name$ to false and $\forall i \in \{0...N\text{-}1\}, Snapshot[i] = (Reg[i].Name, Reg[i].Names)$ to true and thus ConsistentSnapshot to false. Finally, no processor is enabled in C then C is terminal and the algorithm is silent.□

Proposition 3.5.10 *Algorithm 3.4.1 is self-stabilizing for the predicate of definition 3.5.1.*

Proof : From proposition 3.5.4, we have that in any strategy st of a system that executes algorithm 3.4.1 under the distributed demon, $Pr_{st}(\mathcal{E}_{P(2)}) = 1$.

Then we get that $P(2)$ is a probabilist attractor for TRUE, $TRUE \rhd_{prob} P(2)$.

From proposition 3.5.5 and the theorem 2.7.1 of local convergence we have that $P(2) \rhd_{prob} P(0)$. Thus we have $TRUE \rhd_{prob} P(2) \rhd_{prob} P(0)$.

Finally from proposition 3.5.7 and $TRUE \rhd_{prob} P(2) \rhd_{prob} P(0)$ we have that algorithm 3.4.1 is self-stabilizing for the predicate of definition 3.5.1 and thus for the naming problem. □

Proposition 3.5.11 *Algorithm 3.4.1 is a self-stabilizing 1-adaptive silent probabilistic algorithm for the predicate of definition 3.5.1.*

proof From propositions 3.5.11 3.5.9 and 3.5.8 we have that algorithm 3.4.1 is a self-stabilizing 1-adaptive silent probabilistic algorithm for the predicate of definition 3.5.1.

Chapter 4

Conclusion

We have introduced here the new concept of 1-adaptivity for self-stabilizing systems. This concept is powerful and interesting since it makes possible to guarantee an almost instantaneous correction of memory corruptions hitting only one processor of the system. It ensures that the fault is not propagated and thus that the non faulty processors are not affected by the corruption of their neighbors. 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 two neighbors are not simultaneously corrupted. If it is not the case, the general stabilizing mechanisms allow anyway the system to recover (unfortunately in more than one step). Secondly the results can be extended in a straightforward way to the case where at most k (for a fixed integer k) corruptions hit the system. Analogous sufficient and necessary conditions can be obtained. They can also be decided locally.

In conclusion we would like to emphasize the fact that, more than a way for verifying that existing self-stabilizing algorithms are not 1-adaptive (everybody knew that before), the conditions we presented give a systematical way to design 1-adaptive algorithms. The idea is to enlarge the state of the processes with variables whose unique role is to realize the necessary and sufficient condition. We presented in the example of naming how this idea could be exploited by hand (rule A5), but our approach leads to a more systematic method. The goal here is to produce an automatic transformer having as an input a self-stabilizing algorithm and producing as an output an equivalent self-stabilizing algorithm, having the supplementary property to be 1-adaptive (or more generally k-adaptive for any given k). One of the most often cited drawback of self-stabilization is the loss of the safety during the stabilization phase, that makes self-stabilization inadequate for critical tasks. Our work can be seen as an attempt to overcome this drawback.

Bibliography

[AD97] Y. Afek and S. Dolev. Local stabilizer. In *Israel Symposium on Theory of Computing Systems*, pages 74–84, 1997.

[AKP03] 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.

[BDDT98] 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.

[BGK99] 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*, May 1999.

[Dij74] E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, 2000. Ben-Gurion University of the Negev, Israel.

[GGHP96] S. Ghosh, A. Gupta, T. Herman, and S.V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Symposium on Principles of Distributed Computing*, pages 45–54, 1996.

[Gra00] M. Gradinariu. *Modélisation vérification et raffinement des algorithmes auto-stabilisants*. PhD thesis, Université Paris XI, Orsay, 2000.

[GT02] 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.

[Her97] T. Herman. Observations on time adaptive self-stabilization, 1997.

[KP95] S. Kutten and D. Peleg. Fault-local distributed mending. In *Proceedings of the 14th Annual ACM Symposium on Principels of Distributed Computing (PODC'95)*, August 1995.

