k-stabilization is Strictly Stronger than Self-stabilization

Christophe Genolini Sébastien Tixeuil

Laboratoire de Recherche en Informatique, UMR CNRS 8623
Bâtiment 490, Université de Paris Sud, FR 91405 Orsay cedex, France
Contact author: Sébastien Tixeuil, `tixeuil@lri.fr`

Abstract

Self-stabilization is a useful and versatile technique for supporting an arbitrary number of transient faults: a self-stabilizing protocol recovers in finite time starting from any initial system configuration. However, the cost for self-stabilization is often high: *(1)* several problems are impossible to solve in a self-stabilizing way (such as deterministic token passing in uniform rings of arbitrary size), and *(2)* even the slightest inconsistency may require the recovery process to cover the whole network, causing a high recovery time.

The recent definition of k-stabilization studies the scenario where up to k (for a given k) faults hit processors of a synchronous distributed system by corrupting their state undetectably. The exact number of faults, the specific faulty nodes, and the time the faults hit are *not* known. So far, k-stabilizing protocols were only used to circumvent drawback *(2)*, and the question of having problems admitting no solution in the self-stabilizing setting while admitting solutions in the k-stabilizing setting remained open.

We positively answer this question by presenting a k-stabilizing solution for the deterministic token passing in uniform arbitrary-sized uniform rings. Our solution assumes that no more than k faults hit the system (and k is lower than $\frac{n-1}{8}$, where n is the size of the system). Our algorithm has the additional property of being time adaptive: the time needed to recover from failures is proportional to the actual number of faults.

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.

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, among which are *(i)* some problems are impossible in particular settings ([16]

proves that there exists no deterministic self-stabilizing token passing algorithm in uniform rings), and *(ii)* the recovery process may cover the whole network even if only a few nodes failed (e.g. [2, 3, 5, 11, 17]).

k-stabilization. Let *k-stabilizing* protocols (see [7]) 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, 18, 19]) 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 [4], or the case of non-reactive tasks [18, 19] (a distributed function computation that is performed once, and the result is not supposed to ever change), and then (see [7, 12]) for the token passing problem.

k-time adaptivity. Self-stabilizing algorithms can be seen as a special case of k-stabilizing algorithms (when $k = n$, the size of the network). Similarly, Time adaptive algorithms can simply be seen as 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.* [7, 14, 18, 19]) 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 for the algorithm-centric point of view, where the arrow symbols denote "is a special case of". For example the arrow between "time adaptivity" and "self-stabilization" denotes that every time adaptive protocol is also a self-stabilizing protocol.

Figure 2 captures the problem-centric point of view of our classification. For example, every problem that admits a self-stabilizing solution also admits a $(n-1)$-stabilizing solution (*e.g.* the self-stabilizing solution), thus the set of problems that can be solved by self-stabilizing algorithms is included in the set of problems that can be solved by $(n-1)$-stabilizing algorithms. For the same reasons, k-time adaptive problems (problems that can be solved using a k-time adaptive protocol) are included in the set of k-stabilizing problems (that can be solved using the same protocol as in the k-time adaptive case).

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 $TOKEN$

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

Figure 1: Taxonomy of stabilization

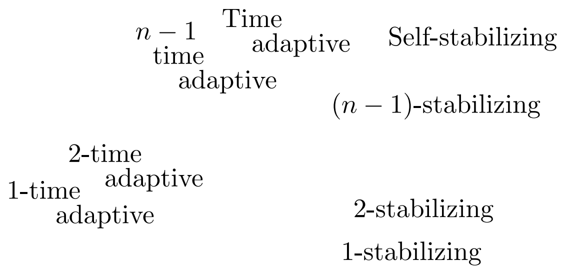


Figure 2: Stabilizing problems

holds for that node) at any moment, and that every node eventually holds the token. A common point of this seminal paper with many subsequent works on self-stabilizing deterministic token passing is that the system is non-uniform: a distinguished processor may execute a local algorithm that is different from the other processors. This hypothesis is needed to break symmetry, since [16] proved that there exists no self-stabilizing uniform deterministic algorithm for the token passing problem on arbitrary-sized ring networks.

Time adaptivity was first introduced in the context of non-reactive systems. In [18] 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 effective number of faults that hit the system. In [19] 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 [19] is described in [20]. In a previous work [13], we prove that there exists a lower bound on the number of processor actions for dynamic k-stabilizing systems (such as token passing systems) to recover from as low as 1 memory fault. This lower bound implies that either round complexity must be used to compute stabilization time, or synchronous systems must be considered.

