SELF-STABILIZATION IN DISTRIBUTED SYSTEMS – A SHORT SURVEY

Jerzy BRZEZIŃSKI[*] , Michał SZYCHOWIAK [**]

Abstract. Self-stabilization is a very interesting and promising research field in computing science. This is due to its guarantees of automatic recovery from any transient failure, without any additional effort. This paper presents an overview of self-stabilizing distributed algorithms. First, the outlook of the self-stabilization paradigm is shown, followed by a simple example and some formal definitions. Then, characteristics of stabilization types are described. Finally, the paper presents several self-stabilizing algorithms and further lines of investigation strive for distributed systems.

1. Introduction

One of the most wanted properties of distributed systems is fault tolerance. This can be achieved, in general, by two different approaches: pessimistic and optimistic. In the former, we deal with robust algorithms protected against any possible (i.e. the most pessimistic) or admissible set of failures. In the latter case, we use self-stabilizing algorithms, which after any failure guarantee to automatically reach a legal state in a finite time. Self-stabilizing algorithms are constructed in such a way that a given process will execute the same operations in both faulty and non-faulty states. Even a failure puts the system into a faulty state, the next steps of normally continued execution will put it back into a correct state. In general, self-stabilizing algorithms need not know anything about the failure, its type, duration, scope nor even whether it actually happened or not. The only requirement is that the failure is not permanent

Institute of Computing Science, Poznań University of Technology,
ul. Piotrowo 3A 60-965 Poznań Poland, fax: +48 61 8771525
[*] phone: + 48 61 6652370, e-mail: Jerzy.Brzezinski@cs.put.poznan.pl
[**] phone: + 48 61 6652378, e-mail: Michal.Szychowiak@cs.put.poznan.pl

and will eventually stop affecting the system, enabling finally to reach a correct state in a finite number of steps.

A self- stabilizing algorithm does not require correct initialization, can recover from any transient failure of arbitrary types occurring at any time and is insensitive to dynamic topology reconfiguration. This makes self-stabilization an interesting solution for a large number of applications including distributed reset problems (for the most interesting solutions see [3,5]), routing and communication protocols ([7,21, 29,32,44]), clock synchronization ([19,22,40,52]), graph theory ([12,40,41,43,50,57]), and others.

Since the very first presentation of a self-stabilizing algorithm, made by Dijkstra in 1974, progress made in this area has proved that it is one of most important and promising topics of fault tolerance. The goal of this work is to systematize a relatively large number of current research issues and results in self-stabilization and point out some new interesting research directions. This paper presents the current state of the art of self-stabilization with special attention to its use in distributed systems. It is organized as follows: Section 2 presents some fundamentals aimed of giving the intuitive notion of stabilization. Section 3 gives formal definitions and shows different types of stabilization and their significance in practical problems. Then Section 4 discusses some examples of known stabilizing distributed algorithms. Some concluding remarks are presented in the last Section.

2. Self-stabilization

Self-stabilization of a system guarantees that regardless of a current system's state, the system will converge to a legal state in a finite number of steps. In this section we present an outline of the theory of system stabilization and we deal with its important concepts: correctness and convergence.

2.1 Basic definitions

Definition 1. A **system** is the pair $S = (C,\rightarrow)$, where C is a set of states of the system S and $\rightarrow$ is a binary transition relation on C. A **computation** of S is a non-empty sequence $(c_1, c_2,...)$ such that for all $i\geq 0$: $c_i \in C$ and $c_i \rightarrow c_{i+1}$.

In distributed systems the above mentioned c_i expresses a global state, which is a concatenation of the local states of every process and the contents of every communication channel.

Definition 2. System S is **self-stabilizing** if there exists a subset $L \subset C$ of all legal states, such that:

- every computation starting in a state of L is correct (**correctness**)
  and
- every computation of S has a state from L (**convergence**) [58].

Subset L is problem specific and depends on the definition of a legal state related to the correctness of a given algorithm. So we do not attempt to give any formal definition of L here. Sometimes we explicitly add that subset L must be closed, i.e. every non-fault move from a legal state puts the system into a legal state. This will be discussed further in Section 3.1.

2.2 General properties of self-stabilizing algorithms

The convergence to a subset of legal states provides the possibility of automatic recovery from any system failure, after the fault has ceased, regardless of the state the system had been moved into and regardless of the quantity of the data that had been corrupted. It is then obviously required that the failure has to be transient (but of an unspecified duration).

The convergence implies three important properties of self-stabilization: embedded tolerance of arbitrary transient failures, unneeded proper initialization of the algorithm (as the initial state doesn’t have to be legal) and obvious adaptivity to dynamic changes of the system configuration, e.g. network topology changes (as any of them can be treated as a transient fault). Many algorithms would potentially benefit from these advantages.

But, on the other hand, there are some disadvantages of self-stabilizing algorithms. They are in particular: tolerance of only non-permanent faults, possible inconsistency of system states during convergence to legal states and no explicit detection of accomplishing convergence. Moreover, during failure time, a system can behave arbitrarily. Finally, a self-stabilizing algorithm can be generally more complex and hard to construct.

2.3 Classical example – Dijkstra's mutual exclusion

One of the clearest and most demonstrative examples of self-stabilizing algorithms is, already a classic, mutual exclusion in a ring of n processes, published by Dijkstra in 1974 [17]. It requires the number K of local states to be not less than the number of processes.

Dijkstra introduced the notion of a privilege. A *privilege* authorizes a given process to *make a move* (i.e. change its local state, enter the critical section). The legal (global) system state must satisfy the following properties:

1. There must be at least one privilege in the system.
2. During an infinite time every process should be able to receive a privilege an infinite number of times.

There is one exceptional process P_0 in the ring. It follows different rules than the other n-1 (P_1 – P_{n-1}) processes which are equal and behave uniformly. The current local state of a given process P_i is represented by a state variable l_i. The algorithm assumes any model of communication that lets every process in the ring know the value of the state variable of its left neighbor (l_{i-1} for $1 \le i \le n$-1, and l_{n-1} for P_0), e.g. a shared variables communication model*) (link-registers).

