Optimal k-Stabilization: the case of synchronous Mutual Exclusion

Christophe GENOLINI
LRI-Universite Paris Sud, Batiment 490, F91405 ORSAY Cedex, France,
Christophe.Genolini@lri.fr

Key words : distributed algotithme, fault-tolerance, self-stabilization, time-adaptive

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 study the scenario where up to k (for a given k) faults hit nodes in 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.

We present an algorithm that stabilizes 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 self-stabilizes in $O(n)$ time and uses only a single variable (size one bit) per node. This first protocol self-stabilizes, but does not k-stabilize. Then we present a second algorithm built on the basic one that stabilizes in $O(k)$ rounds.

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. [?, ?, ?, ?, ?]). This approach does not scale to modern very large networks. To enable scaling, it was suggested by several researchers (e.g. [?, ?]) 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 [?], or the case of non-reactive tasks (a distributed function computation that is performed once, and the result is not supposed to ever change).

In this paper, we design fault local protocols for the reactive task that is the rather standard "benchmark" for self stabilization 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.

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* (introduced in [?] by Dijkstra) protocol Predicate $\mathcal{Q}$ 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.

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 the Hamming distance between configurations (see e.g. [?]). 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.

New Techniques: Let us first mention the techniques used here, that we hope will be proven useful also for other reactive tasks. 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 mainly 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 present two algorithms for token passing over an synchronous ring of nodes (processes). The first one (call basic algorithm) self-stabilizes in $O(n)$ rounds using a constant space per node (single variable of 1 bit).

The second one k-stabilizes in $O(k)$ rounds, and uses three variables (the same variable as the basic algorithm, and two aditionnal one in range $O(k)$).

Related Work: The study of self-stabilizing protocols was initiated by Dijkstra [?]. In [?], Gouda and Furman propose a stabilizing version of asynchronous token ring using only three bits. *Reset-based* approaches to self-stabilization are described in [?, ?, ?, ?, ?, ?] 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 [?] 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 [?, ?, ?, ?, ?]. In [?] 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 (and a generalization for a general input-output task) with output stabilization time $O(k \log n)$ for $f = O(n/\log n)$. In [?] 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 [?] is described in [?]. In [?], 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 $O(T \cdot diameter)$ step 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 [?] 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 [?] 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. In [?], an asynchronous k-stabilizing algorithm is presented for the token ring task. It stabilizes in $O(k^2)$ and its space overhead is $O(kn)$

2 Basic definitions

In this section we precise the definition of k-stabilization.

2.1 Self-stabilization

Self-stabilizing algorithms are modeled as transition systems, the start configuration of which can be arbitrary (including ones that are not reachable from some other states).

Definition 2.1 *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 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 2.2 *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 subset $\mathcal{L} \subseteq \mathcal{C}$ of legitimate configurations, with the following properties:*

- *Correctness: Every execution starting in a configuration in $\mathcal{L}$ satisfies $\mathcal{SP}$.*
- *Convergence: Every execution contains a configuration in $\mathcal{L}$.*

Application A common way to modelize a self-stabilizing algorithm running on a network is to set the following transition system :

- A configuration C is the set of the value of all the network processes.
- $\mathcal{C}$ is the set of all the possible configuration C.
- 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 processes.

More formally, let the (Hamming) distance $Dist(C_1, C_2)$ between two configurations C_1 and C_2 be the number of processes the states of which 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 2.3 *Let S be a system and $\mathcal{L}$ a set of configurations. The* ball of center $\mathcal{L}$ with radius k *is the set $Ball^k_{\mathcal{L}}$ of all the configurations C such that the Hamming distance between C and $\mathcal{L}$ is smaller or equal to k.*

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

Definition 2.4 *Let S be a system, C a configuration and $\mathcal{L}$ 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 processes $\mathcal{P}$ in C that are corrupted relatively to L is the set of processes that do not have the same value in C and in L.*

Given a configuration C, the notion of corrupted process 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 $\mathcal{L}$ that has a minimal distance from C. Thus, when the set $\mathcal{L}$ is known, we will use the term "corrupted processes", omitting the reference to L.

