A Lower Bound on Dynamic k-stabilization in Asynchronous Systems*

Christophe Genolini
UFR Application Physique et Sportive
Bâtiment M, Université de Paris X Nanterre, FR 92001 Nanterre cedex
France

Sébastien Tixeuil
Laboratoire de Recherche en Informatique, UMR CNRS 8623
Bâtiment 490, Université de Paris XI Sud, FR 91405 Orsay cedex
France

January 10, 2003

Abstract

It is desirable that the smaller is the number of faults to hit a network, the faster should a network protocol recover. We study the scenario where up to k (for a given k) faults hit processors of a synchronous distributed system by corrupting their state undetectably.

In this context, we show that the well known step complexity model is not appropriate to study time complexity of time-adaptive protocols (*i.e.* protocols that recover from memory corruption in a time that depends only on the number of faults and *not* on the network size). In more details, we prove that for non trivial dynamic problems (such as token passing), there exists a lower bound of $\Omega(D)$ (where D is the network diameter) steps on the stabilization time even when as few as 1 corruption hits the system.

This implies that there exist no time adaptive protocol for those problems in the asynchronous step model, even if we assume that the number of faults is bounded by 1 and that the scheduling of the processors is almost synchronous (between two actions of an enabled processor, any other processor may execute at most D actions).

Keywords: Self-stabilization, Time adaptivity, Transient failures, Dynamic problems, Asynchronous systems, Lower bound.

1 Introduction

Robustness is one of the most important requirements of modern distributed systems, that go through transient faults because they are exposed to constant change of their environment. Resilience to transient failures has been investigated well in the literature, and four families of techniques can be distinguished.

*This work was supported in part by the french MobiCoop project. Contact Author: Sébastien Tixeuil, Email: `tixeuil@lri.fr`, Tel: (33) 1 69 15 42 39, Fax: (33) 1 69 15 65 86.A preliminary abstract of this paper appears in SRDS'02.

1.1 A taxonomy of stabilization

Self-stabilization. Introduced by Dijkstra in [9], *self-stabilization* guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior or the set of *legitimate* states. In more details, let a *configuration* (or a *global state*) be a collection of the states of the individual network nodes, and let $\mathcal{Q}$ be some predicate on a configuration. A self-stabilizing protocol for $\mathcal{Q}$ is one that, when starting from an arbitrary configuration reaches a *legitimate* configuration (a configuration for which $\mathcal{Q}$ holds) and remains in legitimate configurations henceforth (for insight details, see *e.g.* [10]).

Due to its support to an arbitrary large number of failures, this technique often suffers from several drawbacks. For example, the recovery process may cover the whole network even if only a few nodes failed (e.g. [1, 2, 4, 11, 15]).

k-stabilization. Let *k-stabilizing* protocols (see [5]) be stabilizing protocols for the case that an upper bound $k \leq n$ on the number of faults is known (where n is the total number of processors). To model a fault, we use the Hamming distance between configurations (see e.g. [11]). That is, let C_1 and C_2 be configurations, the distance between them is the number of nodes whose local states are different in C_1 and C_2. Let C be an illegitimate configuration, and L be a legitimate one with the smallest distance from C. If L is not unique then let L be any configuration with the smallest distance (this does not influence the analysis). The *number of faults* in C is the distance from L. The *faulty nodes* are those whose state in C and L are different.

Time adaptivity. The fact that the recovery process may involve the whole network does not scale to modern very large networks. To enable scaling, it was suggested by several researchers (e.g. [14, 16, 17]) that the smaller is the number of faults to hit a network, the faster should a network protocol recover. Such protocols (called *fault local*, *fault scalable* or *time adaptive* in the literature) were suggested first for relatively easier (and less typical) cases, such as the case where a faulty node can detect that it is faulty [3], or the case of non-reactive tasks [16, 17] (a distributed function computation that is performed once, and the result is not supposed to ever change), and then (see [5, 12]) for the token passing problem.

k-time adaptivity. It is straightforward to see that self-stabilizing algorithms are a special case of k-stabilizing algorithms (when $k = n$, the size of the network). Similarly, Time adaptive algorithms are simply a refinement of self-stabilizing algorithms. We introduce the notion of k-time adaptivity to denote the ability to recover from at most k corruptions in a time that depends only on the actual number of faults. For historical reasons, most of the so-called "time adaptive" aforementioned papers (*e.g.* [5, 14, 16, 17]) are k-time adaptive according to our terminology: they all assume a bound on the number of faults that hit the network in order to provide fast stabilization time. Our classification is captured by Figure 1 (see also [13] for more details).