Here is Dijkstra's algorithm:

for P_0:	if $l_0 = l_{n-1}$	then $l_0 := (l_0+1)_{\text{mod } K}$ and P_0 has a privilege;
for P_i:	if $l_i \neq l_{i-1}$	then $l_i := l_{i-1}$ and P_i has a privilege; $1 \le i \le n$-1

Figure 1 shows an exemplary run of this algorithm in a ring of 3 processes, where K=3. Note that because of the assumed initial state, all three processes get a privilege (process P_0 satisfies the condition $l_0 = l_{n-1}$, process P_1: $l_1 \neq l_0$, and process P_2: $l_2 \neq l_1$) thus all three enter their critical sections and change their state in step 1. This global state is obviously not legal from the mutual exclusion point of view. Then, once again, all of them receive a privilege – leading to step 2 (another illegal global state). Now, condition $l_0 = l_{n-1}$ is violated – process P_0 cannot get into its critical section, but P_1 and P_2 still can (this is why only P_1 and P_2 change their state in step 3; the new state is still illegal). In the configuration after step 3 only P_2 has a privilege – the system has converged to a set of legal states. From now on, only one process will have a privilege. The privilege will circulate clockwise in the ring – e.g. in step 4: from P_1 to P_2, next from P_2 to P_0, etc.

It can easily be seen that this algorithm needs $O(n^2)$ system steps before reaching a legal global state (for further discussion of its time complexity please refer to [58],[26]). The required number K of possible local states depends of the ring size and must satisfy the relation: $K \ge n$.

---

* This stands in clear opposition to message passing, natural for distributed processing. However, we will not focus on a communication model now, presenting this very first example of self-stabilization, instead we delay the discussion of stabilizing distributed algorithms until Section 4.

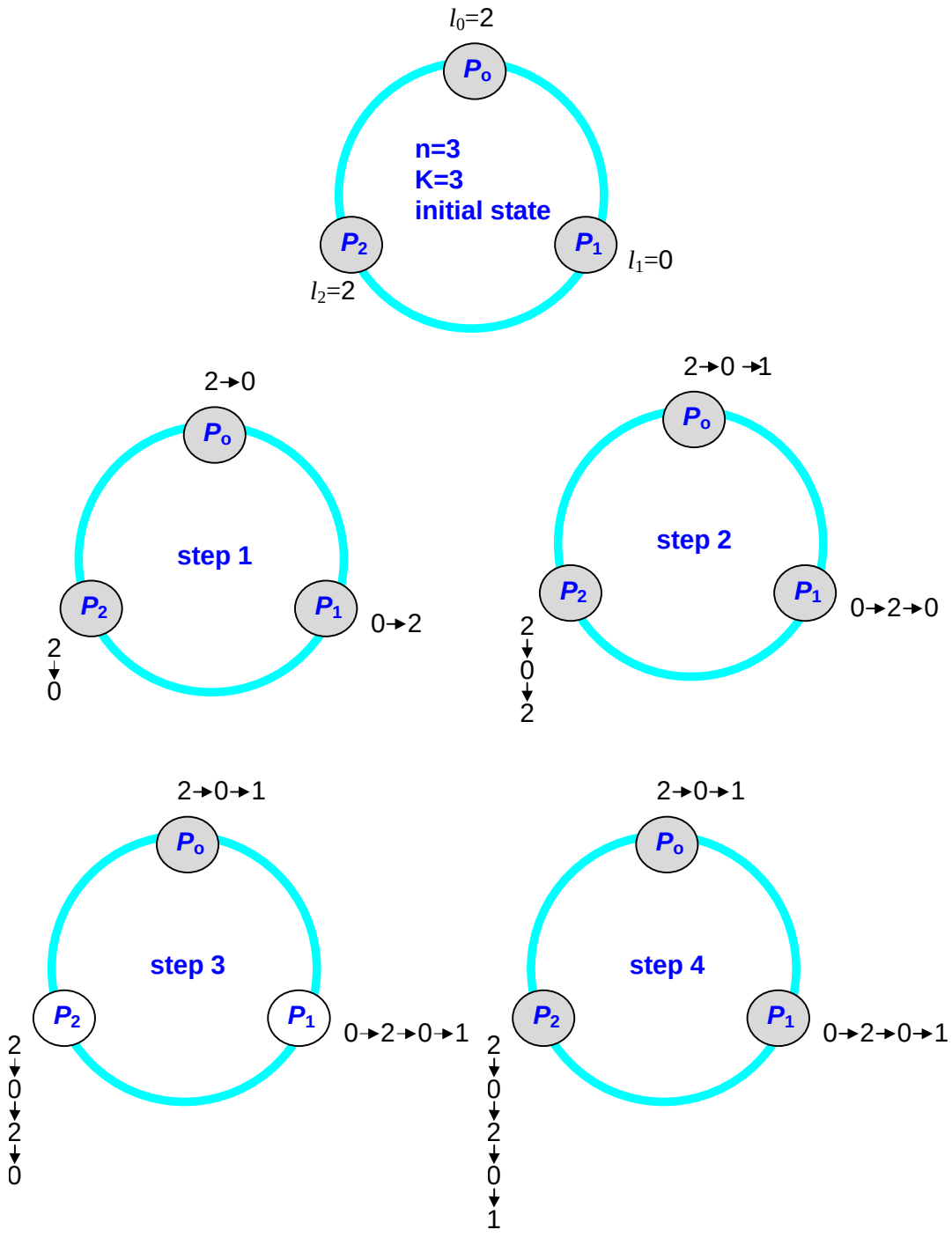

Figure 1. Exemplary run of Dijkstra's stabilizing mutual exclusion algorithm

2.4 System models and their implications

Network topology

Section 2.3 has shown a very simple algorithm. Unfortunately it requires a number of local states which depends on the ring size: $K \geq n$. In practice, we prefer algorithms that need a constant, and possibly low, number of states. For the ring topology there is a mutual exclusion algorithm which needs no more than 3 states, regardless of the size of the ring ([17]), but it requires bidirectional communication. It was also proven that 3 is the minimal number of states required by self-stabilizing solutions for mutual exclusion in the ring topology ([28]). Sometimes it is possible to reduce the number of states within a **special topology** of the network. Ghosh proposed in [28] a special topology and obtained a 2-state mutual exclusion algorithm. Although in a specific network we can find a less space-consuming solution, in general the need of a special topology is considered as a serious limitation.

Synchronous vs. asynchronous communication model

Many known distributed algorithms assume the asynchronous communication model. This is a very practical view because in most distributed systems we have finite but unbounded transmission delays. Unfortunately, many stabilizing algorithms expect the communication to be synchronous, like, for example, shared variables communication. This enables to concentrate mainly on processes state transitions, not on communication details and timing problems. Therefore, we assume often either the *state-reading model* or the *link-registers model*. In the first one, any given process P_i can read in one step the entire state of its neighbor P_j. In the second model, communication is performed by two registers shared between P_i and P_j, thus each process is able to transfer different data to its neighbor, depending on the neighbor.