Definition 2.5 *A system S is k-stabilizing for a specification $\mathcal{S}$ if there exists a subset $\mathcal{L} \subseteq \mathcal{C}$ of legitimate configurations with the following properties:*

- *Correctness: Every execution starting in a configuration in $\mathcal{L}$ satisfies $\mathcal{S}$.*
- *k-Convergence: Every execution starting in $Ball^k_{\mathcal{L}}$ contains a configuration in $\mathcal{L}$.*

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

Note that a k-stabilizing system is not necessarily self-stabilizing.

2.3 Stabilization time

A way to measure the efficiency of self-stabilizing and k-stabilizing systems is to evaluate the number of rounds before it reached a legitimate state.

Definition 2.6 *Given a self-stabilizing or a k-stabilizing system and an execution, the stabilizing phase is the prefix of the execution 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 exists) of all the possible executions of the system.*

Note that a self-stabilizing system is also k-stabilizing. The aim for considering k-stabilizing systems rather than self-stabilizing ones is that the former, under the assumption of a small number of corrupted processes, has a shorter stabilization time.

3 The basic algorithm

In this paper, we present two solutions of the mutual exclusion problem on an oriented ring. We adopt most of the Dijkstra token ring convention [?]. We consider an oriented ring with a distinguished process. When a process satisfies some predicate, we say it has a token. By applying a guarded rule, it loses its token and transmits it to the next processor on the ring. The algorithm is stabilized when there is exactly one token on the ring.

When we want an algorithm to self-stabilize to mutual exclusion, the main difficulty is to make sure that if there are several tokens on the ring, some of them will disappear during the execution. For that, our first idea is to insure a permanent odd number of tokens on the ring. That is, if a process gets two tokens and makes both of them disappear, there will still be some other tokens left on the ring. The second idea is to let the tokens move with different velocity. Then some fast token will catch some slow token, and two by two, all the additional tokens will disappear.

3.1 Model & Protocol

Model: We consider an unidirectional ring with one distinguished process D. Each process has a variable Val that can take two values 0 or 1. If P is a process, P^- denotes the process just before P on the ring, and P^+ denotes its successor.

The algorithm is a set of guarded commands. Each rule is on the form $< Condition > \Longrightarrow < Action >$. Our model is the classical synchronous shared-memory distributed system. All the processes can read variables of their neighbors, and read/write their own variable. The transitions (also called rounds) are synchronous, which means that all the processes perform simultaneously the following sequence : 1) all the processes read

their variable and their neighbors variable; 2) all the processes evaluate conditions of the guarded rules; 3) all the actions whose condition is true are executed.

Token: The distinguished process D is said to have a *token* if its variable value $Val(D)$ is equal to the value of its predecessor; the other processes P have a token if their value $Val(P)$ are different from the value of their predecessor. If a process (distinguished or not) has a token T, its token will be called a *fast token* if $Val(P) = 0$ and a *slow token* if $Val(P) = 1$. If T is a token, the token successor of T is the next token encounter running the ring clockwise, its token predecessor is the token encounter running the ring counter-clockwise

A process P having a token may apply a rule and perform an action (call *transmission of the token*). That is, by changing the value of its variable, the process loses its token. If its successor P^+ has no token or has one but loses it, then it gets P's token. In this case, we say that *the token moves* (from a process P to its successor P^+).

The distance between two processes $Dist(P, P')$ is the number of processes (counting clockwise) between P (include) and P' (exclude).

A figurative way to visualize a state on the ring is given figure 1. The ring network is drawn as a small tower where each portion of the wall denotes a process. The value of a process $Val(P)$ is depicted by the higher wall if $Val(P) = 1$ and by a small wall if $Val(P) = 0$.

