Optimal Reactive k-Stabilization: the case of Mutual Exclusion

Joffroy Beauquier* Christophe Genolini† Shay Kutten‡

Abstract

Recently, it was suggested by multiple researchers that the smaller is the number of faults to hit a network, the faster should a network protocol recover. This goal proved hard, however, so such protocols have been suggested only for relatively easier (and less typical) cases, such as non-reactive tasks, or the case where a node can detect that it is faulty. We present solutions for the reactive problem that is often used as a benchmark for such protocols: the problem of *token passing*. We treat the realistic case, where no bound is known on the time a node can hold the token (a node holds the token as long as the node has not completed some external task). We study the scenario where up to k (for a given k) faults hit nodes in a *reactive asynchronous* 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.

We present several algorithms that stabilize into a legitimate configuration (in which exactly one node has a token) in time that depends only on k, and not on n (the number of nodes). We present our solutions in stages, by first presenting a basic protocol that stabilizes in $O(k^2)$ time and uses only a constant number of (logarithmic size) variables per node. For this first protocol it is required that k is smaller than $\sqrt{n}$, that is, the first protocol k-stabilizes, but does not self-stabilize. In terms of the number of individual nodes' steps the stabilization takes $O(kn)$ steps, and it is shown that any 1-stabilizing algorithm (that is, when $k = 1$) must use at least $n - 3$ steps.

The other algorithms are built on the basic one: one stabilizes in $O(k^2)$ time and is self-stabilizing (so k can be larger than $\sqrt{n}$), another enhanced version stabilizes in $O(k)$ time (and is time optimal) but the space it uses is larger by multiplicative factor of k.

1 Introduction

Traditionally, self stabilizing protocols, and fault tolerant protocols in general, were global in nature. That is, the recovery process covered the whole network even if only a few nodes failed (e.g. [5, 6, 7, 8, 13]). This approach does not scale to modern very large networks. To enable scaling, it was suggested by multiple researchers (e.g. [1, 4, 2]) that the smaller is the number of faults to hit a network, the faster should a network protocol recover. Designing such protocols (called *fault local*, or *time adaptive*) proved hard, however. Thus such protocols have been suggested only for relatively easier (and less typical) cases, such as the case where a faulty node can detect that it is faulty [9], or the case of non-reactive tasks (a distributed function computation that is performed once, and the result is not supposed to ever change).

Real systems, however, react to the outside world by changing their states when their inputs change. At first glance dealing with this case seems very problematic: assume that a new input is presented to some node, and then the new input is destroyed by a fault, before being copied by other nodes. How can one ever hope to perform anything that is meaningful to the user presenting the inputs?

In this paper we design fault local protocols for the reactive task that is the rather standard "benchmark" for self stabilization in in general. and, especially, for reactive systems. This is 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.

Note that this is a reactive system: a node P holds the token until some source, outside the system, eventually signals P, and P alone, that P can release the token. Let us comment that an alternative assumption, that other nodes are aware of the signal (e.g.

if it arrives after a fixed time), would have simplified the task considerably. However, this would not have been a realistic assumption, since, in reality, the signal models the completion of some local (mutual exclusion user) task at P.

Let a *configuration* (or a *global state* be a collection of the local states of the individual network nodes, and let $\mathcal{Q}$ be some predicate on a configuration. A *Self Stabilizing* ([3]) protocol Predicate $\mathcal{Q}$ is one that, when starting from an arbitrary configuration reaches, eventually, a *legitimate* configuration (a configuration for which $\mathcal{Q}$ holds) and remains in legitimate configurations henceforth.

Let *k-stabilizing* protocols be time adaptive 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 processes). That is, k-stabilizing protocol stabilizes in a time proportional to k and not to n.

To model a fault we use (see e.g. [13]) the Hamming distance between configurations. That is, let $\mathcal{C}_\infty$ and $\mathcal{C}_\in$ be configurations, the distance between them is the number of nodes whose local states are different in $\mathcal{C}_\infty$ versus $\mathcal{C}_\in$. Let $\mathcal{C}$ be an illegitimate configuration, and $\mathcal{L}$ be a legitimate one with the smallest distance from $\mathcal{C}$. If $\mathcal{L}$ is not unique then let $\mathcal{L}$ be any configuration with the smallest distance (this does not influence the analysis). The *number of faults* in $\mathcal{C}$ is the distance from $\mathcal{L}$. The *faulty nodes* are those whose state in $\mathcal{C}$ and in $\mathcal{L}$ are different.

New Techniques: Let us first mention the techniques used here, that we hope will be proven useful also for other reactive tasks as well. For 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. Techniques for bounding were shown in the past for non-reactive systems. However, bounding is more difficult in reactive systems, since such systems must still propagate the inputs.

Results: We use the two common time related complexity measures (for precise definitions see the full paper): (1) The sum (over all nodes i) of *steps*, where in one (atomic) step, node P reads a neighbor's state, computes, and writes P's variables. (2) *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. Note that there exist methods that can be used to translate our results to the weaker model of atomic read/atomic write [11, 6].