As an illustration of the problems arising with asynchronous communication, let us analyze the behavior the Dijkstra's algorithm with asynchronous message passing. We introduce only one change into the original algorithm – instead of reading the state variable of the preceding neighbor, we force each process to send its local state variable to the succeeding neighbor every time that variable changes. Then, a simple local comparison of the received value with current value of the local state is performed, and its result determine the ownership of the privilege. Figure 2 shows two initial steps of a sample execution of the algorithm. Within the first step, every process sends a value of its local state variable. Receiving this value causes an appropriate update of state variables, if possible. In the situation in Figure 2 all three processes enter their critical sections upon reception of the first message, change their state variables and send them to their neighbors in step 1. Now, let's assume that some transient failure has moved the system into a state corresponding to step X in the Figure. From this moment, the system will possibly never stabilize because of several

messages being transmitted in the communication channels. So we see, that Dijkstra's mutual exclusion algorithm is not suited for asynchronous message passing communication.

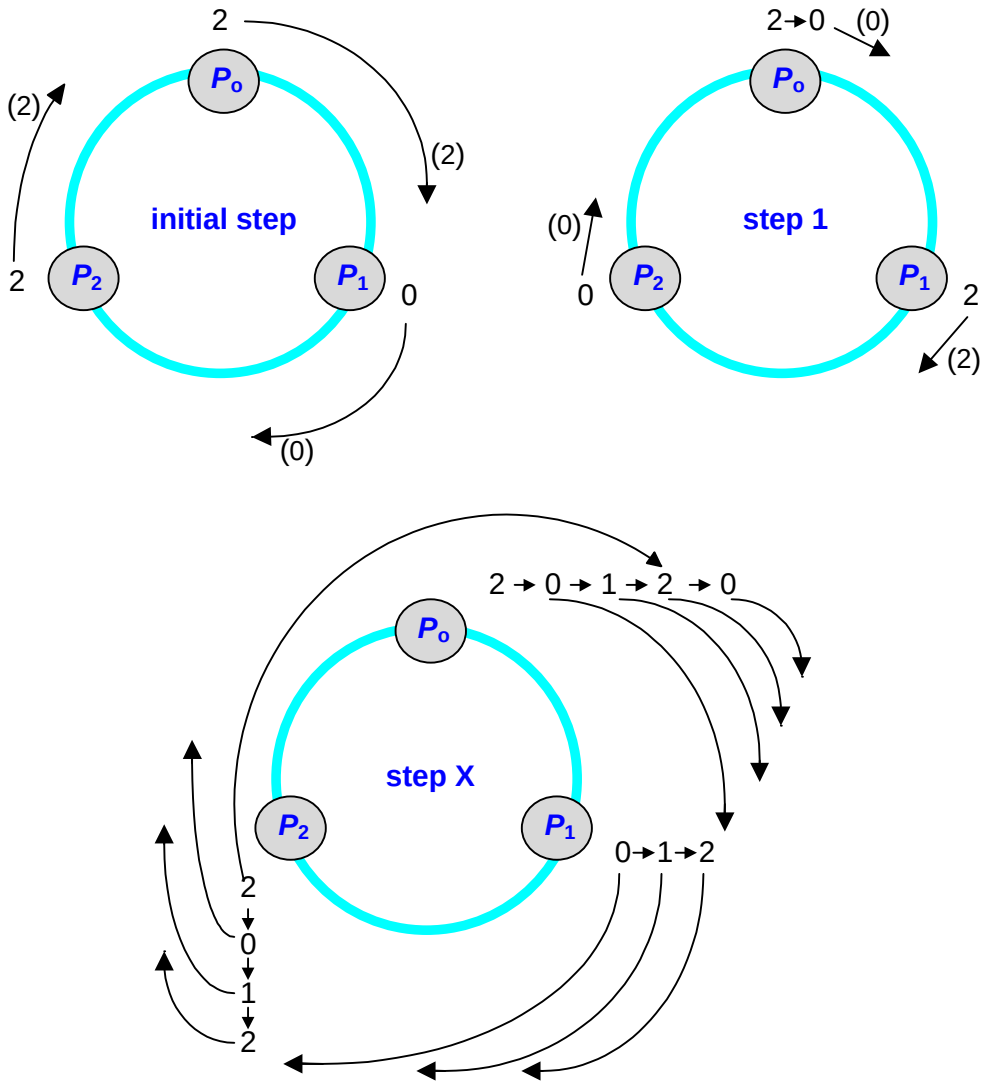

Figure 2. A run of modified Dijkstra's stabilizing mutual exclusion algorithm with message passing communication

In conclusion we can say that most stabilizing algorithms assume the synchronous communication model. Such algorithms are obviously less interesting for distributed computing, whose natural communication paradigm is asynchronous message passing. We will focus on distributed issues in self-stabilization in Section 4.

Reliability of communication channels

It should be easy to see that stabilization algorithms need not assume anything about the reliability of the communication. Because any possible channel (link-register) failure is supposed to be non-permanent, it shall fall into the category of transient failures.

3. Types and classes of stabilization

This section provides an analysis of known stabilization types, with formal specification of the concepts of closure and convergence.

3.1 Formal definitions

Let $S=(C,\rightarrow)$ be a system, according to Definition 1.

Definition 3. A subset P of C is **strongly closed** iff for every transition $p_1 \rightarrow p_2$: if $p_1 \in P$ then also $p_2 \in P$. A subset P of C is **weakly closed** iff for every computation $(p_1, p_2, ..., p_n, ...)$ beginning in P, there is non-empty suffix where each state is in P, i.e. if $p_1 \in P$ then there exists $n \geq 1$, such that for all $i \geq n$: $p_i \in P$. We can see that every strongly closed P is also weakly closed.

Now let P be a strongly closed subset of C and Q be a weakly (so perhaps also strongly) closed subset of C.

Definition 4. P **strongly converges** to Q iff every computation $(p_1, p_2, ...)$ beginning in P, has a state in Q, i.e., if $p_1 \in P$ then there exists $n \geq 1$, such that $p_n \in P$. P **weakly converges** to Q iff for every computation $(p_1, p_2, ...)$ beginning in P, for every state p_i in this computation exists a computation $(p_i, q_2, q_3, ...)$ beginning with this state p_i, which has a state in Q, i.e., if $p_1 \in P$ then there exists a computation $(q_1, q_2, q_3 ...)$ satisfying $q_1 = p_i$ for every $i \geq 1$, and there exists $n \geq 1$, such that $q_n \in Q$.