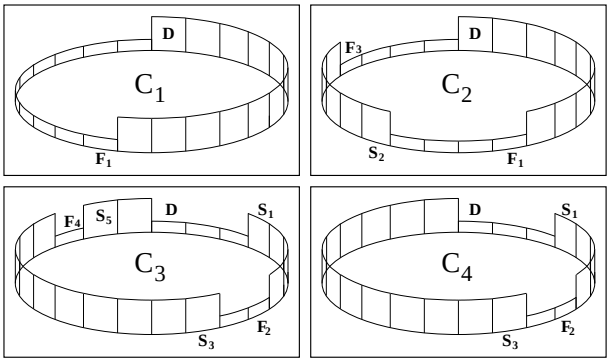


Figure 1: Four example of configurations

Definition 3.1 *The specification $\mathcal{ME}$ of mutual exclusion is :*

- *In every configuration, at most one process has a token.*
- *In any execution, every process has the token infinitely often.*

3.2 Algorithm

The protocol is given in figure 2. The idea of this protocol is based on the following remarks, that can be easily verified according to the rules :

Predicate :

$Token(P) \Leftrightarrow \begin{cases} Val(P^-) = Val(P) & \text{if } P \text{ is distinguish.} \\ Val(P^-) \neq Val(P) & \text{else.} \end{cases}$

$FastToken(P) \Leftrightarrow Val(P) = 0$

$SlowToken(P) \Leftrightarrow Val(P) = 1$

Action :

$TransmitT(P) \quad \Leftrightarrow Val(P) \longleftarrow Val(P) + 1 \ \ (modulo 2)$

Rules :

Every pulse : P has a fast token $\Longrightarrow TransmitT(P)$

Every two pulses : P has a slow token $\Longrightarrow TransmitT(P)$

Figure 2: The basic Algorithm

- Since the variable Val that determines the existence of a token has only two possible values, there is always an odd number of tokens on the ring (see figure 1).
- If we consider the sequence of the tokens, numbering clockwise and starting by the closest token following D, there is an alternation of fast token and slow token (see figure 1, C_2, C_3 and C_4).
- When a token reaches D, its nature (fast or slow) changes.
- Let P a process such that both P and its successor P^+ has a token. If P transmits its token and P^+ does not transmit its token, then both tokens are removed (if in C_3, the token F_4 moves and the token S_5 does not move, both of F_4 and S_5 disappear, as we can see in C_4).

Then the idea is to let the fast tokens move at every round, and the slow tokens move every two rounds.

Theorem 3.2 *The basic algorithme given figure 2 self-stabilizes to the mutual exclusion $\mathcal{ME}$.*

Intuitively, if there is more than one token on the ring, the fast token will catch the slow one, and both will disappear. This will be repeated until there will be only a single token on the ring.

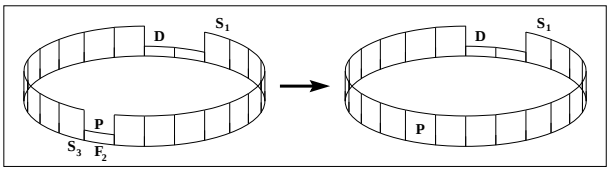
4 The k-stabilizing algorithm

The k-stabilizing algorithm can be viewed as an addition of k-stabilizing components to the basic algorithm.

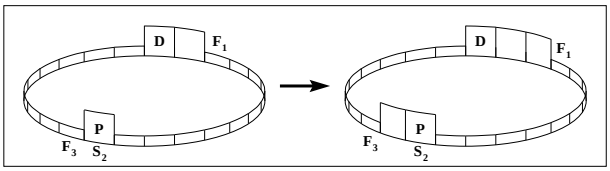
4.1 Changing the basic algorithm

Consider a configuration C obtained by corrupting a single process P in a legitimate configuration. Because of the corruption, two tokens appear, one in P, the other in P^+. Two cases are possible:

- A "lucky" case, where P^+ has a slow token and P has a fast token. Then, after one or two rounds, the fast token catches up with the slow token and the execution reaches a legitimate state.
- An "unlucky" case, where P has a slow token and P^+ has a fast token. Then, few rounds later, the fast token has moved faster than the slow one and has let several corrupted processes behind it.