We present two k-stabilizing algorithms for token passing over an asynchronous ring of nodes (processes). The first stabilizes in $O(k^2)$ rounds, or $O(k^2 n)$ steps, using a constant number of variables per node (each of $O(\log n)$ bits).

The second algorithm can be viewed as a "parallelized" version of the first. Its round complexity is $O(k)$. Thus it is (asynchronous) time optimal. However, the space it uses is larger by a multiplicative factor of k.

We construct the solutions in stages, first we describe the version that is k stabilizing, but only for $k \leq \sqrt{n}$, and only if the number of faults is smaller than k. We then present additional components of the algorithms that make them self stabilize for any number of faults. Note that if there are more than k faults then the stabilization time of our algorithms is not better than that of Dijkstra's original algorithm.

On the negative side we show that any token passing self stabilizing algorithm requires at least $\Omega(n)$ steps, and thus its step complexity cannot be a function of k alone. (This also means that our algorithms are step optimal for a constant k.)

Related Work: The study of self-stabilizing protocols was initiated by Dijkstra [3]. *Reset-based* approaches to self-stabilization are described in [5, 6, 7, 8, 13, 17] In reset-based stabilization, the state is constantly monitored; if an error is detected, a special reset protocol is invoked, whose effect is to consistently establish a correct configuration, from which the system can resume normal operation (either some agreed upon configuration, or [13] a state that is in some sense "close" to the faulty state). One of the main drawbacks of this approach is that the detection mechanism triggers a system-wide reset in the face of the slightest inconsistency.

The papers most closely related to our work are [1, 4, 10, 9, 2]. In [1] the notion of fault locality was introduced, as well as an algorithm for the simple task, called the *persistent bit*, of recovering from a corruption of one bit at some k nodes, for an unknown k (and a generalization for a general input-output task) with output stabilization time $O(k \log n)$ for $f = O(n/\log n)$, Other tasks, in the same model, are solved in [4] in time $O(F(k))$, for sub linear func-

tions F. In [2] 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 algorithms in this case) and the stabilization time for the output is $O(k)$ (complete state stabilization takes $O(diameter)$ time, and this is shown to be optimal). These algorithms are for synchronous networks. An asynchronous, and self stabilizing version of [2] is described in [12]. In [10], 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 whose output stabilization time is $O(1)$ if $k = 1$. The transformed protocol has $O(T \cdot diameter)$ state stabilization time, where T is the stabilization time of the original protocol (no analysis is provided for output stabilization time when $k > 1$). The protocol of [10] 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 [9] 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.

Paper Structure Section 2 presents the model, Section 3 contains a short description of the original algorithm of Dijkstra, and points at the specific points in the algorithm that can be used to make it k-stabilizing. Section 4 includes the description of the first algorithm: first an overview, then a semi formal code, and, finally, the main idea of the proof and an evaluation of the complexity. Section 5 presents an "accelerated" version of the basic algorithm, with the optimal asynchronous time (round) complexity. Section 6 contains the enhanced versions of the algorithms that make them self stabilizing. Section 7 contains the proof that the step complexity of any algorithm must depend on n.

2 Basic definitions

2.1 Self-stabilization

Self-stabilizing algorithms as well as k stabilizing algorithms are modeled as transition systems. The definitions of a transition system, a configuration, an execution and the stabilization time can be found in [18]. The definition of k stabilization is rather similar to that of self-stabilization. The formal definitions are deferred to the full paper- for this extended abstract let us make do with the informal "definitions" brought in the introduction.

3 The intuition behind the basic protocol

3.1 The underlying Dijkstra's algorithm

The basic algorithm can be presented as an addition of k-stabilization components to one of Dijkstra's algorithms introduced in his pioneering paper [3]. There, the tokens circulate on an unidirectional ring with one distinguished process D (*the leader*). Each process has a variable Val in the range (n is the size of the ring). The leader is said to have a token if its variable value is equal to the value of its predecessor; a process other than the leader has a token if its value is different from the value of its predecessor.

Dijkstra's protocol is given in Figure 1 using guarded rules (a rule is *enabled* for execution if the guard holds). The addition operation is modulo $n+1$. For any node executing a rule, Val is its own value, while Val^- is the Val of the node's predecessor. Note that possessing a token is equivalent to having an enabled rule, and that the application of a rule causes the applying node to loose the token. If the global configuration is legitimate, then this application also causes the next node to start possessing the token.

A figurative way to visualize it is given in Figure 2, where the ring network is drawn as a tower, and a higher Val value of a node is depicted by a higher wall (e.g. the Val of D in that figure is higher than the Val of nodes to D's left). The node marked by T has the token. Note that in the tower on the right (the same ring after one step) the token moved to the next node.

For an insight that can be gained from this visualization note that any collection of consecutive faults actually manifests itself as sections of the tower walls that are either higher, or lower, than the rest of the wall (see the top left tower and the bottom right tower in Figure 3).