The weak convergence can be referred to as "probabilistic convergence" (see [34]), because it guarantees that from each state of a computation beginning in P there exists a way (another computation) leading to Q, but it does not guarantee that the system will choose this way. It is only a probable way. There is also another definition of probabilistic convergence, which explicitly introduces the probabilistic function f from the natural numbers set $\mathbb{N}$ into the interval $\langle 0;1 \rangle$, such that $\lim_{k \to \infty} f(k) = 0$, and the probability that P converges to Q within k transitions is at least $1\text{-}f(k)$.

Probabilistic convergence is deeply studied in [35].

From the above brief presentation of closure and convergence, we can derive the four following concepts of convergence:

a) strong convergence to strong closure;
b) strong convergence to weak closure;
c) weak convergence to weak closure;
d) weak convergence to strong closure.

These concepts imply four types of stabilization:

I. strong stabilization to a strongly closed set of legal states*);
II. strong stabilization to a weakly closed set of legal states;
III. weak stabilization to a weakly closed set of legal states;
IV. weak stabilization to a strongly closed set of legal states.

Types I and II constitute a **deterministic stabilization** class, types III and IV – **probabilistic stabilization** class. Types II and III belong to a **pseudo-stabilization** class. Please note that pseudo-stabilization is not mutually exclusive with deterministic nor probabilistic stabilization, therefore we have, in fact, a deterministic pseudo-stabilization (referring to type II above) as well as a probabilistic pseudo-stabilization (type III).

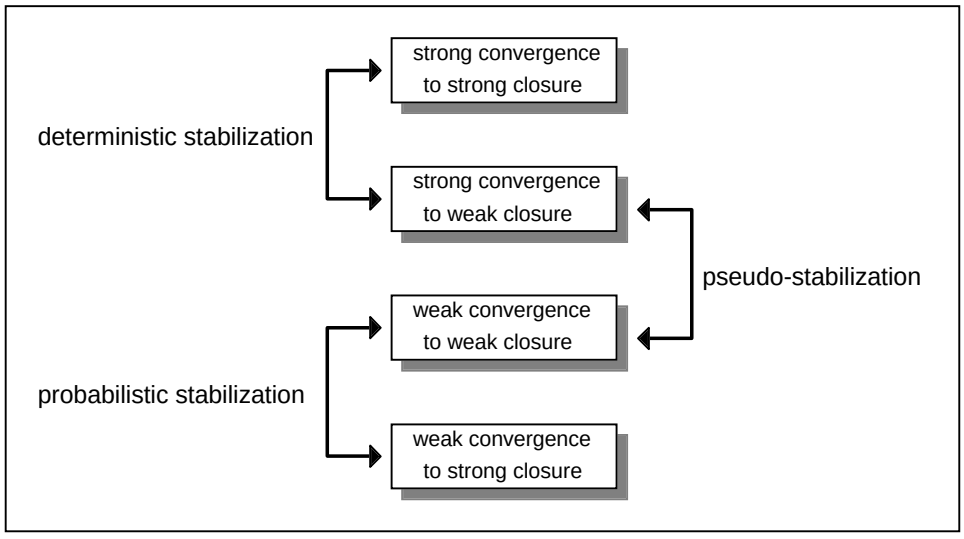

Figure 3. Four types of stabilization and three classes of stabilization

* meaning that system S strongly stabilizes to strongly closed set L

Only deterministic stabilization offers a bounded time of stabilizing from an illegal state, but probabilistic stabilization and pseudo-stabilization can often be easier to achieve or more efficient.

If the entire set C of all possible states converges (weakly or strongly) to the set of legal states L, we have a **self-stabilization**.

There is also another class of stabilization, called **superstabilization**, related to the reliability of distributed protocols. It concerns the amount of time of recovery or adaptation to a topology change. A stabilizing protocol is superstabilizing if it satisfies an additional so called *passage* predicate, which is a specific safety property that has to be guaranteed while the system recovers or undergoes topology changes starting from a legal state. For more information about superstabilization the reader is referred to [19] and [36].

3.2 Properties of convergence and closure

Convergence and closure follow some mathematical laws, like junctivity or transitivity, applicable to all states of a system. Here we present some of the laws for a given system $S=(C,\rightarrow)$.

Let P, R be subsets of C.

Junctivity of closure:
- if both P and R are strongly closed then strongly closed are also $P\cup R$ and $P\cap R$.
- if both P and R are weakly closed then $P\cup R$ is weakly closed and $P\cap R$ is weakly closed (but $P\cap R$ may be strongly closed).

Junctivity of strong convergence:
- if P strongly converges to Q_P and R strongly converges to Q_R then $P\cup R$ strongly converges to $Q_P\cup Q_R$ and $P\cap R$ strongly converges to $Q_P\cap Q_R$.

Junctivity of strong stabilization:
- if $(C,\rightarrow)$ strongly stabilizes to P and $(C,\rightarrow)$ strongly stabilizes to R then $(C,\rightarrow)$ strongly stabilizes to both $P\cup R$ and $P\cap R$.

Transitivity of strong convergence:
- if P strongly converges to Q and Q strongly converges to R then P strongly converges to R.

Transitivity is a very useful lemma, enabling stabilization prove of the whole system S, by showing the stabilization of every subsystem of S.

Several other laws as well as requirements for establishing stabilization properties can be found in [34]. Moreover, because in most cases the formal design of

stabilizing algorithms and their proofs are very complicated and difficult to follow, a significant attempt to make these tasks less tedious was made by Prasetya and Swierstra [52]. They have developed and applied a set of new operators for the UNITY programming logic [14], a tool for describing properties of distributed algorithms. Their work presents some formal structures and strategies for designing stabilizing distributed algorithms, as well as shows how various formal laws can be used to reason stabilization.

In some cases it is practical to concentrate not on system execution but only on system behavior perceptible to an external observation entity – i.e. *external behavior*. Analysis of stabilization based on external behavior can be found in [59]. The external behavior paradigm allows us to construct a system which stabilizes to another system even though they both have different sets of states. This is a very suitable technique for verifying stabilization of a given representation or model, when any other representations/models of the same concept are known to be stabilizing (e.g. verifying implementation stabilization by means of specification stabilization).

3.3 Properties of stabilizing algorithms

There are many classical problems, that have stabilizing solutions. We would like to discuss now a few properties of stabilizing algorithms belonging to distinct classes and having different characteristics.