In [21], 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 (no analysis is provided for output stabilization time when $k > 1$), and D is the diameter of the system. The protocol of [21] 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 [4], 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 [8, 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 [7], 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 [7], [12] performs on synchronous systems, yet still requires that the setting is non-uniform.

1.3 Our contribution

Until now, the notion of k-stabilization was defined and used only to provide better convergence time when the number of faults that hit the system is small. From a problem-centric point of view, it is trivial to see that if a problem admits a self-stabilizing solution, it also admits a k-stabilizing solution (because self-stabilizing algorithms are a special case of k-stabilizing algorithms). Yet there remains the open question of whether there exists a problem that admits a k-stabilizing solution and no self-stabilizing one.

We answer positively to this question by showing that the token passing problem, that do not admit self-stabilizing solutions in some setting, admits a k-stabilizing solution (that we provide) in the same setting. In more details, we present the first deterministic uniform k-stabilizing algorithm for token passing in arbitrary-sized rings. Unlike the algorithms previously presented in [7, 12], we do not assume that a distinguished processor (that may execute actions that are different from the other processors) is available. All processors are identical and the network setting is symmetrical (a unidirectional ring): the system is uniform and anonymous.

While k-stabilization permits us to unlock an impossibility result, we moreover retain its advantages in terms of time complexity. Our token passing algorithm is also k-time adaptive and its stabilization time is $O(f)$, where f is the actual number of faults that initially hit the system. We assume that the number of corruptions k is bounded by $\frac{n-1}{8}$.

Outline. Section 2 presents our system settings along with some formal definitions that will be used during the proofs. In Section, 3 we present a deterministic uniform k-stabilizing algorithm for token passing, while Section 4 provides proofs and complexity results that are related to this algorithm. Section 5 provides some concluding remarks.

2 Model

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.*

Definition 2 (Self-stabilization) *A transition system S is self-stabilizing 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 $\mathcal{L}$.*

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

Definition 4 *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 5 (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 $Ball_{\mathcal{L}}^{k}$ contains a configuration in $\mathcal{L}$. An execution starting with a configuration in $\mathcal{L}$ is called a legitimate execution.*

Note that since $Ball_{\mathcal{L}}^{k}$ 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 rounds before reaching a legitimate state.

Definition 6 *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 7 (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 8 (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.*

2.5 System settings

We consider a *synchronous* system. All processors execute one step of their local code at the same time. One such local step consists in: *(i)* reading the neighbors' state, *(ii)* performing some local calculus, and *(iii)* updating variable values. One global step where every processors executes a local step is called a *round*.

3 Uniform k-time adaptive token passing

In this section, we describe our uniform token passing algorithm for ring networks, first informally then formally. Next we informally give the main ideas for its correctness proof, that is extensively provided in Section 4.

3.1 Informal description

The algorithm that we formally present in Section 3.2 solves the token passing problem in a uniform network where each processor code is identical and deterministic, yet support up to k faults, where $k \leq \frac{n-1}{8}$. In what follows, k denotes the maximal number of supported failures, whereas f is the effective number of failures.

The proof of impossibility of [16] stating that there exists no self-stabilizing solution to this problem is based on a symmetry reasoning: assume that the ring size n is even and that one contiguous half of the network is corrupted in order that the state of processor i is the same as that of processor $i + \frac{n}{2}$. Assume the token was on the non corrupted part of the network: now for symmetry reasons, two symmetric tokens are present. Assume the system is synchronous, and the two tokens remain at the same distance from each other forever. This implies that the system never reaches a configuration where a single token exists in the network, and proves that the system can not be self-stabilizing.