Distinguished	$Val^- = Val$	$\longrightarrow$	$Val = Val^- + 1$
Others	$Val^- \neq Val$	$\longrightarrow$	$Val = Val^-$

Figure 1 : Dijkstra first algorithm

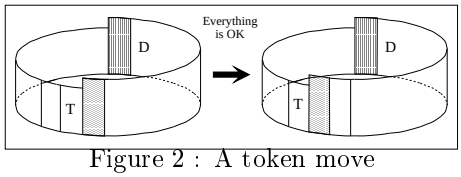

Figure 2 : A token move

3.2 Changing Dijkstra's protocol

We now demonstrate two kinds of schedules for Dijkstra's algorithm. (1) An "unlucky" schedule, where even one fault may propagate to the whole ring, and (2) a "lucky schedule" where the fault is corrected without any special fault correction actions been taken. Intuitively, our first algorithm (presented later) suppresses the "unlucky" schedule, using our propagation bounding components.

The case of a ring with one fault is visualized in Figure 3 by the top left tower. Note that there are up to three tokens, marked T_T ("true token"), F_T ("faulty token"), and V_T ("virtual token"). (Here V_T is at the node where the fault occurs.) Since, in Dijkstra's algorithm, only a node with a token may move, there are only three alternative possible configurations resulting from the next step:

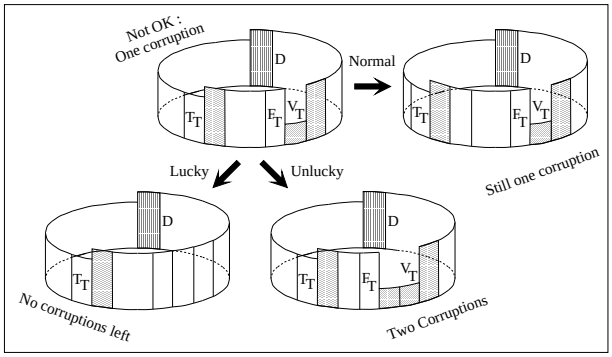

Figure 3 : Example, a single fault

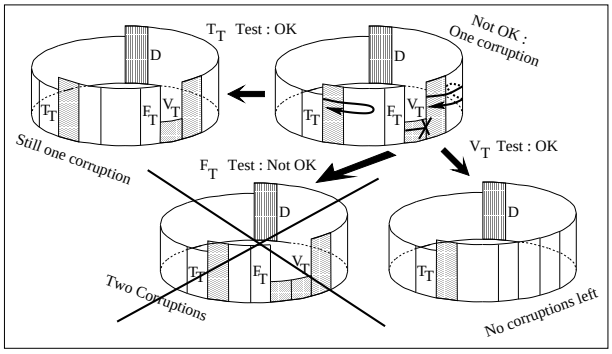

Figure 4 : Test example

- A non corrupted process Q whose predecessor's value has not been corrupted applies a rule (to have an enabled rule Q must possess a token). The new configuration of the ring is visualized by the top right tower of Figure 3. The number of tokens stays the same. The token that moved is called the *true token* and is unique.
- A corrupted process Q whose predecessor's value has not been corrupted (note this implies that Q has a token) applies a rule. This results in Q's Val being assigned the uncorrupted value of the predecessor (see the bottom left tower of Figure 3). Note that the number of corrupted nodes in the resulting configuration is smaller (by one) than the number of faults before the step. We say that the token that moved in this case was the *virtual token*.
- The non corrupted successor Q of a corrupted process (note that this implies that Q has a token) applies a rule (see the bottom right tower of Figure 3) Doing that, Q assigns (to Q's Val) the **corrupted** Val of Q's predecessor. In this case the number of corrupted processes has increased by one. We say that the *false token* moved.

Note that a "lucky" schedule (the second case) reduces the Hamming distance from a legitimate configuration, while an "unlucky" schedule (the third case) propagates the fault. In particular, if we are so lucky that a virtual token reaches a node already occupied by a false token, then at both tokens disappears! (In the visualized representation of the tower, the hole in the wall gets filled when the back of the hole reaches the front of the hole.) Our basic algorithm introduces a component that, in effect, "disables" the unlucky schedule.

The crucial observation here is

Observation 3.1 *if Node P has a false token then the previous token (a virtual token) is at distance no larger than $k+1$ from P.*

(In the visualized description of the tower, the segment of the wall that is outstandingly high, or low, is of length at most k- the number of faults.) In other words, faults do not introduce *one* new token, but, rather, at least *two*, that are not too far apart.

Thus P can detect faults by initiating the following distributed *Test procedure*, before passing the token on: $Test_P$ returns the value true $\Longleftrightarrow$ no faults were detected by $Test_P$ at the $k+1$ predecessors of P. We show, in the sequel, that if $Test_P$ is initiated and no additional faults occur (until $Test_P$ terminates) then the evaluation of $Test_P$ is correct. Figure 4 illustrates the results of the tests performed by the three different kinds of tokens discussed above:

- P has a true token. Then its predecessors positively evaluate $Test_P$ and allow P to pass its token (top left tower in figure 4).