Uniformity and impossibility of uniformity

Already presented in Section 2.3, algorithm for mutual exclusion [17] is an example of deterministic self-stabilization. That algorithm has one distinguished process, whose behavior is different than the behavior of other processes. In general, we will prefer *uniform* algorithms that have all processes identical and indistinguishable. Although uniformity is very desirable, in most cases it is impossible or hard to obtain. Dijkstra showed in [18] that no deterministic uniform algorithm exists for mutual exclusion if the size n of the ring is not prime.

Another well known problem of concurrent processing is the dining philosophers problem. Lehman and Rabin showed in [47] that there is no deterministic uniform solution to this problem. But an asymmetric solution proposed by Gouda [33] can be used.

Space complexity-reduction

Dijsktra's solutions works in a simple unidirectional ring. But we have mentioned that Ghosh reduced the minimum number of states to only 2 assuming a special network topology [28]. On the other hand, Herman presented a simple probabilistic

mutual exclusion protocol for an odd sized unidirectional ring of uniform processes in [35] and reduced the computation space to 3 with non-uniform processes, where the distinguished process owns a random number generator used to yield a new state number.

As a conclusion to this discussion, we can state that there are two possible ways of reducing the number of states (i.e. memory space requirements) – usage of a special topology and action randomization, each one leads to relaxing the strength of stabilization properties and the scope of possible applications.

Central deamon

Originally, Dijkstra's model of computation assumed the presence of a kind of central daemon. Its role was to assure that in a single move only one process would read the state of the neighbors and eventually change its state[*)]. The concept of a central daemon may be useful in proving stabilization, but in many systems it is not realistic because there often is no possibility to use such a central daemon. The notion of a randomized central daemon was used in [25] to get a 2-state self-stabilizing algorithm for a unidirectional ring. In any state when more than one process can make a move, the daemon chooses one of them in a random way.

Systems with continuous states

To end with, we mention an interesting attempt to find an appropriate computational model for self-stabilizing systems with continuous states, presented in [37]. That paper poses numerous questions provoking future work over some new fields such as: the construction of continuous computation models, the significance of different stabilization types and the influence of the continuous nature of processes upon behavior of known algorithms.

4. Self-stabilization in the field of distributed systems

We will mention some common applications of self-stabilization in the field of distributed systems here. They include particularly such fields as initialization and reset problems, termination detection, leader election, graph problems (e.g. spanning tree construction), clock synchronization or communication protocols.

Most of existing self-stabilizing solutions for distributed processing problems use the shared variables communication model instead of message passing. Only a few

[*] Fortunately, the Dijkstra's mutual exclusion algorithm also works without the central daemon [17]. In fact, the exemplary run of this algorithm we have presented in Section 2.3 did not use any central daemon.

algorithms are message-driven (essentially [21][32]). The explanation of such a state can be found in [21], where it is proven that any message-driven protocol has a possible global state in which all processes are waiting for messages but there are no messages on any channel (*communication deadlock*). A self-stabilizing system should stabilize when started from any possible initial configuration, including a configuration with communication deadlock. This implies that a non-trivial, completely asynchronous, self-stabilizing system cannot be message-driven. There are two ways of tackling this problem: the use of time-outs (as in [21]), compromising complete asynchrony, and development of protocols for which at any configuration there is at least one process whose next operation is to send a message [44]. Dolev, Israeli and Moran [21] propose a general method for simulating self-stabilizing shared variable protocols by self-stabilizing message-driven protocols using token passing.

Communication protocols

We start the presentation of stabilizing distributed algorithms with some communication protocols, a very important subject in distributed systems. The sliding window and alternating bit are fundamental asynchronous communication protocols. The *sliding-window* technique [56] consists of a sequence number generator on the side of the transmitter process which is responsible for enumerating every message sent through the channels, and an acknowledgment mechanism used by the receiver. In the case of suspected message lost, a retransmission is involved. Gouda and Multari [32] present a self-stabilizing sliding-window algorithm with three properties: (1) it is non-terminating (2) processes are infinite-state automata and (3) the network layer offers a timeout utility. Furthermore, the authors show that it is impossible to develop a self-stabilizing sliding-window protocol using bounded sequence numbers and deterministic finite state automata. Burns and Gouda [11] noticed that the sliding-window protocol, using bounded sequence numbers, does not work if the sequence numbers are periodic. Burns and Gouda propose a pseudo-stabilizing sliding-window protocol that uses a bounded aperiodic sequence number generator. Instead of aperiodic sequences, random sequences can be used. Random sequences are easier to generate. The *alternating bit* protocol is a special case of the sliding-window, when sequence numbers are taken from the alphabet {0,1}. Afek and Brown [1] describe an infinite-state deterministic version of the alternating bit protocol and also a finite-state probabilistic version.

Graph theory problems

Graph problems unceasingly receive wide attention in the domain of distributed processing. Many topology related issues are directly connected with well known problems of the graph theory. One of those issues is the orientation of an undirected ring [41][57]. Let us consider, as an example of a self-stabilizing ring orientation

algorithm, an Israeli and Jalfon [41] deterministic uniform solution for the link-register model. The aim of this algorithm is to consistently label links between neighboring processes, accordingly to the current sense of direction. Each process *P* has to label one of its links by *successor* (when a current link direction is from *P* to the neighbor) and the other by *predecessor* (direction to *P* from the other neighbor). A process immediately changes one of its labels if both labels are equal. All processes continuously circulate tokens, setting their labels accordingly to the token passing direction (from successor to predecessor). If two tokens meet, one is eliminated. Eventually all remaining tokens will keep circulating in the same direction. Figure 4 presents an algorithm for computing *P*'s label of the link *P*-*Q* (denoted $link_{pq}$). The algorithm is identical for both *P* links. We will briefly describe this algorithm now.

The solution for the orientation problem requires the system to reach the following postcondition: "for every edge *P*-*Q*, $label_{pq}$=*successor* iff $label_{qp}$=*predecessor*". The algorithm uses link-register $label_{pq}$ (one for each neighbor) and a state variable $state_p$ holding the values *sending*, *receiving*, *idle*. In one step, process *P* checks its state and register value, the state of the neighbor *Q* and the appropriate link-register value of this neighbor, and changes the state if one of the five guards is true. In the state *sending*, the process *P* waits to forward a token to its successor (the token will be transferred in action 2). In the state *receiving*, *P* waits to receive a token from its predecessor. The process *P* holds a token if either its state is *sending* or its state is *receiving* while its predecessor is not in state *sending*, in which case *P* tries to propagate the token to its successor (action 3). A token is destroyed when two tokens travelling in different directions meet (action 4). A new token is created when *P* and *Q*, both idle, are inconsistently oriented (action 5) – this token will overrule *Q*'s orientation. The algorithm requires at most n^2 steps to reach a legal state, when all *n* processes are oriented equally. Note that this algorithm requires a central deamon.