1.2 Related work

The study of self-stabilizing protocols was initiated by Dijkstra (see [9]), that proposed three deterministic algorithms on ring networks for the token passing problem. In this problem it is required that exactly one node "possesses the token" (i.e. a locally computable predicate $TOKEN$ holds for that node) at any moment, and that every node eventually holds the token.

Time adaptivity was first introduced in the context of non-reactive systems. In [16] the notion of fault locality was introduced, as well as an algorithm for the simple task, called the *persistent bit*, of recovering from the corruption of one bit at some k nodes, for an unknown k with output stabilization time $O(k \log n)$ for $f = O(n/\log n)$, where f is the actual number of faults that hit the system. In [17] a stabilizing fault local algorithm is presented for the persistent bit task. If the number of faults is smaller than $n/2$ then that algorithm achieves a legitimate state (the same as a self stabilizing algorithm in this case) and the stabilization time for the output is $O(k)$. These algorithms are for synchronous networks. An asynchronous, and self stabilizing version of [17] is described in [18].

		stabilization time depends on the number of faults ↓		stabilization time is finite yet unbounded ↓
any memory corruptions	$\longrightarrow$	time adaptivity	$\subset$	self-stabilization
		$\cap$		$\cap$
at most k memory corruptions	$\longrightarrow$	k-time adaptivity	$\subset$	k-stabilization

Figure 1: Taxonomy of stabilization

In [19], an algorithm for the following problem is presented: given a self-stabilizing non-reactive protocol, produce another version of that protocol which is self-stabilizing, but with output stabilization time in $O(1)$ if $k = 1$. The transformed protocol has $O(T \cdot D)$ step stabilization time, where T is the stabilization time of the original protocol and where D is the diameter of the network (no analysis is provided for output stabilization time when $k > 1$). The protocol of [19] is asynchronous, and its space overhead is $O(1)$ per link. However, it requires a self-stabilizing protocol to start with, and it may suffer a performance penalty in the case of $k > 1$. In [3], faults are stochastic, and consequently the correctness of information can be decided with any desired certainty less than 1. Under this assumption, a time-adaptive algorithm is presented. The algorithm handles both Input-Output relations, and reactive tasks, however, in reactive tasks inputs may be lost if faults affect the nodes that "heard" about these inputs. Additional examples for the special case of $O(1)$ recovery time appears in [7, 11].

Finally, the work by Gosh and Xee (see [14]) provides a method for adding stabilizing properties to non-stabilizing non-reactive systems, when the number of faults is greatly lower than the size of the network. However, their complexity results are strongly related to the repartition of the faults: best results are obtained when the k faults are contiguous (the time complexity is then $O(k^3)$), but performance decreases (exponential in k) when faults are arbitrarily located.

In [5], Beauquier, Genolini and Kutten provide an asynchronous k-stabilizing algorithm for the token ring task in a non-uniform setting (one processor is distinguished from the others and may execute different code). This solution require $O(k^2)$ rounds to stabilize and its space overhead is $O(kn)$. A self-stabilizing algorithm that is also k-stabilizing (with stabilization time $O(k)$) appears in [12]. Unlike [5], [12] performs on synchronous systems, yet still requires that the setting is non-uniform.

1.3 Our contribution

In order to present our results, we recall the two common time related complexity measures: *(i)* The sum (over all nodes) of *steps*, where in one (atomic) step, node P reads a neighbor's state, computes, and writes P's variables, and *(ii) asynchronous time*, or *rounds*, the time assuming (for the sake of time complexity calculation only) that no step lasts longer than one time unit, and that nodes take steps in parallel.

We show that any dynamic global algorithm (*e.g.* solving mutual exclusion) that tolerates at least one memory corruption requires at least $\Omega(D)$ steps to converge, where D is the diameter size of the network. This result implies that there exists no dynamic k-time adaptive global algorithm in the step complexity model, since its step complexity cannot be a function of k (and thus f) alone. Note that this result does not mean that asynchrony should be avoided in distributed system to guarantee good convergence time, but rather that complexity should be studied in models that do support actual parallelism. This also justifies that round complexity was used in previous works to exhibit actual time adaptivity.

Outline. Section 2 presents our system settings along with some formal definitions that will be used during the proofs. Section 3 describes first informally then formally our negative result : it is impossible to construct a time adaptive algorithm for any non-trivial problem in the step model. Section 5 provides some concluding remarks.

2 Model

We use the classical definitions on graphs that are defined in [6]. In this section, we refine informal definitions given in the introduction for self-stabilizing, k-stabilizing, time adaptive and k-time adaptive systems, and give complexity measures.