Algorithm 3.1 Uniform k-stabilizing token passing

Variables :

$Val(P)$	used to compute the token position	$Val(P) \in [0..SND(n)-1]$
$Distance(P)$	used to compute the distance to the nearest predecessor holding a token	$Distance(P) \in [0..2n]$
$Counter(P)$	used to delay token retransmission	$Counter(P) \in [0..2n]$

Predicates :

$Token(P)$	$\Longleftrightarrow$	$Val(P) \neq Val(P^-) + 1 \ mod \ SND(n)$
$Ready(P)$	$\Longleftrightarrow$	$Counter(P) > Int\left\{\frac{2n}{(Distance(P^-))}\right\}$

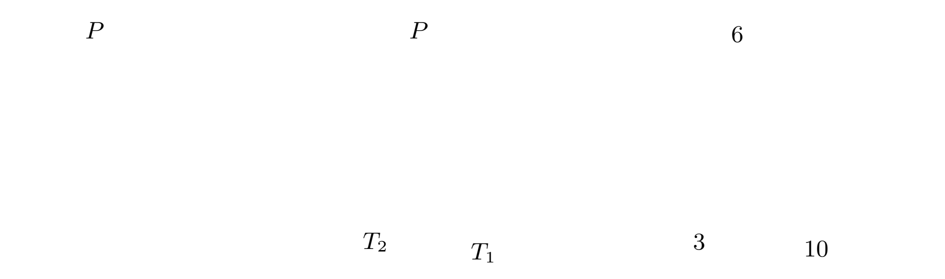
Guarded Rules :

R1	$Token(P) \wedge Ready(P)$	$\Longrightarrow$	$\begin{cases} Val(P) \longleftarrow Val(P^-) + 1 \ mod \ SND(n) \\ Distance(P) \longleftarrow Distance(P^-) + 1 \\ Counter(P) \longleftarrow 0 \end{cases}$
R2	$Token(P) \wedge \neg Ready(P)$	$\Longrightarrow$	$\begin{cases} Distance(P) \longleftarrow 0 \\ Counter(P) \longleftarrow Counter(P) + 1 \end{cases}$
R3	$\neg Token(P)$	$\Longrightarrow$	$\begin{cases} Distance(P) \longleftarrow Distance(P^-) + 1 \\ Counter(P) \longleftarrow 0 \end{cases}$

In order to provide support for up to k memory corruptions, we associate to each token a *speed*, which depends on how many correct predecessors a processor have. The knowledge of the number of correct predecessors is carried out through a variable that captures the distance to the next predecessor token. Then, a token whose k predecessors are correct will have maximal speed and move to the next processor every two synchronous rounds, while a token whose k predecessors are not all correct will get a lower speed, and eventually be catched by a fast token. Since the actual number of faults is lower or equal to k, we are ensured that none of the tokens resulting from memory corruption will get a maximal speed and hold it forever. Moreover, when $k \leq \frac{n-1}{8}$, we are ensured that at least one token has k correct predecessors, and then get maximal speed to remove extra tokens. Finally, a careful analysis carried out in Section 4 shows that the stabilization time from a faulty configuration with f faults is proportional to f.

3.2 Formal description

We assume that when a variable overflows, its value is set to the maximum possible value for that variable. For example, if variable $v = n$ is in range $[0..n]$ and that we execute $v \longleftarrow v + 1$, then v still equals n. If the calculus is made explicitly using the modulo operator, the modulo operation takes place after valuing the variable. For example, if variable $v = n$ is in range $[0..n]$ and that we execute $v \longleftarrow v + 1 \ mod(n)$, then v would then equal 0, because $v + 1$ remains equal to n, and then $nmod(n)$ equals 0. The core of our algorithm is presented as Algorithm 3.1, where SND is a function that returns the smallest non divisor of its argument, and where $Int\{x\}$ denotes the



(a) Legitimate conf. L_1 (b) Corrupted conf. C_2 (c) Distance between tokens

Figure 3: Example of configurations

integer part of x.

Notations and definitions If P is a processor, then P^- denotes the predecessor of P and P^+ its successor. If T is a token, the *support* of T is the processor P holding T, and T^- is equal to P^-. The token predecessor of T is the closest predecessor of P holding a token. According to the unidirectional ring orientation, $Dist(P, P')$ is the distance between P and P' including P and excluding P'. Similarly, $Dist(T, T')$ is the distance between T and T', including T and excluding T'.

Theorem 1 *Provided that no more than $k \leq \frac{n-1}{8}$ faults hit the network, Algorithm 3.1 recovers correct behavior with respect to token passing specification within $O(f)$ rounds, where f is the actual number of initial faults.*

Proof overview: We define two kinds of legitimate configurations: $\mathcal{TP}$-legitimate configurations are configurations that are legitimate enough for the algorithm to satisfy immediately its specification (the *output* variables are correct), while the fully legitimate configurations (or simply legitimate configurations) are those where *all* variables are correct.