Very important graph problems useful for many distributed algorithms are also: leader election and spanning tree construction. In the *leader election* problem all processes agree on the identity of one process which will perform some special action becoming e.g. computation initializator, coordinator or root of a communication tree, etc. Most stabilizing election algorithms works in a ring topology. A very common election method consists in gathering adjacent nodes into segments and joining segments until there will be only one segment – the winner. Then, one node – namely the head of the winning segment – becomes the leader. Stabilization must ensure that any configuration change will result in composing a new winning segment (perhaps identical to the former). Therefore, a segment composition procedure must never end, enabling any node to automatically become the head of a new segment, if necessary. Obviously, this may require a continuous message exchange (e.g. token circulation), leading to high communication overhead.

(1) [$state_p$=*idle* $\wedge$ $state_q$=*sending* $\wedge$ $label_{qp}$=*successor*] $\Rightarrow$
$state_p$:= *receiving*; [$label_{pq}$=*successor*] $\Rightarrow$ flip()

(2) [$state_p$=*sending* $\wedge$ $label_{pq}$=*successor* $\wedge$ $state_q$=*receiving* $\wedge$ $label_{qp}$=*predecessor*] $\Rightarrow$
$state_p$:=*idle*

(3) [$state_p$=*receiving* $\wedge$ $label_{pq}$=*predecessor* $\wedge$ $\neg$($state_q$=*sending* $\wedge$ $label_{qp}$=*successor*)] $\Rightarrow$
$state_p$:=*sending*

(4) [$state_p$=$state_q$=*sending* $\wedge$ $label_{pq}$=$label_{qp}$=*successor*] $\Rightarrow$
$state_p$:=*receiving*; flip()

(5) [$state_p$=$state_q$=*idle* $\wedge$ $label_{pq}$=$label_{qp}$=*successor*] $\Rightarrow$
$state_p$:= *sending*

procedure flip(): reverse values of both *P* link labels

Figure 4. Israeli and Jalfon ring orientation algorithm for one link

Because of the impossibility result shown by Dijkstra [17] there can be no deterministic uniform stabilizing leader election in a ring of unknown (composite) size. The intuition of the proof is that there is no way to break the symmetry in a ring of composite size. Mayer, Ofek, Ostrovsky and Yung [50] presented a randomized constant space leader election on asynchronous bidirectional rings which works under the assumption of no deadlock. For odd size unidirectional rings a probabilistic synchronous protocol of Herman [35] can be used. Averbuch, Itkis and Ostrovsky [43] propose a sublogarithmic space randomized self-stabilizing protocol for leader election for arbitrary topologies. And finally Itkis, Lin and Simon obtained a deterministic constant space self-stabilizing leader election on uniform bidirectional rings in [41]. Their protocol requires a central deamon.

The problem of constructing a spanning tree consists of reducing a given graph to a tree containing every node of the former graph. Solutions to this problem are used by several network protocols and in diffusing computation. A very common approach is to start with electing a leader which will become the root of the tree being constructed. Then the root performs a labeling protocol that assigns a unique identifier to any node of the graph. The identifier can be constructed in a way that unambiguously indicates a parent node in the new tree. Here, the stabilization role is to ensure that any possible configuration change will result in new identifier assignment. Thus the labeling protocol never ends, enabling any node to successfully change its parent. Due to the same reason as for leader election, there exists no deterministic uniform stabilizing algorithm for spanning tree construction in an anonymous graph. A uniform randomized self-stabilizing spanning tree construction with message passing can be found in [12].

Termination detection

One of the most widely known problems of the distributed processing is termination detection. In distributed systems it is impossible for a given process to decide about a global termination basing on a local state. To make the computed results definitive, a global termination detection is necessary. Several self-stabilizing algorithms solving this problem have been developed by Gouda and Evangelist [23]. Each of these algorithms has a different convergence time and response time, but the authors show that if the convergence speed is improved or decreased by some factor, the response is always delayed, or respectively increased, by the same factor.

Clock synchronization

An interesting topic is the problem of clock synchronization required for many distributed protocols. Solutions to this problem must ensure that eventually the values of local clocks of all nodes of the system are kept within the maximum allowable drift rate, regardless of their initial values. Several stabilizing solutions have been proposed by many authors (see the most recent: [19] [22] [40] [52]).

Fault containment

A very desirable feature of a self-stabilizing system is that the amount of disruption caused by a fault would be proportional to the severity of the fault, represented e.g. as a number of processes that have been corrupted. However, this is not true for many self-stabilizing systems. Ghosh, Gupta and Pemmaraju [29] address this issue by introducing the notion of fault containment. According to this paper, a *fault-containing* self-stabilizing algorithm is one that contains the effects of limited transient faults while retaining the property of self-stabilization. Fault containment can be expressed e.g. as a recovery time (a bound on the worst case recovery time) proportional to the number of failed processes or as a number of processes involved in the recovery (a bound on the worst case recovery space overhead) proportional to the number of faulty states. This paper proposes a formal framework for specifying and evaluating fault-containing self-stabilizing protocols. The authors show that self-stabilization and fault containment are contradictory goals in general. The paper also presents a transformer $\mathcal{T}$ that maps any self-stabilizing algorithm into an equivalent fault-containing self-stabilizing algorithm that can repair any 1-faulty state in O(1) time and O(1) space overhead. This transformation is based on a special stabilizing timer paradigm which significantly simplifies the task of achieving fault containment. The paper concludes by generalizing the transformer $\mathcal{T}$ into a parameterized transformer $\mathcal{T}(k)$ that can obtain varying performance measures for varying k. This subject is further studied in [30].