- P has a virtual token. Then its predecessors positively evaluate $Test_P$ and allows P to pass its token (bottom right tower in figure 4).
- P has a false token, meaning that at least one of the $k+1$ predecessors of P possesses a token. The execution of $Test_P$ is not allowed to proceed, so P cannot move (bottom left token).

Note that if the number of faults (plus 1) is not smaller than $\sqrt{n}$, and the faulty nodes are equally spaced on the ring, at distances $k+1$ from each other, then the basic algorithm can deadlock. The test of every node with a token, in this case, discovers a previous token, and thus fails.

4 The basic algorithm

The basic algorithm is intended for the case that $k+1 < \sqrt{n}$. We consider a bidirectional but oriented ring of size n. For a node P let P^+ denote P's successor and P^- denote P's predecessor. The distance between two processes $Dist(A, B)$ is the number of links on the route (that agrees with the orientation) from A to B. One of the nodes, say D, is distinguished. The algorithm is a set of guarded rules for each process in the ring. The guards depend on the process' own variables and on the variables of its two neighbors. The application of a rule can only modify the variables of the process applying the rule.

A protocol is a set of guarded rules for each process. A *transition* is the evaluation and the execution of one of the rules by a process. This is considered an atomic operation (central demon). The application of Rule R by the process P is denoted R_P.

Because of the shortage of space we omit here the pseudo-code. Informally, the algorithm contains two parts- one rule, R_0, to move the token, and 7 rules to perform the test. Rule R_0 at the leader [resp. a non-leader] is similar to the rule of Dijkstra's algorithm (Figure 1) for a leader [resp. non-leader]. This rule uses a variable Val, the meaning of which is similar to the meaning of the state in Dijkstra's algorithm.

The main difference between the algorithms lies in an additional condition in the guard of R_0: for the rule to be enabled (i.e., for the token to move) this additional condition, *Tested* checks the result of a test. The test (and Condition *Tested*) use two additional variables: *Activity*, and *Index*. $Activity \in \{w, q, a\}$ determines the state of the *Test* procedure. if $Activity_P = w$, then P is waiting, not currently involved in a test. If $Activity_P = q$, then P is carrying out a question (P is in *question state*), if $Activity_P = a$, then P is carrying out an answer (P is in an *answer state*). $Index \in [0..k+1]$ is used to record the distance between P and the process which started the test procedure P is currently participating in.

The test procedure Recall that this test is used by a node P, possessing a token, to check whether one of its $k+1$ predecessors has a token (and thus a fault is detected). Let a *locally correct* process Q be a process whose state, together with the states of its two neighbors, can be extended to a legal configuration (by adding possible values for the states of the rest of the nodes). Thus, intuitively, a *locally incorrect* node can detect that either itself is corrupted, or some other node is corrupted. Note that in a correct configuration, a node who possesses a token cannot receive a test originated by some other node (who supposedly also has a token). Thus a node with a token who receives such a test is not locally correct.

To perform the test, a process P (that has a token) sends a question message to its predecessor. Each node P of the first k nodes receiving the question forwards it to its predecessor if and only if P is locally correct and does not posses a token. If the $k+1^{th}$ predecessor receives the test and is locally correct, then it sends an positive answer, which is relayed back to P by the k first predecessors.

We now discuss the implementation of the test, using the variables *Activity* and *Index*. Node P (having a token) asks (by setting its variable *Activity* to q and its variable *Index* to $k+1$) P^- to confirm that P's predecessors do not have tokens. If there is no token at P^-, still,before giving the confirmation, P^- must check that none of its k predecessors has a token. Thus P^- asks (by setting *Activity* to q and *Index* to k) P^{--} to confirm P's token. If P^{--} agrees (that is, P^{--} does not have a token), it asks (by setting *Activity* to q and *Index* to $k-1$) P^{---} to confirm P's token and so on. When the test procedure arrives at $P^{(k+1)(-)}$ (that means that P^-, P^{--}, ..., $P^{(k)(-)}$ agree with P's token), $P^{(k+1)(-)}$ can agree (by setting *Activity* to a and *Index* to 1) or disagree- in the case that it has a token. Any predecessor who disagrees does nothing, thus blocking the test, and preventing the token at P from moving. If $P^{(k+1)(-)}$ does agree, an acknowledgment is send back to P (all the $P^{(i)(-)}$ will set *Activity* to a) and P receives the authorization to pass the token. The scenario described above for the token at P is now repeated for the token at P^+, then for P^{++}, and so forth.

The main difficulty arises from the fact that we want the test to stabilize. This is done by local checking (see e.g. [6, 7]- indeed rather like the checking of the *grant intervals* of [6]). A legal sequence of nodes

performing the test starts with a node who has the token. The (potentially empty) prefix of the sequence is composed of nodes in question states ($Activity$=q) with descending $Index$ (the $Index$ of the node holding a token is $k+1$). The suffix are nodes are either in wait states $(Activity, Index)=(w,0)$ or answer states. (In the latter case the index of a node should be k+1 minus the distance to the token holder. For example, if node P did not enter the question state, then P^- is not supposed to be in the answer state. So the entrance of node P to the question state is not enabled until its predecessor P^- is in a wait state $(Activity, Index)=(w,0)$; This will happen since $(w,0)$; is the state to which these variables are reseted in this case, since an illegal state is encountered locally by P^-.