2.1 Self-stabilization

We model self-stabilizing algorithms as transition systems, whose set of initial configuration is arbitrary (including configurations that are not normally reachable from some other states).

Definition 1 *A* transition system *is a triple $S=(\mathcal{C},\rightarrow, \mathcal{I})$, where $\mathcal{C}$ is a set of configurations, $\rightarrow$ is a binary relation (transition) on $\mathcal{C}$, and $\mathcal{I}$ is a subset of $\mathcal{C}$ of initial configurations. An* execution *(or computation) of S is a sequence $\mathcal{E} = (T_0, T_1, T_2, ...)$ where for all $i \geq 1$, $T_i = C_i \rightarrow C_{i+1}$ and $C_0 \in \mathcal{I}$. A* partial execution *is a (finite) prefix of an execution.*

In asynchronous systems, the difference of speeds between nodes is modeled using a *scheduler*.

Definition 2 *In each configuration, the* scheduler *chooses a subset of the enabled processors. The chosen processor(s) execute an action in the next atomic step. A scheduler is* central *if each time, it chooses exactly one processor. A scheduler is k-*bounded *if, between any two actions of an enabled processor, every other processor executes at most k actions.*

The idea of a k-bounded scheduler was introduced in [8]. In the sequel of the paper, we assume a D-bounded central scheduler, one of the weakest adversary for a distributed asynchronous system.

Then, our lower bound result remains valid for all stronger adversaries: *e.g.* k-bounded scheduler (with $k > D$), central scheduler, arbitrary distributed scheduler.

Definition 3 (Self-stabilization) *A transition system S is self-stabilizing for a specification (of a problem) $\mathcal{SP}$ if there is no initial condition in S (that is $\mathcal{I} = \mathcal{C}$) and if there exists a non-empty subset $\mathcal{L} \subseteq \mathcal{C}$ of legitimate configurations, with the following properties:*

- *(i)* Correctness*: every execution starting in a configuration in $\mathcal{L}$ satisfies $\mathcal{SP}$,*
- *(ii)* Convergence*: every execution contains a configuration in $\mathcal{L}$.*

A common way to model a self-stabilizing algorithm running on a network is to set up the following transition system : *(i)* a configuration C is the set of register values of all network processors, *(ii)* $\mathcal{C}$ is the set of all possible configurations C, *(iii)* a couple of configurations (C_1, C_2) belongs to the transition relation $\rightarrow$ if from C_1, the execution of the algorithm leads to C_2.

2.2 k-stabilization

Roughly speaking, k-stabilizing algorithms have nearly the same definition as self-stabilizing algorithms, except for the addition of some assumptions on the initial configuration: an initial configuration is one that can be constructed from a legitimate configuration by changing the register values of up to k processors. More formally, let the (Hamming) distance $Dist(C_1, C_2)$ between two configurations C_1 and C_2 be the number of processors whose states are different in C_1 and C_2; the distance between C_1 and a set of configuration $\mathcal{C}_2$ is $Dist(C_1, \mathcal{C}_2) = \min_{C \in \mathcal{C}_2}\{Dist(C_1, C)\}$.

Definition 4 *Let S be a system and $\mathcal{L}$ a set of configurations. The* ball of center $\mathcal{L}$ with radius k *is the set $Ball^k_{\mathcal{L}}$ of all configurations C such that the Hamming distance between C and $\mathcal{L}$ is smaller or equal to k.*

The following notion is used heavily in the description of the algorithms, as well as in the proofs.

Definition 5 *Let S be a system, C be a configuration and $\mathcal{L}$ be a set of configurations. Let L be a configuration of $\mathcal{L}$ such that the distance between C and L is minimal. The set of processors $\mathcal{P}$ in C that are* corrupted relatively to L *is the set of processors that do not have the same value in C and in L.*

Given a configuration C, the notion of corrupted processor exists only relatively to a specific legitimate configuration. For some configurations, neither L nor $\mathcal{P}$ are unique. However, for the sake of the proofs it suffices to choose an arbitrary legitimate configuration L in $\mathcal{L}$ whose distance from C is minimal. When the set $\mathcal{L}$ is known, we use the term "corrupted processor", omitting the reference to L.

Definition 6 (k-stabilization) *A system S is* k-stabilizing *for a specification $\mathcal{S}$ if there exists a non-empty subset $\mathcal{L} \subseteq \mathcal{C}$ of legitimate configurations with the following properties:*

- *(i)* Correctness*: every execution starting in a configuration in $\mathcal{L}$ satisfies $\mathcal{S}$,*
- *(ii)* k-Convergence*: every execution starting in a configuration in $Ball^k_{\mathcal{L}}$ contains a configuration in $\mathcal{L}$.*

An execution starting with a configuration in $\mathcal{L}$ is called a legitimate execution.