A figurative way to visualize each configuration is given in Figure 3, where the ring network is drawn as a tower, and a higher Val value at a processor is depicted by a higher wall. Given this representation, a processor has a token if its top line is cut (like processor P in Figure 3.a). Since operations on Val variables are performed modulo $SND(n)$, the break between the highest and the lowest value does not denote a token, so that in Figure 3, L_1 is a $\mathcal{TP}$-legitimate configuration. The **correctness** proof shows that starting from a $\mathcal{TP}$-legitimate configuration, the algorithm satisfies the token passing specification, and that a fully legitimate configuration is reached after at most $2n$ rounds. This proof reuses the techniques of [9, 12]. The **convergence** proof shows that starting from a fully legitimate configuration that suffered f memory corruptions, a $\mathcal{TP}$-legitimate configuration is reached in time $O(f)$.

In more details, the convergence part is related to the following property: when a group of contiguous processors is corrupted, at least two tokens appear, one at the "head" of the group, and the other at the "tail" of the group. Moreover, the head token can not be far from the tail token since by hypothesis there is at most f corrupted processors on the ring. For example, in configuration C_2 obtained from L_1 by corrupting 3 processors (see Figure 3.b), there are two additional tokens.

Token T_2 is at the "head" of the corruption, while token T_1 is at the "tail" of the corruption, and the distance between T_1 and T_2 is 3. Then we distinguish three kinds of tokens: *(i)* the *true token* is the token that existed in the legitimate configuration L_1 (hold by P), *(ii)* a *false token* (here T_2) appeared due to a corruption: we wish this token does not move, since a corrector token is nearby, *(iii)* a *corrector token* (here T_1) also appeared due to a corruption: we wish this token moves and catches the false token. According to our algorithm, token speeds are conversely proportional to the distance between a token and its predecessor token. Then the false token would move slowly, whereas the corrector token would move quickly, so that the corrector token eventually catches the false token and both tokens disappear.

The **convergence time** is strongly related to the number of rounds needed by corrector token to catch false tokens. This time is proportional to the distance between the two tokens, in turn this distance is bounded by the number of faults.

4 Proof of Algorithm 3.1 (Theorem 1)

For this proof, we reuse some techniques previously used in [7, 12], that consider the *propagation of inputs* and the *propagation of faults*: Intuitively, in a non-trivial reactive system, nodes change their states as a result of the states of their neighbors; for example, when a node P stops holding the token, and its neighbor P' starts holding the token as a result. This *propagates* (to P') the input that told P to release the token. If, however, P acted as a result of a fault, then P' should not have changed its state; now that it did, P's state is now corrupted, and we say that the fault has *propagated* (to P). Intuitively, the techniques we use here bound the propagation of faults. Such bounding is essential for time adaptivity, since, if faults propagate to the whole network, any recovery process would have to be global too.

Hereby we define two kinds of legitimate configurations: $\mathcal{TP}$-legitimate configuration are configurations that are legitimate enough for the algorithm to satisfy immediately its specification (the *output* variables, *i.e.* the variables that are involved in the problem specification, are correct), while the fully legitimate configurations (or simply legitimate configurations) are those where *all* variables are correct. In the context of the Token Passing problem (denoted by $\mathcal{TP}$), the output variable of a node is a boolean that is true if the node holds a token and false otherwise. This boolean may be either an actual variable or a predicate of the node.

The correctness proof shows that starting from a $\mathcal{TP}$-legitimate configuration, the algorithm satisfies the token passing specification, and that a fully legitimate configuration is reached after $2n$ rounds. The convergence proof shows that starting from a fully legitimate configuration that suffered f memory corruptions, a $\mathcal{TP}$-legitimate configuration is reached in time $O(f)$.

4.1 Legitimate configurations

We hereby define legitimate configurations relatively to the token passing problem specification.

Notation 1 *A processor P such that $Token(P)$ is true is said to "hold a token".*

Definition 9 (Correct Distance) *If $T : C_1 \longrightarrow C_2$ is a transition, a* correct distance variable *is defined recursively:*

1. *If P holds a token in C_2, then $Distance(P)$ is correct in C_2 if it is equal to* 0.