[KP97] S. Kutten and B. Patt-Shamir. Time-adaptive self stabilization. In *Symposium on Principles of Distributed Computing*, pages 149–158, 1997.

[KP98] S. Kutten and B. Patt-Shamir. Asynchronous time-adaptive self stabilization. In *Symposium on Principles of Distributed Computing*, page 319, 1998.

[KP01] S. Kutten and B. Patt-Shamir. Adaptive stabilization of reactive distributed protocols, 2001.

[NA02] M. Nesterenko and A. Arora. Local tolerance to unbounded byzantine faults, 2002.

Chapter 5

Annexe

(N-1)SnapshotsCoherents :

/* First we check if two processors have two index of their $Snapshot$ that do not correspond to the current state of a processor */

$$
\begin{array}{l}
\neg \quad (\exists i \in \{1...N\text{-}1\} \\
\qquad \exists k \in \{1...N\text{-}1\} \\
\qquad\quad (Reg[i].Snapshot[k] \neq (Name, Names) \\
\qquad\quad \wedge \forall m \in \{1...N\text{-}1\} Reg[i].Snapshot[k] \neq (Reg[m].Name, Reg[m].Names) \\
\qquad\quad \wedge \exists l \in \{1...N\text{-}1\} \\
\qquad\qquad l \neq k \\
\qquad\qquad \wedge Reg[i].Snapshot[l] \neq (Name, Names) \\
\qquad\qquad \wedge \forall m \in \{1...N\text{-}1\} Reg[i].Snapshot[l] \neq (Reg[m].Name, Reg[m].Names) \\
\qquad\quad) \\
\qquad \wedge \exists j \in \{1...N\text{-}1\} \\
\qquad\quad (j \neq i \\
\qquad\quad \wedge \exists k \in \{1...N\text{-}1\} \\
\qquad\qquad Reg[j].Snapshot[k] \neq (Name, Names) \\
\qquad\qquad \wedge \forall m \in \{1...N\text{-}1\} Reg[j].Snapshot[k] \neq (Reg[m].Name, Reg[m].Names) \\
\qquad\qquad \wedge \exists l \in \{1...N\text{-}1\} \\
\qquad\qquad\quad l \neq k \\
\qquad\qquad\quad \wedge Reg[j].Snapshot[l] \neq (Name, Names) \\
\qquad\qquad\quad \wedge \forall m \in \{1...N\text{-}1\} Reg[j].Snapshot[l] \neq (Reg[m].Name, Reg[m].Names) \\
\qquad\quad) \\
\quad)
\end{array}
$$

/* We check if p has not two index of its Snapshot that do not correspond to the current state of a processor */

$$
\begin{array}{l}
\wedge \quad \neg \quad (\exists i \in \{1...N\text{-}1\} \\
\qquad\quad \forall m \in \{1...N\text{-}1\} Snapshot[i] \neq (Reg[m].Name, Reg[m].Names) \\
\qquad\quad \wedge \exists j \in \{1...N\text{-}1\} \\
\qquad\qquad i \neq j
\end{array}
$$

$\wedge \forall m \in \{1...N\text{-}1\} Snapshot[j] \neq (Reg[m].Name, Reg[m].Names)$

)

/* We check if all the processors but one have an index in their $Snapshot$ that all represent the same state of a processor that is not the current state of a processor*/

$\wedge \quad \neg \quad (\exists i \in \{1...N\text{-}1\}$

$\forall m \in \{1...N\text{-}1\} Snapshot[i] \neq (Reg[m].Name, Reg[m].Names)$

$\wedge \exists j \in \{1...N\text{-}1\}$

$\forall m \in \{1...N\text{-}1\} Reg[j].Snapshot[m] \neq Snapshot[i]$

$\wedge \exists k \in \{1...N\text{-}1\},$

$k \neq j$

$\wedge \forall m \in \{1...N\text{-}1\} Reg[k].Snapshot[m] \neq Snapshot[i]$

)