Note that since $Ball^k_{\mathcal{L}}$ is not necessarily equal to $\mathcal{C}$, a k-stabilizing system is not necessarily self-stabilizing. The converse is false (every self-stabilizing system is also k-stabilizing for any $k \leq n$, where n is the number of processors in the network).

2.3 Stabilization time

A way to measure the time efficiency of self-stabilizing and k-stabilizing systems is to evaluate the number of steps before reaching a legitimate state.

Definition 7 *Given a self-stabilizing (or a k-stabilizing) system and an execution $\mathcal{E}$, the* stabilizing phase *is the prefix of $\mathcal{E}$ that ends at the first legitimate configuration. The* stabilization time *of the execution is the length of the stabilizing phase. The* stabilization time *of the system is the greatest stabilization time (if it exists) of all the possible executions of the system.*

The motivation for studying k-stabilization in previous works was to obtain shorter stabilization time under the assumption of a small number of corrupted processors (making the solution time adaptive when the number of corrupted processors was bounded by k).

2.4 Time adaptivity, k-time adaptivity

A distributed system is time adaptive if it is self-stabilizing and its stabilization time is polynomial in the actual number of faults f.

Definition 8 (Time adaptivity) *A transition system S is time adaptive for a specification $\mathcal{SP}$ if there is no initial condition in S (that is $\mathcal{I} = \mathcal{C}$) and if there exists a non-empty subset $\mathcal{L} \subseteq \mathcal{C}$ of legitimate configurations, with the following properties:*

- *(i)* Correctness*: every execution starting in a configuration in $\mathcal{L}$ satisfies $\mathcal{SP}$,*
- *(ii)* Convergence*: every execution contains a configuration in $\mathcal{L}$,*
- *(iii)* f-Polynomial stabilization time*: the stabilization time is bounded by a polynomial in f.*

Definition 9 (k-time adaptivity) *A system S is* k-time adaptive *for a specification $\mathcal{S}$ if there exists a non-empty subset $\mathcal{L} \subseteq \mathcal{C}$ of legitimate configurations with the following properties:*

- *(i)* Correctness*: every execution starting in a configuration in $\mathcal{L}$ satisfies $\mathcal{S}$,*
- *(ii)* k-Convergence*: every execution starting in a configuration in $Ball^k_{\mathcal{L}}$ contains a configuration in $\mathcal{L}$,*
- *(iii)* f-Polynomial stabilization time*: the stabilization time is bounded by a polynomial in f.*

3 Impossibility of time adaptivity in the step model

In this section, we prove that any dynamic global algorithm that tolerates at least one memory corruption requires at least $\Omega(D)$ steps to converge, where D is the diameter size of the network.

3.1 Informal description

Considering arbitrary reactive tasks on arbitrary graphs hints at the same principle. In a reactive system, at least one processor may be activated in any configuration (otherwise, there would exist a terminal configuration and the system would not be dynamic). Moreover, we assume that in order to be corrected, a faulty processor or one of its neighbors has to be activated by the scheduler. More precisely, we assume that a configuration C can be obtained from a legitimate configuration L by corrupting processor P and such that any computation whose initial configuration is C may not reach a legitimate configuration before P or one of its neighbors has been activated (otherwise, a reactive system would solve a task independently of the state of one of the processors). Under these hypotheses, it is possible to construct a system computation whose stabilization time in steps is close to the network diameter.

In the case of the mutual exclusion on a ring network, this result comes from the fact that the system scheduler may choose to activate any processor in the system. Then, in the case of a single memory corruption, the system scheduler may ignore the corrupted processor and its neighbors, and activate only the processor that hold the correct token. This token may move around the (almost) entire ring before entering in the corrupted zone and correcting memory. By not activating the corrupted processors, the scheduler prevents them from correcting themselves quickly.

The same idea also hold for other dynamic problems set up on more complex communication graphs.

3.2 Formal result

Hypothesis and result. We consider a distributed system whose communication graph is arbitrary, and we assume that the system scheduler is central (it activates a single processor at any time) and D-bounded (between two actions of an enabled processor, every other processor executes at most D actions, wher D is the network diameter). In addition, the communication graph is bidirectional, the distributed system is non-anonymous (processors have access to unique identifiers) and non-uniform (processors may execute different code according to their identifier).