2. *If P does not hold a token in C_2, then $Distance(P)$ is correct in C_2 if $Distance(P^-)$ is correct in C_1 and that $Distance_{C_2}(P)$ equals $Distance_{C_1}(P^-)$ plus one.*

Definition 10 ($\mathcal{TP}$-Legitimate configuration) *A configuration C is $\mathcal{TP}$-*legitimate *if the two following conditions are satisfied:* (i) *a single processor P holds a token, and* (ii) *any other processor Q has its variable $Counter(Q)$ equal to* 0.

Definition 11 (Fully legitimate configuration) *A configuration C is* fully legitimate *if the three following conditions are satisfied:* (i) *a single processor P holds a token,* (ii) *any other processor Q has its variable $Counter(Q)$ equal to* 0*, and (iii) all distance variables are correct.*

Definition 12 (Token passing) *A computation $\mathcal{E}$ satisfies the $\mathcal{TP}$ (Token passing) specification if the two following conditions are satisfied:* (i) *in any configuration of $\mathcal{E}$, exactly one processor holds a token, and* (ii) *in $\mathcal{E}$, every processor holds the token infinitely often.*

4.2 Proof of correctness

Theorem 2 *Any computation whose initial configuration is $\mathcal{TP}$-legitimate satisfies the $\mathcal{TP}$ specification.*

Lemma 1 *In any computation $\mathcal{E}$ whose initial configuration is $\mathcal{TP}$-legitimate, the number of processors holding a token never increases.*

Proof: Let $\mathcal{T} : C_1 \longrightarrow C_2$ be a transition such that C_1 is $\mathcal{TP}$-legitimate and let P be the processor holding the token in C_1. Processor Q holding a token is determined by the value of the two variables $Val(Q)$ and $Val(Q^-)$. But only rule $R1$ may modify the value of variable $Val(Q)$.

Since P is the only processor in C_1 that may execute $R1$, only P and P^+ (the two processors that are concerned by the modification of $Val(P)$) may acquire or loose a token. Thus at most two tokens are present in C_2. When executing $R1$, P modifies its $Val(P)$ variable and looses its token, which means that P does not hold any token in C_2. In turn, starting from any $\mathcal{TP}$-legitimate configuration, the overall number of processors holding a token never increases. □

We notice that starting from any configuration, there exists at least one processor that hold a token.

Lemma 2 *In any configuration, at least one processor holds a token.*

Proof: Let $P_0, P_1, \ldots, P_n$ denote the successive processors of our n-sized ring (with $P_0 = P_n$). Assume that no processor hold a token, this means that for any P_i (with $i > 0$), $Val(P_i) \cong Val(P_{i+1}) - i \ (mod\ SND(n))$. By induction, we get $Val(P_0) \cong Val(P_i) - i \ (mod\ SND(n))$. When $i = n$, we obtain $Val(P_0) \cong Val(P_n) - n \ (mod\ SND(n))$. This is impossible since $P_0 = P_n$. □

Lemma 3 *Let $\mathcal{E}$ be a computation and let C_1 be a legitimate configuration in $\mathcal{E}$. A processor P that holds a token in C_1 may not keep it more than $2n$ rounds.*

Proof: We consider the $2n+1$ configurations following C_0 in $\mathcal{E}$. If predicate $Ready(P)$ is true in any of those configurations, P would apply $R1$ and transmit its token to P^+.

Assume that $Ready(P)$ is false in any of the C_i configurations following C_0 (with $i \in [1..2n+1]$). Then in each round, P would apply $R2$ and increment its $Counter(P)$ variable. Thus in C_{2n+1}, we would have $Counter(P) \geq 2n+1$.

Since $\frac{2n}{Distance(P^-)+1}$ is bounded by $2n$, the predicate $Ready(P)$ is satisfied in C_{2n+1}, which contradicts our hypothesis. □

Theorem 2 is then deduced from the following argument: *(i)* in any configuration of $\mathcal{E}$ (that starts from a legitimate configuration) there is exactly one token (Lemmas 1 and 2), *(ii)* a processor may not hold a token more than $2n$ rounds (Lemma 3), thus every processor is ensured to receive a token infinitely often.

4.3 Proof of convergence

In this section, we assume that there are no more than k memory corruptions, where $k \leq \frac{n-1}{8}$. Moreover, we consider the more difficult case of a scattered corruption that consists of x blocks of contiguous corrupted processors. A corrupted processor has at least one corrupted variable (either the Val, $Distance$, or $Counter$ variable).