4.1 Analysis

The next theorem presents the main result for the basic algorithm.

Theorem 4.1 *The basic algorithm is k-stabilizing for the mutual exclusion problem $\mathcal{ME}$. Moreover, a corrupted process cannot propagate its value to its successor during the stabilization phase.*

Definition 4.1 Token: *Let L be a legitimate configuration, and C an illegitimate configuration that can be obtained from L by corrupting at most k nodes. A process Q has a virtual token* if it has a token and Q^- is not corrupted. A process Q has a *false token* if it has a token and Q^- is corrupted. Let $\mathcal{P}$ be the set of the processes having virtual tokens. For every node A, $\mathcal{P}Pred(A)$ is the nearest predecessor of A in $\mathcal{P}$. Let P be the process having the token in Configuration L. The *true token* in C is in process P if P^- is not corrupted and is in Process $\mathcal{P}Pred(P)$ otherwise.

Potential functions Ψ: Let C be a configuration and P a processor that has a token. $\Psi_{Test}(C,P)$ is the number of *Test Rules* that the k+1 predecessors of P may apply before P moves. $\Psi_{Test}(C)$ is the sum of the $\Psi_{Test}(C,P)$ for all the processes that have tokens.

Potential function Φ: Let C be a configuration where P has the true token. $\Phi_{Corrupted}(C)$ is the the number of corrupted processes and $\Phi_{Distance}(C)$ is the distance between P and its nearest predecessor (with respect to the orientation). that has a token, if such a proper predecessor exists. Otherwise $\Phi_{Distance}=0$.

Then, let $\Phi(C)$ be the couple $(\Phi_{Corrupted}(C) + \Phi_{Distance}(C), \Psi_{Test})$. (We use the lexicographical order when comparing the potential of two configurations).

Intuitively, Ψ is the number of steps that can be taken before some token must move. The other field of Φ bounds the number of such token moves before stabilization.

Remark: The set of legitimate configurations $\mathcal{L}$ is the set of all the configurations L such that $\Phi_{Corrupted}(C) + \Phi_{Distance}(C)$ is zero.

4.2 Sketch of the proof

The proof has three parts: First we prove that the protocol is k-converging, then we show the non-propagation of values of corrupted processes, and, finally, we prove the correctness.

4.2.1 k-convergence

k-convergence is proved through a sequence of lemmas (the proofs are omitted).

Lemma 4.2 *Let C and C' be two configurations and $T=(C,Q_R,C')$ be a transition such that no token moves between C and C' ($R \neq R0$). Then $\Psi(C) > \Psi(C')$.*

Intuitively, this lemma is used later to prove that a token eventually moves. Similarly, the next lemma is used to show that the value of Potential Function Φ decreases when a token moves; recall that when this value reaches zero the configuration is legitimate.

Lemma 4.3 *Let C and C' be two configurations such that C' is an illegitimate configuration and $T=(C,R_P,C')$ is a transition. Then $\Phi(C) > \Phi(C')$.*

Lemma 4.4 *In any configuration with less than $\sqrt{n}-1$ corrupted nodes there is at least one process that can apply a rule.*

Finally, to prove the k-convergence, we use the following lemma :

Lemma 4.5 *Let $\mathcal{S}$ be a transition system, $\mathcal{L}$ a set of (legitimate) configurations and T a transition. Assume that*

- *There is no terminal configuration.*
- *There exists a norm function $\Phi : \mathcal{C} \rightarrow \mathbf{N}^+$ such that for each transition $T=(C,R_P,C')$, $\Phi(C)$ is strictly bigger than $\Phi(C')$ or C' is in $\mathcal{L}$ (that is C' is a legitimate configuration).*

Then L satisfies the k-convergence property.

There is no terminal configuration (by lemma 4.4) and Φ decreases (lemma 4.3). Thus L satisfies the k-convergence property.

4.2.2 Non propagation

Lemma 4.6 *Let $\mathcal{E}$ be an execution and C_0 be the initial configuration of $\mathcal{E}$. If in C_0, some processor P has a false token that is not in a question state (that is, $Activity_P \neq q$) then P will not transmit its false token during the stabilization phase.*

Lemma 4.7 *Let $\mathcal{E}$ be an execution and C_0 the initial configuration. Let Q be a process that does not have the true token in C_0, and is not corrupted. Then Q is not corrupted during the stabilization phase.*

Only the true token or a corrupted process may be initially in a question state. Thus Q does not become corrupted (by lemma 4.6) during the stabilization phase.

4.2.3 Correctness

Lemma 4.8 *In any execution that starts with a legitimate configuration, exactly one process has a token.*

The definition of a token implies (as in Dijkstra's algorithm) that there is always at least one token; moreover, one can verify that no rule creates a new token.