We assume that the problem P to solve satisfies:

1. in any configuration, at least one processor can be activated by the scheduler,
2. in any correct computation, each processor is either activated or has at least one of its neighbors activated at least once,
3. if configuration C is obtained from configuration L by corrupting processor P, then any computation whose initial configuration is C may reach a legitimate configuration only after either P or one of its neighbors has been activated.

Note that *e.g.* mutual exclusion satisfies these three hypotheses.

Result We are now able to state our theorem:

Theorem 1 *Under the previous hypotheses, any distributed system that tolerates at least one fault for P has a stabilization time of $\Omega(D)$ steps.*

This lower bound leads up to the following impossibility result concerning time-adaptivity in asynchrnous systems.

Corollary 1 *Under the previous hypotheses, there exists no k-time adaptive system for P in a graph whose diameter is proportional to the number of processors.*

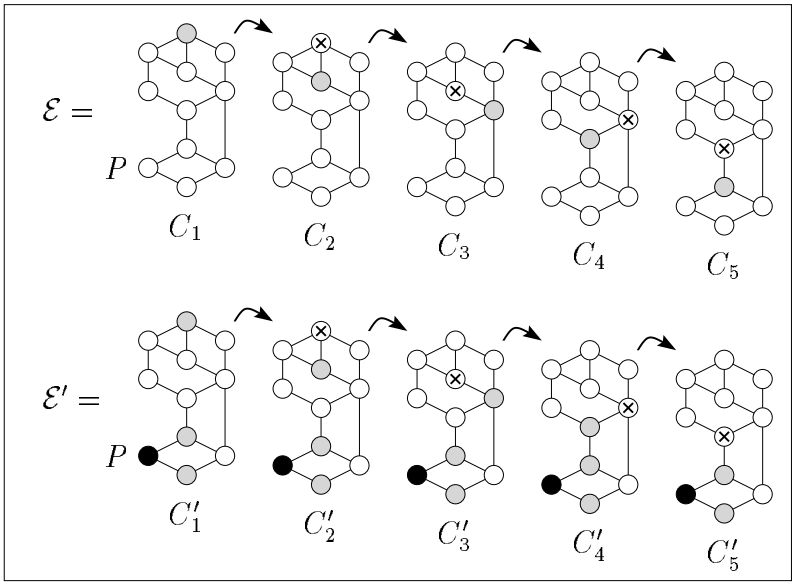


Figure 2: Constructing a computation whose stabilization time is $D-2$

Proof overview of Theorem 1 Given an arbitrary graph G and an arbitrary distributed system $\mathcal{DS}$, we construct a computation of $\mathcal{DS}$ whose initial configuration is 1-faulty and whose stabilization time is $D-2$. Consider a computation $\mathcal{E}$ of $\mathcal{DS}$ starting from a legitimate configuration C_1 and a processor P such that the scheduler activates neither P nor one of its neighbors during the first $D-2$ first steps of $\mathcal{E}$. Then, if C'_1 is the configuration obtained from C_1 by corrupting P's variables, computation $\mathcal{E}'$ starting from C'_1 (and such that the scheduler activates the same processes as in $\mathcal{E}$) cannot stabilize in less than $D-2$ steps.

An example of such a construction is given in Figure 2. The grayed processors denote processors that can be activated by the system scheduler. In computation $\mathcal{E}$ starting from C_1, neither P nor one of its neighbors are activated during the 4 first steps. Then we consider C'_1, obtained by corrupting processor P in configuration C_1. Then, if the scheduler activates the same sequence of processors in $\mathcal{E}'$ (the computation starting from C'_1) as in $\mathcal{E}$, then $\mathcal{E}'$ does not converge in less than $5-2$ steps.

Extensive proofs are provided in the following Section.

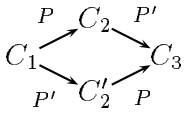
4 Proof of impossibility result (Theorem 1)

We consider a distributed system $\mathcal{S}$ that tolerates at least one memory corruption for a given specification $\mathcal{P}$, and whose communication graph G is of size n. Each processor P maintains a set of variables denoted by $State(P)$. If C_1 and C_2 are two configurations, then $State_{C_1}(P) = State_{C_2}(P)$ means that P's variables have the same values in both C_1 and C_2. Similarly, two configurations are identical if all processor variables have the same values in any of these two configurations.

Constructing a $\Omega(D)$ steps computation We define two processors P_1 and P_2 to be *independent* if the order they are activated by the scheduler is unsignificant to the remaining of the computation. Otherwise, P_1 and P_2 are *dependent*. Our notion of dependency is related to the notion of causality [20].

In other terms, if two processors P and P' are independant in C_1, the configuration obtained by activating P then P' is the same as the configuration obtained by activating P' then P.