(a) A lucky example



(b) An unlucky example

Figure 3: A single fault

In example 3.(a), the corruption of P is removed after one round. In example 3.(b), the corruption of P is not removed. Worst, P's successor is also corrupted.

In the lucky case, the fast token runs around the ring, correcting the corruption in a very short time. In the unlucky case, the fast token propagates corruption. The idea of our k-stabilizing algorithm is to let the fast token F_2 move only in the lucky case. This is possible, when the number of faults k is small, by checking if the token is closer from its token successor S_3 than its token predecessor S_1. So the lucky case will be as before, but not the unlucky one. For that, the token move will be submitted to the additional condition: the distance between F_2 and S_3 must be smaller than the distance between S_1 and F_2. Therefore, the lucky case will be as before, but in the unlucky case, the token will not move.

In order to implement this additional condition, each process P has two additional variables. $DistPred(P)$ is the distance between P's closest predecessor having a token and P. The second one, $DistSucc(P)$, is the distance between P and its closest successor having a token. Both of them are in range $\{0, .., 4k\}$. We call them *distance indice of* P or *indice of* P.

The circulation of the information has to be faster that the token, because a token could have a false information that will possibly never be corrected. Then the reset of the two indices is faster than any token move (every round for the information, every two rounds for the fast token, and every four rounds for the slow). This ensures that after some time, the token has a correct information on the distance of its token predecessor and token successor.

Anyway, the propagation of the information takes time and when a process P has its $DistPred$ variable to i (resp. $DistSucc(P) = j$), that means that i rounds before, the previous token T_1 was at distance i of P (resp. j rounds before, $Dist(P, T_3) = j$). But in i rounds, T_1 and T_3 may have moved.

More precisely, if P has a fast token F_2, we know that T_1 and T_3 are slow tokens. So they move once every four rounds and the actual distance between P and T_1 is approximatively $i - i/4$ (resp. $Dist(P, T_3) \simeq i + i/4$). So, a process P will transmit its fast token if $DistSucc(P) \times 1,25$ is smaller or equal to $DistPred(P) \times 0,75$. In this case, we say that P has a *living token*, or that F_2 is alive. In the opposite case, if the condition is false, P will not transmit its token, that will be called a *sleeping token*.

4.2 The algorithm

Predicate :

$$Token(P) \Leftrightarrow \begin{cases} Val(P^-) = Val(P) & \text{if } P \text{ is distinguesh} \\ Val(P^-) \neq Val(P) & \text{else} \end{cases}$$

$PredToken(P) \Leftrightarrow Token(P^+)$

$FastToken(P) \Leftrightarrow Token(P)$ and $Val(P) = 0$

$SlowToken(P) \Leftrightarrow Token(P)$ and $Val(P) = 1$

$$Alive(P) \Leftrightarrow \begin{cases} DistS(P) \times 1,25 \leq DistP(P) \times 0,75 \\ \text{or } DistP(P) = 4k \end{cases}$$

Action :

$TransmitT(P) \Leftrightarrow Val(P) \longleftarrow Val(P) + 1 \ \ (modulo 2)$

$$ActualisePred(P) \Leftrightarrow \begin{cases} DistP(P) \leftarrow 0 \text{ if } P \text{ has a token.} \\ DistP(P) \leftarrow DistP(P^-) + 1 \text{ else} \end{cases}$$

$$ActualiseSucc(P) \Leftrightarrow \begin{cases} DistS(P) \leftarrow 0 \text{ if } P \text{ has a token.} \\ DistS(P) \leftarrow 1 \text{ if } P^+ \text{ has a token.} \\ DistS(P) \leftarrow DistS(P^-) + 1 \text{ else} \end{cases}$$