General techniques to achieve stabilization

A great amount of work has been done to find a universal technique for converting an arbitrary asynchronous system into its stabilizing equivalent. Katz and Perry showed a method called *global checking and correction* [44]. Its idea is to periodically do a snapshot of the system and reset the computation if a global inconsistency is detected. Stabilizing algorithms for distributed reset can be found in [3] and [5]. This transformation applies to several asynchronous protocols, but is rather expensive. In [6] and [59] we can find a more efficient method of global inconsistency detection – a *local checking*. Those papers describe further a *local correction* mechanism, which enables a process to move into a global legal state by local actions (by its definition a local action affects only the state of a given process and its neighbor). Unfortunately, not every protocol is locally correctable. Varghese [59] proposes a *local checking and global correction* method that can be used for every locally checkable protocol. This method uses a stabilizing distributed reset protocol (e.g. one proposed in [3] or other from [5]). In the distributed reset problem, reset requests may be issued at any node and processed for example in a diffusing way (wave propagation). Varghese shows that every locally checkable protocol on a tree can be space-efficiently stabilized.

When the protocol is not locally checkable another general technique can be used – *counter flushing*, presented in [61]. It attaches a counter that is checkable by a leader process to every message and state of every node. The leader uses broadcasts to synchronize other processes and maintain legal global states. A randomized version of this method uses random counter values. It is worth mentioning that this version has a very good expected stabilization time proportional to the diameter of the network. Varghese has applied this simple technique to, among others, the Misra's deadlock detection algorithm ([51]), Chandy-Lamport snapshot algorithm ([13]) and Dijkstra's mutual exclusion ([17]) in [62]. Bourgon and Datta uses counter flushing in their protocol for heap maintenance [9].

Costello and Varghese also introduced the next technique for constructing self-stabilizing systems called *window washing* ([16]) – a generalization of the sliding-window protocol. Window washing is a modification of counter flushing and does not require atomic broadcasts and has weaker space consumption.

Kutten and Peleg propose in [45] a technique called *mending* and Kutten and Patt-Shamir extended it in [46], to a *time adaptivity* property, both intended to repair locally the global system state for any distributed algorithm with O(*f*) time depending only on the number *f* of faults. These ideas use replication and voting strategy to repair corrupted states. However, this approach has some limitations, e.g. there must

be $f < \frac{n}{2}$, where n is the total number of processors, and the system may not recover if a fault occurs during reparation.

Herman [37] overcomes these limitations by proposing a *composite* protocol combining the time-adaptive algorithm of [46] with a non-time-adaptive self-stabilizing "backup" algorithm for situations in which the first one may not converge. His solution deals with any number of faults. Also the *local stabilizer* of Afek and Dolev [2] recovers from any number of faults in O(f) expected time. Unfortunately both time-adaptivity and local stabilizer mechanisms assume a synchronous network with unit message delivery times, which makes these results less interesting in the context of general distributed systems.

5. New investigations and conclusions

Finally we would like to specify some problems in the theory of system stabilization. First of all, a very important but difficult problem is verifying stabilization. Several techniques have been developed to help algorithm designers to accomplish this task; we only mention system decomposition, the convergence stair method (step-by-step verification, [32]) and the assumption of the central deamon utility. Another problem is due to the vulnerability of some stabilization properties, which can disappear when transforming from the specification of an algorithm to its implementation. Some help can be given with the concept of stabilization based on external behavior, but there is still much work to be done before finding useful behaviors of a given system. Also, very promising is the construction of provably correct self-stabilizing algorithms using UNITY logic [14].

Some well-known models of concurrent and distributed computation cannot be used to construct stabilizing algorithms. Ghosh, Gupta and Pemmaraju [29] show examples of systems whose properties disable stabilization, such as Petri nets or finite state automata communicating by message passing over unbounded channels. Some problems cannot be solved in a self-stabilizing manner in specific situations (as the unlimited number of faults, presence of Byzantine faults etc.). The discovery of impossibility results will be an important topic of further research (some can be found in [9][19] or [48]).

Although there are still many problems that have no stabilizing solution, a good task for future research would be to improve existing algorithms or models of stabilization. Improvement can concern the reduction of the required number of states, reduction of convergence time, enforcement of stabilization type, relaxation of used assumptions, uniformity or better automatic correction verification methods, and more. An interesting research direction could be the investigation of a problem of

specific randomization to achieve self-stabilizing solutions and the design of generic methodologies for systematic randomization. Recently, self-stabilization has become an interesting issue also in real-time systems and this appears to be another important research direction (see [15][19][55]).

Still an unexhausted topic is the problem of combining self-stabilization with other kinds of fault-tolerance (although a very interesting attempt to combine self-stabilization with failure detectors is presented in [9]). So we hope that the field of stabilization will stay one of the most interesting research areas for at least the next few years.

References

1. Y. Afek, G. M. Brown, "Self-Stabilization Over Unreliable Communication Media", *Distributed Computing*, vol.7, 1993, pp.27-34.
2. Y. Afek, S. Dolev, "Local Stabilizer", In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing PODC97*, 1997, pp. 287-297.
3. A. Arora, M. G. Gouda, "Distributed Reset", *IEEE Transaction on Computers, vol. 43, No. 9*, 1994, pp.1026-1038.
4. B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese, "Time Optimal Self-Stabilizing Synchronization", In *Proceedings of 25th ACM Symposium on Theory of Computing*, 1993.
5. B. Awerbuch, R. Ostrovsky, "Memory-efficient and Self-Stabilizing Network Reset", In *Proceedings of 13th Annual ACM Symposium of Distributed Computing*, 1994.
6. B. Awerbuch, B. Patt-Shamir, G. Varghese, "Self Stabilization by Local Checking and Correction", In *Proceedings of 32nd IEEE Symposium on Foundation of Computer Science*, 1991.
7. B. Awerbuch, B. Patt-Shamir, G. Varghese, "Self Stabilizing End-to-End Communication", *Journal of High Speed Networks*, 1995.
8. B. Awerbuch, B. Patt-Shamir, G. Varghese, S Dolev, "Self Stabilization by Local Checking and Global Reset", In *Proceedings of 8th International Workshop WDAG'94*, 1994.
9. J. Beauquier, S. Kekkonen-Moneta, "Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors", *International Journal of Systems Science*, 28(11):1177-1187, 1997.
10. B. Bourgon, A. K. Datta, "A Self-Stabilizing Distributed Heap Maintenance Protocol", In *Proceedings of 2nd Workshop on Self-Stabilizing Systems*, 1995.
11. J. Burns, M. G. Gouda, "Stabilization and Pseudostabilization", Tech. Report TR-90-13, DCS University of Texas, Austin, 1990.
12. F. Butelle, C. Lavault, M. Bui, "A Uniform Self-Stabilizing Minimum Diameter Spanning Tree Algorithm", In *Proceedings of 9th International Workshop WDAG'95* in *Lecture Notes in Computer Science, vol. 972*, Springer, 1995, pp.257-272.
13. K. M. Chandy, L. Lamport, "Distributed snapshots: determining global states of distributed systems", *ACM Trans. on Computer Systems, 3(1)*, vol. 3, 1985, pp.63-75.
14. K. M. Chandy, J. Misra, "Parallel Programs Design – A Foundation", Addison-Wesley Publishing Company Inc., 1988.