Definition 13 (Tokens) *Let C be a non-legitimate configuration obtained from legitimate configuration L. Let P_0 be the processor holding a token in L. We distinguish in C between three kinds of tokens:*

1. *P has a* False Token *if $Val(P^-)$ is corrupted (like token T_2 in Figure 3.b).*
2. *P has a* Corrector Token *if $Val(P^-)$ is not corrupted (like T_1 in Figure 3.b).*
3. *The* True Token *is:*
   - *(a) The token that P_0 holds if P_0^- is not corrupted (like P in figure 3.b).*
   - *(b) The nearest Corrector Token on the ring before P_0 if P_0^- is corrupted.*

We assume without loss of generality that in each of the x blocks of contiguous corrupted processors, there is exactly one false token per corrector token. Indeed, the case where there are several false tokens for one corrector token is easier to deal with, since the speed of the false token is lower, and they are more quickly catched by the corrector token.

Notation 2 *Let $\mathcal{E}$ be an execution, C_0 be its initial configuration and $T_{F1}, \ldots, T_{Fx}$ be x False Tokens. Each False Token has a Corrector Token as token predecessor. We note it T_{Ji}*[1]*.*

We now introduce several formal definitions that will be used throughout the proof: We denote by $f_i(t)$ the number of corrupted processors in block i in the configuration C_t, that is the distance between T_{Ji} and T_{Fi}. $c_i(t)$ is the number of (non corrupted) processors between the false token T_{Fi} and the next corrector token T_{Ji+1}. v_{Fi} and v_{Ji} are the speeds of the false token T_{Fi} and the corrector token T_{Ji}, respectively. Note that a speed of *e.g.* $1/4$ means that the token moves every three rounds.

[1] T_{Ji} and not T_{Ci} to avoid any confusion between T_{Ci} and C_i.

Definition 14 *Let C_t be a configuration of $\mathcal{E}$ and let T_{Ji}, T_{Fi} be two tokens. Then we set :*

$$f_i(t) = \begin{cases} Dist(T_{Ji}, T_{Fi}) & \text{If } Distance(T_{Fi}) \text{ is correct} \\ 2 \times Dist(T_{J_i}, T_{F_i}) \text{ in } C_0 & \text{otherwise} \end{cases}$$

$$c_i(t) = Min\{Dist(T_{Fi-1}, T_{Ji}), n\}$$

$$v_{Fi}(t) = \begin{cases} Distance(T_{Fi}^{-})/2n & \text{If } Distance(T_{Fi}) \text{ is correct} \\ 0 & \text{otherwise} \end{cases}$$

$$v_{Ji}(t) = \begin{cases} Distance(T_{Ji}^{-})/2n & \text{If } Distance(T_{Ji}) \text{ is correct} \\ 0 & \text{otherwise} \end{cases}$$

We now show that within $O(f)$ rounds, the distance variables get corrected and upper bounded, whatever the initial values of the processor variables may be.

Lemma 4 *After $2f$ rounds, the $2f$ successors of T_F have a Distance variable that is correct and lower than $2f$.*

Proof: After $2f$ rounds, the $2f$ successors of T_F have correct distance information, and the T_F may only have moved by f positions (if its speed is maximal and equal to 1/2). □

The two following technical lemmas involve properties on sums of processors that will be corrected vs. sums of processors that will be corrupted. If the number of corrupted processors is small, then the average speed of the false tokens is low and the number of corrupted processors does not increase (indeed, a small number of corrupted processors implies a large number of correct processors, so that the average speed of the corrector tokens is high).

Lemma 5 shows that if the sum of the false token speeds is small, then only few extra processors will be corrupted.

Lemma 5 *Let I be the set of all the indices of false tokens. If for any round j between 0 and $t-1$, $\sum_{i \in I} v_{Fi}(j) \leq \alpha$, then $\sum_{i \in I} f_i(t) \leq f + t\alpha$.*

Proof: We wish to find an upper bound for $\sum_{i \in I} f_i(t)$. For that, we notice that a Token with speed v can move only once every $\frac{1}{v} + 1$ rounds. From that, we get:

$$\begin{array}{rcccc} \sum_{i \in I} f_i(t) & \leq & \sum_{i \in I} f_i(0) & + & \sum_{i \in I} Int[(t-1) \times v_{Fi}(t-1)] \\ & \leq & \sum_{i \in I} f_i(0) & + & \sum_{i \in I} (t-1) \times v_{Fi}(t-1) \\ & \leq & \sum_{i \in I} f_i(0) & + & (t-1) \sum_{i \in I} v_{Fi}(t-1) \\ & \leq & \sum_{i \in I} f_i(0) & + & (t-1)\alpha \\ & \leq & f & + & t\alpha \end{array}$$