Remark 1 *If two processors are not neighbors, they are independent.*



Similarly, a processor P and a set of processors $\{P_i\}$ are independent if P is independent with each of the P_i. Otherwise P and $\{P_i\}$ are dependent.

A sequence of processors $(P_i)_{0 \leq i \leq m}$ is dependant if for each processor P_i, P_i and $\{P_j\}_{0 \leq j \leq i-1}$ are dependant.

Lemma 1 *Let $\mathcal{E}$ be a system computation. Let*

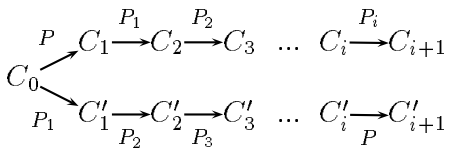
$$C_0 \xrightarrow{P} C_1$$

be the first transition of $\mathcal{E}$ and

$$C_1 \xrightarrow{P_1} C_2 \xrightarrow{P_2} C_3 \xrightarrow{P_3} ... \xrightarrow{P_i} C_{i+1}$$

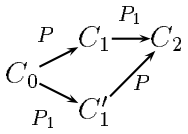
be the next i transitions. If processor P and the processor set $\{P_i\}$ are independent, then the computation obtained by activating first P_1, P_2, ..., P_i and then P also leads to configuration C_{i+1}.

Proof: *A priori*, we have :

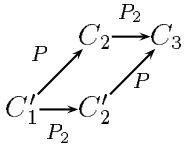


Let us prove that C_{i+1} and C'_{i+1} are identical.

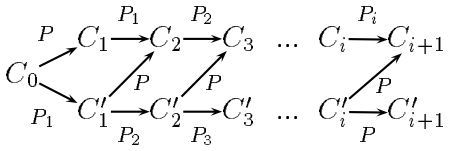
By hypothesis, P and P_1 are independent, thus:



In C'_1, P et P_2 are independent, thus:



Finally, we obtain:

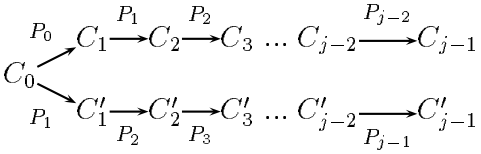


Which proves $C_{i+1} = C'_{i+1}$. □

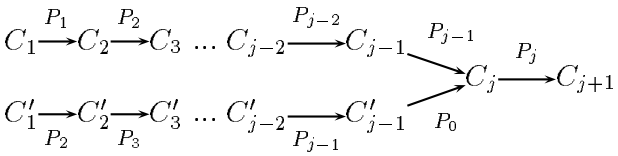
Lemma 2 *Let $\mathcal{S}$ be a distributed system that tolerates at least one memory corruption for a given specification $\mathcal{P}$. Then there exists a computation whose first activated processor is on the border of the communication graph and such that the first $D-2$ first activated processors are dependent.*

Proof: We prove by recurrence that there exists a computation whose first i activated processors are dependent. Let $\mathcal{E}$ be a system computation whose first activated processor is on the border of the communication graph ($\mathcal{E}$ exists since in any computation, every processor must be activated at least once).

1. We suppose that the system communication graph has a diameter at least 4. We construct a computation whose two first activated processors are dependent: Let $C_0 \xrightarrow{P_0} C_1$ be the first transition of $\mathcal{E}$. Let P_j the first processor of $\mathcal{E}$ that is dependent to P_0 (P_j exists by Assumption 3 in Section 3.2, note that P_j might be P_0 itself). All processors between P_0 and P_j are independent from P_0, thus

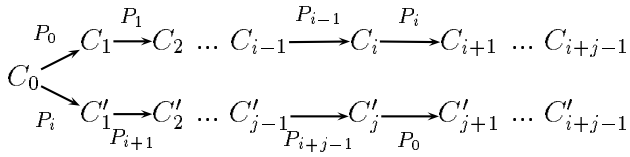


and

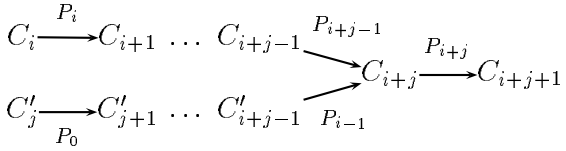


Now consider computation $\mathcal{E}'$ whose initial configuration is C'_{j-1}. The two first transitions of $\mathcal{E}'$ are $C'_{j-1} \xrightarrow{P_0} C_j \xrightarrow{P_j} C_{j+1}$. The two first activated processors in $\mathcal{E}'$ are then dependent.