$ActualiseDist(P) \Leftrightarrow ActualisePred(P)$ and $ActualiseSucc(P)$

Rules :

Every pulses : $\Longrightarrow ActualiseDist(P)$

Every two pulses : $FastToken(P) \wedge Alive(P) \Longrightarrow TransmitT(P)$

Every four pulses : $SlowToken(P) \Longrightarrow TransmitT(P)$

Figure 4: The k-stabilizing Algorithm

The k-stabilizing algorithm is given figure 4. The predicate $Token(P)$ is true if P has a token. The predicate $FastToken(P)$ (resp. $SlowToken(P)$) is true if P has a fast (resp. slow) token. The predicate $PredToken(P)$ is true if P's successor has a token.

The condition $Alive(P)$ is true if P's token successor is closer than P's token predecessor, or if P has no token predecessor at a short distance (if there is k corrupted processes in the initial configuration, no token in the $4k$ predecessors of a process holding a token T means that T is the only token on the ring).

By performing the action $TransmitToken(P)$, the process P changes the value of its variable $Val(P)$. Doing this, P loses its token. By performing $ActualisePred(P)$ (resp. $ActualiseSucc(P)$), P actualizes its distance variable $DistPred(P)$ (resp. $DistSucc(P)$).

The actualization of the indice variable is done every round, the transmission of the token is done every two rounds for the fast tokens, and every four rounds for the slow tokens.

4.3 Complexity

Theorem 4.1 *The complexity of the k-stabilizing algorithm (figure 4) is $O(log(k))$ bits in space and $O(k)$ rounds in time.*

Definition 4.2 *Let C be an illegitimate configuration with no more than k corrupted processes and L a legitimate configuration such that the Hamming distance between C and L is minimum. Then, P has a false token in C if P has a token in C due to the corruption of the principal value of its predecessor $Val(P^-)$.*

If P has not a false token, we say that P has a true token.

Note that a false token T_2 always has a true token T_1 as token predecessor. Moreover, all the processes between a false token and its token predecessor are corrupted. We call them the *trail of* T_2. The length of T_2's trail is the number of processes between T_2 and T_1.

There is mainly two problems for proving fast convergence: in the initial configuration, some indices can be corrupted. So some fast token that should be alive are sleeping, and conversely. That means that some false token might move and increase the size of their trail (that is, they increase the total number of corrupted processes on the ring).

The other problem is that if the corrupted processes are not chained, a false token might be closer from its token successor than its token predecessor. Once more, this token will propagate its corrupted value. So the proof of the theorem consist mainly to bound the number of additional corrupted value that can appear on the ring.

5 Conclusion

In this paper, we have presented two synchronous self-stabilizing token ring algorithms. The first one self-stabilizes in time proportional to the ring size, and is optimal in space. In [?], M. Gouda and F. Furman Haddix gave an algorithm using 3 bits, stabilizing in $O(n)$ and needing the execution of $n/2$ actions for the average advance of the token. We use a single bit, stabilizing in $O(n)$ and need the execution of $1,5$ actions for the average advance of the token.

Our second algorithm stabilizes in time proportional to the number of faults that hits the ring. In [?], J. Beauquier, C. Genolini and S. Kutten gave an algorithm using kn bits and k-stabilizing in $O(k^2)$. So, by introducing different token velocities, we improve the complexity bound of the previous approach by a factor n in space, and k in time.

References

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

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

ù To appear in *Theoretical Comp. Sci.*

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

[4] 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.

[5] 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*,

[6] 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.

[7] J. Beauquier, C. Genolini and S. Kutten Optimal Reactive k-Stabilization: the case of Mutual Exclusion. In *In PODC99 Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, May 1999.

[8] 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.

[9] 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.

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

[11] 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.

[12] M. Gouda and F. Furman Haddix. The Stabilizing Token Ring in Three Bits. In *Journal of Parallel and Distributed Computing 35*, pages 43–48, 1999.

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

[14] 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.

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

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