Lemma 4.9 *Every process has the privilege infinitely often.*

Lemma 4.2 implies that Rule $R0$ is applied infinitely often. Since this moves the token according to the orientation of the ring, the lemma follows.

4.3 Complexity

Space complexity : Each process has three variables, each with $\log k$ bit or less.

Stabilizing time (in steps) : By Lemma 4.3, and using the potential function one can show that the step complexity is $O(n+k)k$. (Recall that $k^2 < n$.) Note that the step complexity does depend on n. We show in the sequel that no algorithm can avoid that. However, the round complexity of our basic algorithm does not depend on n.

Stabilizing time (in round) : The worst case is $O(k^2)$.

The complexity of Dijkstra's algorithm: In Dijkstra's first algorithm, if there are k corrupted processes in the initial configuration, one can demonstrate a scenario for which the stabilization time is $\Omega(kn)$ steps and $\Omega(n)$ rounds. Thus the stabilization time (in rounds) of Dijkstra's algorithm depends on the size of the network, and not on the number of corruption.

5 Accelerated algorithm

In this section we transform the basic algorithm of Section 4 so that it becomes time optimal (in terms of rounds). This, however, is achieved at the expense of increasing the space and the size of the messages.

5.1 Intuition and informal description

A penalty of $O(k)$ rounds is paid by the algorithm of Section 4 for each application of the test procedure. That is, for each move of a token, a node P holding the token first checks with the $k+1$ previous nodes to verify that they do not possess the token. Intuitively, this can be saved, if these $k+1$ predecessors of P broadcast their value to P repeatedly. Thus, intuitively, when P wants to know whether any of them possesses a token, it suffices that P consults their broadcasted values P already received from them, instead of sending test messages to consult these nodes.

There are several obstacles in this direction. First, when a predecessor Q of P, at distance $k+1$ from P, receives a token, it takes some time until the broadcast of Q reaches P. Had P waited for $k+1$ time units for this broadcast, this would have slowed the token at P by a factor of $k+1$. It turns out that it is enough to have the token proceed at half the speed of the broadcast. Thus, by the time the token makes $k+1$ moves, starting at some node P_1 and reaching some node P_2, broadcasts from a distance $2k+2$ have reach P_2. Intuitively, at that time P_2 "knows" the states, as they were at $k+1$ time units ago, simultaneously at P_1 and at P_1's $2k+2-(k+1) = k+1$ closest predecessors. Thus P_2 can simulate the action of P_1. Indeed, there is a delay of $k+1$ (in time and space) in this simulation, but the cost of this delay is additive, not multiplicative, since P_2^+ can act ("on P_1^+'s behalf") already two time units later.

Recall, however, that we assume an asynchronous network, and the above description seems as if it assumes synchronous networks. However, the same effect of the broadcast moving twice as fast as the token is achieved in an asynchronous network using the innovative *power supply* method of [15].

A more severe problem results from the context of self stabilization: faults may corrupt the broadcast of the predecessor. For example, consider the scenario that Q broadcasts a value v_1; Q^+, the successor of Q, is corrupted, and forwards $v_2 \neq v_1$ as the value of Q. Thus Q^{++}, which is *not* corrupted, forwards a wrong value v_2 for Q, and so forth. Eventually P is led to evaluate $Test_p$ based on a wrong value for Q.

This problem, too, is solved using the *power supply* method, or the somewhat similar *regulated broadcasts*

method of [2]. For the sake of simplicity we outline the solution using the regulated broadcasts method, which requires the network to be synchronous. The method can be applied to asynchronous networks using the power supply method.

Every node broadcasts replicas of its value Val to distance $k+1$. For that purpose each node P has a variable $V_P[j]$ for P's jth predecessor among the nearest $2k+2$ predecessors (the number of predecessors tested here by $Test_P$ is double the number that is tested by the basic algorithm for technical reasons involving the proof of this algorithm by using it to simulate the basic algorithms). In the synchronous ring node P copies the replicas of the values for its $k+1$ predecessors from its immediate predecessor *every time unit*. (In [15, 2] the broadcasts are delayed to proceed every two time units.) However, the token proceeds every two time units, and then, only , if the test (performed on its own replicas of the values of its predecessors) allows it to proceed. (We note that, for the sake of simplicity only, we assume here that the even pulses occur at all the nodes at the same time; it is straightforward to get rid of this assumption: all that is really needed is that each node delays every broadcasted value for one time unit before enabling its successor to read it.) The test is similar to the test of the previous algorithm.

5.2 The algorithm

```
Every pulse do:
    (∀_{0<j≤(2k+2)})  V_P[j] ← V_P−[j−1]
    V_P[0] ← Val
Every 2nd pulse do also:
    If (∀_{0<j≤2k+2})  V_P[j] = V_P[j−1] ≠ V_P[0]
        then V_P[0] ← Val ← V_P−[0]
```

Figure 5 : Algorithm for a non-leader node

The full algorithm for an asynchronous network appears in the full paper. See the code of a non-leader node, for a simplified version of the algorithm, and for synchronous networks, in figure 5 (the code for the leader, the distinguished machine, is similar, except that (a) the condition for having the token is that Val is the same as that of the previous node, rather than different; and (b) the leader increments every predecessor value it copies in the first statement to compensate for (a)).