2. We suppose that the system communication graph has a diameter at least $i+3$ (where $i \geq 1$). Assume now that there exists a computation $\mathcal{E}$ whose first i activated processors are dependent, and let us construct a computation $\mathcal{E}'$ such that the first $i+1$ activated processors are dependent. Let P_{i+j} the first processor that is dependent to $(P_0, P_1, ...P_{i-1})$ (P_{i+j} may be in the set $\{P_0, P_1, ...P_{i-1}\}$). None of the processors between P_{i-1} and P_{i+j} is dependent to $(P_0, P_1, ...P_{i-1})$, thus we have:



and



Now consider computation $\mathcal{E}'$ whose initial configuration is C'_j. The $i+1$ first transitions of $\mathcal{E}'$ are

$$C'_j \xrightarrow{P_0} C'_{j+1} \xrightarrow{P_1} C'_{j+2} \longrightarrow \dots$$
$$\dots \longrightarrow C'_{i+j-1} \xrightarrow{P_{i-1}} C'_{i+j} \xrightarrow{P_{i+j}} C'_{i+j+1}$$

The $i+1$ first activated processors in $\mathcal{E}'$ are then dependent.

In conclusion, it is possible to construct a computation whose first $D-2$ activated processors are dependent, where D is the system communication graph diameter. □

Lemma 3 *Let $\mathcal{E}$ be a computation. Let $(P_0, P_1, ..., P_i)$ be a sequence of dependent processors. Then the set of processors denoted by $\{P_j\}_{j\in[0..i]}$ is connected.*

Proof: We prove this lemma by recurrence on the number of processors in the sequence.

1. We consider a sequence of two dependent processors : they are neighbors and their set is connected (see Remark 1).

2. Assume the property is verified for a sequence of i processors ($i \geq 2$). Let us show that the property still hold for $i+1$ processors.

   Let P_i be the last processor of the sequence. The set of the $i-1$ first processors is connected by recurrence hypothesis. Since P_i is dependent to the set $\{P_j\}_{j\in[1..i-1]}$, it is dependent to at least one of the P_j. They are then neighbors and the set

   $$\{P_j\}_{j\in[1..i-1]} \cup P_i$$

   is connected.

Finally, a sequence of dependent processors is connected in the distributed system communication graph. □

Lemma 4 *Let C be a configuration and let CS be a connected set of $D-2$ processors such that at least one of the processors in CS is in the border of the communication graph. Then there exists in G a processor P such that none of P's neighbors is in CS.*

Proof: Let P' be a processor in the border of the graph. Let P be a processor such that the distance between P' and P is the graph diameter. the distance between P' and P's neighbors is then between $D-1$ and D. Thus none of P's neighbors is in CS. □

Lemma 5 *Let $\mathcal{S}$ be a distributed system that tolerates at least one memory corruption for specification $\mathcal{P}$. Then there exists a computation of $\mathcal{S}$ whose stabilization phase is at least $D-2$ steps.*

Proof: Let $\mathcal{E}$ be a computation such that the $D-2$ first activated processors $(P_0, P_1, ..., P_{n-3})$ are a dependent sequence and such that P_0 is in the border of the communication graph. Let P be a processor such that neither P, nor one of its neighbors are in $(P_0, P_1, ..., P_{D-3})$ (P exists from Lemma 4). Let C_0 be the initial configuration of $\mathcal{E}$. Let C'_0 be the illegitimate configuration obtained from C_0 by corrupting processor P memory. Let us prove that there exists a computation $\mathcal{E}'$ whose initial configuration is C'_0 that do not converge within $D-2$ transitions.

In C'_0, the scheduler may choose to activate P_0 (since P_0 and its neighbors have the same values both in C_0 and in C'_0). The transition $C'_0 \xrightarrow{P_0} C'_1$ leads the system into an illegitimate configuration (because neither P nor its neighbors have changed their values). In C'_1, the scheduler may choose to activate P_1. The system would then reach an illegitimate configuration C'_2. By induction, after $D-2$ steps, all processors of the sequence $(P_0, P_1, ..., P_{D-3})$ have been activated by the scheduler, yet the states of P and its neighbors have not been modified, thus the system is still in an illegitimate state. □

Then Theorem 1 is a direct consequence of Lemma 5 (that ensures the existence of a computation whose convergence time is $\Omega(D)$).

That completes the impossibility result for k-time adaptive algorithms in the step model.

5 Conclusion

In this paper, we investigated the problem of reactive tasks in a time adaptive context. In more details, we proved that there exists no time adaptive (or k-time adaptive, for what matters) algorithm for any non-trivial reactive task in the asynchronous step time complexity model.