/* We check if there is no two processors with twice the same $Name$ in their variable $Names$.*/

$\wedge \quad \neg \quad (\exists i \in \{1...N\text{-}1\}$

$\exists j \in \{1...N\text{-}1\}$

$Reg[i].Names[j] = Reg[i].Name$

$\vee \exists k \in \{1...N\text{-}1\}$

$k \neq j$

$\wedge Reg[i].Names[j] = Reg[i].Names[k]$

)

$\wedge \exists l \in \{1...N\text{-}1\}$

$l \neq i$

$\wedge \exists j \in \{1...N\text{-}1\}$

$(Reg[l].Names[j] = Reg[l].Name$

$\vee \exists k \in \{1...N\text{-}1\}$

$k \neq l$

$\wedge Reg[l].Names[j] = Reg[l].Names[k]$

)

)

/* Finally we check if there is no two processors with two $Name$ in their variable $Names$ that correpond not to the current $Name$ of a processor.*/

$\wedge \quad \neg \quad (\exists i \in \{1...N\text{-}1\}$

$\exists j \in \{1...N\text{-}1\}$

$(\forall m \in \{1...N\text{-}1\} Reg[i].Names[j] \neq Reg[m].Name$

$\wedge \exists k \in \{1...N\text{-}1\}$

$k \neq j$

$\wedge \forall m \in \{1...N\text{-}1\} Reg[i].Names[k] \neq Reg[m].Name$

)

$\wedge (\exists l \in \{1...N\text{-}1\}$

$(l \neq i$

$\wedge \exists j \in \{1...N\text{-}1\}$
$\forall m \in \{1...N\text{-}1\} Reg[l].Names[j] \neq Reg[m].Name$
$\wedge \exists k \in \{1...N\text{-}1\}$
$j \neq k$
$\wedge \forall m \in \{1...N\text{-}1\} Reg[l].Names[k] \neq Reg[m].Name$
)
)

Consensus :

/* First we check if all the Snapshot of the processors different from p contain the same values */

if $(\forall i \in \{1...N\text{-}1\}$
$\forall j \in \{1...N\text{-}1\}$
$\wedge \forall k \in \{1...N\text{-}1\}$
$k \neq i$
$\wedge \exists l \in \{1...N\text{-}1\}$
$Reg[i].Snapshot[j] = Reg[k].Snapshot[l]$
$\vee \exists m \in \{1...N\text{-}1\} Reg[i].Snapshot[j] = (Reg[m].Name, Reg[m].Names)$
)

/* We check if all the current state of the processors different from p appears in the $Snapshot$ of their neighbors different from p */

$\wedge$ $(\forall i \in \{1...N\text{-}1\}$
$\forall j \in \{1...N\text{-}1\}$
$i \neq j$
$\wedge \exists k \in \{1...N\text{-}1\}$
$(Reg[i].Name, Reg[i].Names) = Reg[j].Snapshot[k]$
)

/* We check if all the processors different from p have a variable $Names$ which contain the $Name$ of all their neighbors different from p */

$\wedge$ $(\forall i \in \{1...N\text{-}1\}$
$\forall j \in \{1...N\text{-}1\}$
$i \neq j$
$\wedge \exists k \in \{1...N\text{-}1\} Reg[i].Name = Reg[j].Names[k])$
$Reg[i].Name = Reg[j].Names[k])$

/* Finally we check if all the processors different from p have the same value in their $Snapshot$ at the index correponding to p and this value is different from the current state of p */

$\wedge$ $(\forall i \in \{1...N\text{-}1\}$
$\forall j \in \{1...N\text{-}1\}$
$i \neq j$
$(Reg[i].Snapshot[LireMonOrdre(Reg[i])] \neq (Name, Names)$
$\wedge Reg[i].Snapshot[LireMonOrdre(Reg[i])] =$
$Reg[j].Snapshot[LireMonOrdre(Reg[j])]))$