The formal analysis is deferred to the full paper.

6 Self Stabilization

6.1 Self-stabilizing protocol

We continue the stepwise development of the algorithms, by making them self stabilize (the self stabilization of the accelerated algorithm is presented in the full paper). Recall that the basic algorithm may not stabilize if k is larger than $\sqrt{n}-1$. The bad scenario for a large k is illustrated in Figure 6: all test procedures are deadlocked since every token has another token behind it, at a distance no larger than k.

Anti deadlock : A modular extension that overcomes such a deadlock involves the use of a deadlock detector component. Informally, whenever a deadlock is detected, one of the tokens is allowed to proceed one step. Note that the task of constructing the deadlock detector is made easier by the fact that the detector does not have to be k stabilizing since Theorem 4.1 implies that a deadlock is possible only when the number of faults is at least $\sqrt{n}$. Thus, if the detector is (only) self stabilizing, and its round complexity is some $O(F(n))$ for some function F, then its complexity is also $O(F(k^2))$.

Some care is still required, so that false tokens do not receive too many grants to proceed. It is enough that one of the deadlocked tokens is allowed to proceed. Thus output of the detector to the basic algorithm is done only at the distinguished node D. If the detector detects a deadlock, and D is in a question state (for $Test_P$ of some node P) then D sends a positive answer to the test, allowing the token at P to proceed once. The properties of $Test_P$ are used to show that only the false token that is the closest to D can repeatedly get such an exceptional grant to move.

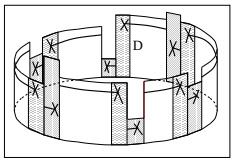

Figure 6: Deadlock

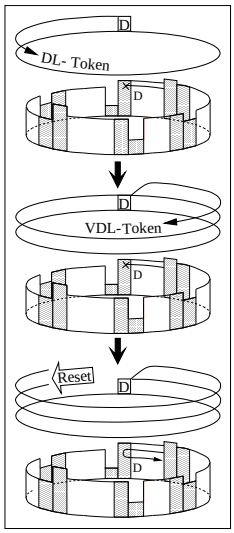

Figure 7: Anti Deadlock

The informal description bellow is given as if the

detector operates on a separate ring, the *Deadlock ring* (implemented using an additional variable at every node). It is composed of mechanisms to circulate two new kinds of tokens that we term *DL-token* (DeadLock token) and *VDL-token* (Verification). Both of these tokens are supposed to reside at D, at times when D does not "suspect" a deadlock, and to be released to walk the ring in opposite orientations when D does "suspect" a deadlock (see below). Since these token passing components are not really required to k-stabilize, just to self stabilize, their fault recovery is somewhat simpler than that of the basic algorithm. Thus, in this informal description we concentrate on the tasks these tokens perform while walking the ring, rather than on the way we overcome faults that disturb the walks of these tokens. The formal code of the detector (including the fault handling) is given in Figure 8.

Node D suspects a deadlock when it, as well as its successor and its predecessor, are in the question state. It then releases *DL-token* that walks *against* the orientation of the original ring (from P to P^-). Each node P passes DL-token to P^- only if both P and P^- are in the question state. Thus, if there is a deadlock on the original ring then the DL-Token eventually gets back to D.

To ensure stabilization it is nevertheless required to verify the deadlock (so that at most one cycle of detection, and then verification, can be misled by faults). For that, process D now releases the *VDL-Token* that walks the ring with the same orientation as the original ring (and against the orientation of the DL- token). A node P releases the VDL-token (to P^+) only if P is in a question state and if P passed the DL-Token (in the other direction) before.

When D receives the VDL-Token, it initiate a reset of the detector variables at all the nodes, and gives a positive answer to the $Test_P$ procedure that D currently participates in (that caused D to initiate the deadlock detector).

The protocol : The anti deadlock component accesses the *Activity* variable of the basic algorithms in order to communicate with the basic algorithm. The DL- Token and the VDL- Token are implemented using the new variable, $DLRing$, the values of which are in {0,1,2}. It determines the location of the DL-Token and the VDL-Token: a process P has a DL-Token if $DLRing_{P^-} = DLRing_P = 0$ and $DLRing_{P^+} = 1$. The passing on of the DL- token from P to P^- is performed by setting P's $DLRing$ to 1. P has a VDL-Token if $DLRing_P = DLRing_{P^+} = 1$ and $DLRing_{P^-} = 2$. The passing of the VDL-Token to P^+ is performed by setting P's $DLRing$ to 2.

Note that the combined algorithm (including both the basic algorithm, and the anti deadlock component) uses additional variables (those are used by the basic algorithm). (The pseudo code for the combine algorithm is deferred to the full paper.) One of those variables is $Index$, and the other is Val.