This impossibility result is due to a lower bound of $\Omega(D)$ (where D is the network diameter) on stabilization time for a 1-stabilizing system (*i.e.* only one corruption may occur) performing under a D-bounded central scheduler (between any two actions of an enabled scheduler, any other processor may execute at most D actions).

This work justifies *a posteriori* that previous works on time adaptivity make use of one of the two following hypotheses:

1. the system is synchronous (in each configuration, the scheduler chooses *all* enabled processors to execute an action) (*e.g.* [17, 13]),

2. the time complexity is calculated in *rounds* (in one round, every enabled processor executes at least one action) instead of steps (*e.g.* [18]).

Also, this result implies that Figure 1 can be refined as Figure 3, since there exists some problems (*e.g.* mutual exclusion in asynchronous systems) that admit a self-stabilizing solution (and *a fortiori* a k-stabilizing solution, for any k), but that admit no k-time adaptive solution (and *a fortiori* no time-adaptive solution).

		stabilization time depends on the number of faults		stabilization time is finite yet unbounded
		$\downarrow$		$\downarrow$
any memory corruptions	$\longrightarrow$	time adaptivity	$\subsetneq$	self-stabilization
		$\cap$		$\cap$
at most k memory corruptions	$\longrightarrow$	k-time adaptivity	$\subsetneq$	k-stabilization

Figure 3: Taxonomy of stabilization revisited

References

[1] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proc. of the 32nd IEEE Symp. on Foundation of Computer Science (FOCS'91)*, pp. 268–277, 1991.

[2] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilization by local checking and global reset. In *Proc. 8th International Workshop on Distributed Algorithms (WDAG'94)*, 1994.

[3] Y. Afek and S. Dolev. Local stabilizer. In *Proc. of the 5th Israel Symposium on Theory of Computing and Systems*, June 1997.

[4] Y. Afek, S. Kutten, and M. Yung. Local Detection for Global Self-Stabilization *Theoretical Computer Science*, No 186, pp. 199-229, 1997.

[5] J. Beauquier, C. Genolini, and S. Kutten. Optimal Reactive k-Stabilization : the case of Mutual Exclusion. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99)*, pp. 209–218, May 1999.

[6] C. Berge. Graphs and hypergraphs. Translated from the French by Edward Minieka. Second revised edition. *North-Holland Mathematical Library* 6 (1976).

[7] I. Chlamtac and S. Pinter. Distributed node organization algorithm for channel access in a multihop dynamic radio network. *IEEE Transactions on Computers*, Vol. C-36, No 6, pp. 728-737, June 1987.

[8] S. Delaët. Auto-stabilisation: modèle et applications à l'exclusion mutuelle. *Ph.D. Dissertation, Université Paris Sud* December 1995.

[9] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, Vol. 17, No. 11, pp. 643-644, Nov. 1974.

[10] S. Dolev. Self-stabilization. *The MIT Press*, 2000.

[11] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, Vol. 3, No. 4, 1997. Also in *Proc. of the Second Workshop on Self-Stabilizing Systems (WSS'95)*, pages 3.1–3.15, May 1995.

[12] C. Genolini. Optimal k-stabilization: the Case of Synchronous Mutual Exclusion. In *Proc. of the International Conference Parallel and Distributed Computing and Systems (PDCS'00)*, M. Guizani and X. Shen editors, pp. 371-376, Nov. 2000.

[13] C. Genolini. Raffinements de l'auto-stabilisation. Ph. D. Thesis, *Université de Paris Sud*, Dec. 2000.

[14] S. Gosh and Xee. Scalable Self-stabilization. In *Proc. of the 4th Workshop on Self-stabilizing Systems (WSS'99)*, pp. 18-24, Austin, Texas, Jun. 1999.

[15] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. In *Distributed Computing*, Vol. 7, 1994.

[16] S. Kutten and D. Peleg. Fault-local distributed mending. In *Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing (PODC'95)*, Aug. 1995.

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

[18] S. Kutten and B. Patt-Shamir. Asynchronous Time-Adaptive Self Stabilization. a Brief Announcement in the *Proc. of the 17th Annual ACM Symp. on Principles of Distributed Computing (PODC'98)*, 1998.

[19] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemamraju. Fault-containing self-stabilizing algorithms. In *Proc. of the 15th Annual ACM Symp. on Principles of Distributed Computing (PODC'96)*, Philadelphia, Pennsylvania, pp. 45-54, USA, May 1996.

[20] G. Tel. Introduction to Distributed Algorithms. *Cambridge University Press*, 1994.