□

Lemma 6 *Let I be the set of all the indices of false tokens. After $2f$ rounds, the execution reaches a configuration C_{2f} where the three following assertions hold: (i) there are at most $3f$ corrupted nodes (ii) all distance variables of corrupted nodes are correct and (iii) $\sum_{i \in I} v_{Fi}(f) \leq 1/4$ holds.*

Proof: Assertion *(ii)* is due to Lemma 5. For assertions *(i)* and *(iii)*, we prove by recurrence that for any t ($0 \leq t \leq f-1$):

(a) In configuration C_t, we have $\sum_{i \in I} v_{Fi}(t) \leq 1/4$

(b) In configuration C_{t+1}, we have $\sum_{i \in I} f_i(t) \leq 2f + t/4$

1. Base case ($t = 0$):

   (a) In configuration C_0, $\sum_{i \in I} v_{Fi}(t) = 0$ since none of the $v_i(t)$ are correct.

   (b) In configuration C_1, $\sum_{i \in I} f_i(t) = f$ since no corrupted token could move.

2. Assume the property is true for $1, 2, ..., t-1$ (with $(0 \leq t \leq f-2)$, and let's prove that the property also hold for t:

   (a) In configuration C_t, the number of corrupted nodes $\sum_{i \in I} f_i(t)$ is lower than $2f + t/4 \leq 2.25f$ (by recurrence hypothesis). Let $I_C = \{i \in I : \mathit{Distance}(T^-_{Fi})$ is correct $\}$. We wish to find an upper bound for the sum of speeds:

$$\begin{array}{rcc} \sum_{i \in I} v_{Fi}(t) & = & \sum_{i \in I_C} \mathit{Distance}(T^-_{Fi})(t)/2n \\ & \leq & \sum_{i \in I_C} 2f_i(t)/2n \\ & \leq & 4.5f/2n \\ & \leq & 1/4 \end{array}$$

   (b) In configuration C_{t+1}, we wish to find an upper bound on the number of corrupted nodes.

$$\begin{array}{rcccc} \sum_{i \in I} f_i(t+1) & = & \sum_{i \in I_C} f_i(t+1) & + & \sum_{i \in I \setminus I_C} f_i(t) \\ & \leq & \sum_{i \in I_C} f_i(0) + (t+1)/4 & + & 2\sum_{i \in I \setminus I_C} f_i(0) \\ & \leq & 2\sum_{i \in I_C} f_i(0) + (t+1)/4 & + & 2\sum_{i \in I \setminus I_C} f_i(0) \\ & \leq & \multicolumn{3}{c}{2\sum_{i \in I_C} f_i(0) + (t+1)/4} \\ & \leq & \multicolumn{3}{c}{2f + (t+1)/4} \end{array}$$

□

The following lemma captures the fact that if the sum of speeds of corrupting tokens is low enough, then the correction may occur in a time adaptive way, and be proportional to f.

Lemma 7 *Let I be the set of all the indices of false tokens. Let $\mathcal{E}$ be a computation whose initial configuration includes $3f$ corrupted nodes having correct distance variables, and such that $\sum_{i \in I} v_{Fi}(0) \leq 1/4$. Then after $16f$ rounds, the computation has reached a $\mathcal{TP}$-legitimate configuration.*

Proof: We prove by recurrence that for any t $(0 \leq t \leq f-1)$:

(a) In configuration C_t, we have $\sum_{i \in I} v_{Fi}(t) \leq 1/4$

(b) In configuration C_{t+1}, we have $\sum_{i \in I} f_i(t) \leq 4f - t/4$

1. Base case ($t = 0$):

   (a) In configuration C_0, $\sum_{i \in I} v_{Fi}(t) = 1/4$ by the definition of C_0.

   (b) In configuration C_1, $\sum_{i\in I} f_i(t) = 3f + x$ since no more than x Corrupted Tokens (*i.e.* all Corrupted Tokens) could move between C_0 and C_1.