The combined algorithm is executed as follows: a node P evaluates the guards of the rules of *both* the basic algorithm and of figure 8 *before* applying the rules. Then, if the guards of a rule in figure 8, and a rule of the basic algorithm both hold, the node executes *both* corresponding rules.

The first four rules are used by the distinguished process D. The first rule is applied when D detects a potential deadlock locally and sends a DL-Token on the virtual ring to D^-. The second rule is applied when D receives a DL-Token and thus it seems that there is a deadlock. It then sends a VDL-Token on the virtual ring to D^+. The third rule is applied when D receives a VDL-Token: it gives a positive answer to the test (and initiate the resetting of the variables of the virtual ring to be ready for the next deadlock detection). The fourth rule is the detection of an inconsistent configuration of the deadlock detection algorithm (due to faults), and the initiation of the resetting of the virtual ring variables, to remove the effect of the faults on the deadlock detection algorithm. (Inconsistency is detected, for example, by the distinguished node on receiving a DL-Token without having already sent a VDL- Token.)

The three last rules are used by processes other than the distinguished one. The first rule is applied by P to transmit a DL-Token to P^-. The second rule is applied by P to transmit a VDL-Token to P^+. The third rule is is the detection of an inconsistency by P (or of a reset), and the initiation of a reset.

Theorem 6.1 *The algorithm obtained by adding the anti deadlock subroutine to the basic protocol is self-stabilizing. Its stabilizing time are $O(nk^2)$ steps and $O(k^2)$ rounds if there are initially less than k faults, $O(kn^2)$ steps and $O(kn)$ rounds otherwise.*

7 A negative result for step complexity

In this section, we prove that no algorithm can have its step complexity as a function of k alone, with no dependence on n. (Recall, however, that the *round* complexities of the k stabilization of our algorithms is only a function of k). In the full paper we prove the following theorem:

Rules for D :			
(*DL-Token*)	$(DL-Token, Activity) = (0,q)$ in D, D^- and D^+	$\longrightarrow$	$DLRing_D = 1$
(*VDL-Token*)	$(DL-Token, Activity) = (1,q)$ in D, D^- and D^+	$\longrightarrow$	$DLRing_D = 2$
(*Decision*)	$(DL-Token, Activity) = (2,q)$ in D, D^- and D^+	$\longrightarrow$	$DLRing_D = 0; Activity_D = a$
(*Reset*)	Otherwise	$\longrightarrow$	$DLRing_P = 0$
Rules for a non-distinguished process P:			
(*DL-Token*)	$(DL-Token, Activity) = (0,q)$ in P, P^- and P^+	$\longrightarrow$	$DLRing_P = 1$
(*VDL-Token*)	$(DL-Token, Activity) = (0,q)$ in P, P^- and P^+	$\longrightarrow$	$DLRing_P = 2$
(*Reset*)	Otherwise	$\longrightarrow$	$DLRing_P = 0$

Figure 8 : The Anti Deadlock Subroutine

Theorem 7.1 *Consider an 1-stabilizing algorithm on a bidirectional ring with a central demon. Then there exists an initial corrupted configuration $C'_0 \in \mathcal{L}_1$ and an execution $\mathcal{E}'$ such that after $(n-3)$ steps, the algorithm is not in a legitimate state.*

References

[1] S. Kutten and D. Peleg. Fault-local distributed mending. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Aug. 1995.

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

[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. ACM*, 17(11):643–644, November 1974.

[4] S. Kutten and D. peleg. Tight Fault Locality. In *36th Annual IEEE Symposium on Foundations of Computer Science*. Milwaukee, WI, USA, October 1995.

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

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

[7] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundation of Computer Science*, 268–277, 1991.

[8] 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*,

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

[10] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemamraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, Philadelphia, Pennsylvania, USA, May 1996.

[11] S. Dolev, A. Israeli, and S. Moran. Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. *Distributed Computing*, vol.7, pp.3-16,1993.

[12] S. Kutten and B. Patt-Shamir. Asynchronous Time-Adaptive Self Stabilization. a Brief Announcement in the proceedings of ACM PODC 1998.

[13] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. In *Proc. of the Second Workshop on Self-Stabilizing Systems*, pages 3.1–3.15, May 1995.

[14] Y. Afek, B. Awerbuch, S. A. Plotkin, and M. Saks. Local Management of a Global Resource in a Communication Network. In *Proceedings of the 28th FOCS* , October 1987.

[15] Y. Afek and A. Bremler . Self-Stabilizing Unidirectional Network Algorithms by Power-Supply. *Chicago Journal of Theoretical Computer Science*, to appear.

[16] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, pages 15–28, Italy, Sept. 1990. Springer-Verlag (LNCS 486). To appear in *Theoretical Comp. Sci.*

[17] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing, San Diego, California*, pages 652–661, May 1993.

[18] Gerard Tel. *Distributed Algorithms*. Cambridge University Press, 1954.