2. Assume the property is true for $1, 2, ..., t-1$ (with $(0 \leq t \leq f-2)$, and let's prove that the property also hold for t:

   (a) In configuration, C_t, the number of corrupted nodes $\sum_{i\in I} f_i(t)$ is lower than $4f$ (by recurrence hypothesis). We wish to find an upper bound for the sum of speeds:

$$\begin{array}{rcc} \sum_{i\in I} v_{Fi}(t) & = & \sum_{i\in I} Distance(T^{-}_{Fi})(t)/(2n) \\ & \leq & \sum_{i\in I} 2f_i(t)/(2n) \\ & \leq & 8f/(2n) \\ & \leq & 1/4 \end{array}$$

   (b) In configuration C_{t+1}, we wish to find an upper bound on the number of corrupted nodes.

$$\begin{array}{rccccccc} \sum_{i\in I} f_i(t+1) & = & \sum_{i\in I} f_i(0) & + & \sum_{i\in I} Int[t\times v_{Fi}(t)] & - & \sum_{i\in I} Int[t\times v_{Ji}(t)] \\ & \leq & 3f & + & t\sum_{i\in I} v_{Fi}(t) & - & t\sum_{i\in I}(v_{Ji}(t)-1) \\ & \leq & 3f & + & t/4 & - & t/2 + \sum_{i\in I} 1 \\ & \leq & & & \text{4f - t/4} & & \end{array}$$

Overall after $t = 4 \times 4 = 16f$ rounds, the computation has reached a configuration where no processor is corrupted. □

We are now ready to prove the convergence property of our algorithm.

Theorem 3 *Algorithm 3.1 converges within* $18f$ *rounds provided that the number of faults* f *is lower than* $k \leq \frac{n-1}{8}$.

Proof: Let $\mathcal{E}$ be an execution of Algorithm 3.1. After $2f$ rounds, $\mathcal{E}$ reaches a configuration in which at most $3f$ processors are corrupted and such that $\sum_{i\in I} v_{Fi}(t) \leq 1/4$ (see Lemma 6). After another $16f$ rounds, $\mathcal{E}$ reaches a $\mathcal{TP}$-legitimate configuration (see Lemma 7). □

5 Conclusion

In this paper, we demonstrated the intrinsic interest of k-stabilizing algorithms over self-stabilizing ones for particular settings where impossibility results hold in the self-stabilizing case due to symmetry reasons. In more details, we presented the first uniform deterministic k-stabilizing algorithm for the token passing problem on unidirectional rings of any size. In addition to tolerating up to k faults (where $k \leq \frac{n-1}{8}$), our algorithm is also k-time adaptive, since it recovers from transient memory corruptions in $O(f)$ time. As captured by [15], fast stabilization time often comes at the expense of slower response time. In our case, when no faults occur in the system, the token speed is simply divided by two, while we can tolerate up to $\frac{n-1}{8}$ faults at a time.

Throughout the paper, we proved that k-stabilization (resp. k-time adaptivity) is strictly stronger than self-stabilization (resp. time adaptivity) in the sense that more problems can by solved by k-stabilizing (resp. k-time adaptive) algorithms than by self-stabilizing (resp. time adaptive) ones. There remains the orthogonal open question that is related to time-adaptivity: does there exist a problem that admits a self-stabilizing (resp. k-stabilizing) solution but no time adaptive (resp. k-time adaptive) one?

References

[1] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Symposium on theory of computing (STOC)*, pp. 82–93, 1980.

[2] 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.

[3] 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.

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

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

[6] J. Beauquier, M. Gradinariu, C. Johnen. Memory space requierements for self-stabilizing leader election protocols. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99)*, pp. 199-208, May 1999.

[7] 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.

[8] 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.

[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 and Sébastien Tixeuil. A Lower Bound on k-stabilization in Asynchronous Systems. In *Proc. of 21st Symposium on Reliable Distributed Systems (SRDS'2002)*, Osaka, Japan, October 2002.

[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] M. G. Gouda and M. Evangelist. Convergence/response tradeoffs in concurrent systems. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 288-292, 1990.

[16] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. of the 9th Annual ACM Symp. on Principles of Distributed Computing (PODC'90)*, pages 119-129, 1990.

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

[18] S. Kutten and D. Peleg. Fault-local distributed mending. J. Algorithms 30(1), pp. 144-165, 1999.

[19] S. Kutten and B. Patt-Shamir. Stabilizing Time-adaptive Protocols. Theoretical Computer Science 220(1):pp. 93-111, 1999.

[20] 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.

[21] 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.