15. A. M. K. Cheng, "Self-stabilizing real-time rule-based systems", In *SRDS92 Proceedings of the 11th Symposium on Reliable Distributed Systems*, 1992, pp.172-179.
16. A. M. Costello, G. Varghese, "Self Stabilization by Window Washing", Washington University, St. Louis, 1996.
17. E. W. Dijkstra, "Self-Stabilization in Spite of Distributed Control", *Commun. ACM vol. 17*, 1974, pp.643-644.
18. E. W. Dijkstra, "Self-Stabilization in Spite of Distributed Control", In *Selected Writing on Computing. A Personal Perspective*, Springer-Verlag, 1982, pp.41-46.
19. S. Dolev, "Possible and impossible self-stabilizing digital clock synchronization in general graphs", *Journal of Real-Time Systems*, no. 12(1) 1997, pp.95-107.
20. S. Dolev, T. Herman, "Super-Stabilizing Protocols for Dynamic Distributed Systems", *Proceedings of 2ng Workshop on Self-Stabilizing Systems*, 1995.
21. S. Dolev, A. Israeli, S. Moran, "Resource bounds for self-stabilizing message-driven protocols", *SIAM Journal on Computing*, no. 26, 1997, pp.273-290.
22. S. Dolev, J. L. Welch, "Wait-free clock synchronization", *Algorithmica*, 1997, pp.486-511.
23. M. Evangelist, M. G. Gouda, "Convergence/Response Tradeoffs", Technical Report STP-124-89, MCC Software Technology Program, 1989.
24. M. Fischer, N. Lynch, M. Merritt, "Easy impossibility proofs for distributed consensus problem", *Information Processing Letters*, vol.14, 1982, pp.183-186.
25. M. Flatebo, A. K. Datta, "Two-State Self Stabilizing Algorithms", DCS University of Nevada, Las Vegas, 1991.
26. M. Flatebo, A. K. Datta, S. Ghosh, "Self Stabilization in Distributed Systems". In *Readings in Distributed Computing Systems*, IEEE Computer Society Press, 1994, pp.100-114.
27. R. Gallager, P. Humblet, P. Spira, "A distributed algorithm for minimum-weight spanning trees", ACM TOPLAS, vol. 5, 1983, pp.66-77.
28. S. Ghosh, "Understanding Self-Stabilization in Distributed Systems", *Technical Report TR-90-02, Dept. of CS, University of Iowa*, Iowa City, 1990.
29. S. Ghosh, A. Gupta, S. V. Pemmaraju, "Fault-containing network protocols", *Proceedings of 12th ACM Symposium on Applied Computing*, 1997.
30. S. Ghosh, S. V. Pemmaraju, "Trade-offs in fault-containing self-stabilization", Technical Report, University of Iowa, 1997
31. M. G. Gouda, R. R. Howell, L. E. Rosier, "The Instability of Self-Stabilization", *IEEE Transactions on Software Engineering*, vol.17, 1990. pp. 697-724.
32. M. G. Gouda, N. Multari, "Self-Stabilizing Communication Protocols", *IEEE Transactions on Computers*, vol.40, no.4, 1991. pp. 448-458.
33. M. G. Gouda, "The Stabilizing Philosophers: Asymmetry by Memory and by Action", Technical Report TR-87-12, Dept. of Computer Science, University of Iowa, 1990.
34. M. G. Gouda, "The Triumph and Tribulation of System Stabilization", *Distributed Algorithms. Proceedings of 9th International Workshop WDAG'95* in *Lecture Notes in Computer Science*, vol. 972, Springer, 1995, pp.1-18.
35. T. Herman, "Probabilistic Self-Stabilization", *Information Processing Letters*, vol. 35, 1990, pp.63-67.
36. T. Herman, "Super-Stabilizing Mutual Exclusion", Technical Report TR 97-04, University of Iowa, 1997.

37. T. Herman, "Observations on time adaptive self-stabilization", Technical Report TR 97-07, University of Iowa, 1997.
38. T. Herman, "Distributed Repair Timers", Technical Report TR 98-05, University of Iowa, 1998.
39. H. J. Hoover, "On the Self-Stabilization of Processes with Continuous States", *Proceedings of 2nd Workshop on Self-Stabilizing Systems*, 1995.
40. S. T. Huang, T. J. Liu, "Four-state stabilizing phase clock for unidirectional rings of odd size", *Information Processing Letters*, 65(6), 1998, pp.325-329.
41. A. Israeli, M. Jalfon, "Self-Stabilizing Ring Orientation", *Proceedings of 4th Workshop on Distributed Algorithms*, vol. 486 of Lecture Notes in Computer Science, Springer-Verlag, pp.1-14, 1990.
42. G. Itkis, C. Lin, J. Simon, "Deterministic, Constant Space, Self-Stabilizing Leader Election on Uniform Rings", In *Distributed Algorithms. Proceedings of 9th International Workshop WDAG'95* in *Lecture Notes in Computer Science*, vol. 972, Springer, 1995, pp.288-302.
43. G. Itkis, "Self-stabilizing distributed computation with constant space per edge", *Presentation at MIT*, 1992.
44. S. Katz, K. Perry, "Self-stabilizing extensions for message-passing systems", *Distributed Computing*, vol.7 no.1, 1993, pp.17-26.
45. S. Kutten, D. Peleg, “Fault-local distributed mending”, *PODC95 Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp.20-27.
46. S. Kutten, B. Patt-Shamir, "Time-adaptive self stabilization", *PODC97 Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, 1997, pp.149-158.
47. D. Lehman, M. Rabin, "On the advantages of free choice: A symmetric and fully distributed solution of the dining philosophers problem", *Proceedings of 8th Annual ACM Symposium on Principles of Programming Languages*, 1981, pp.133-138.
48. C. Lin, J. Simon, "Possibility and impossibility results for self-stabilizing phase clocks on synchronous rings”, *Proceedings of 2nd Workshop on Self-Stabilizing Systems*, 1995, pp.10.1-10.15.
49. F. Mattern, "Algorithms for distributed termination detection", *Distributed Computing*, vol. 2, 1987, pp.161-175.
50. A. Mayer, Y. Ofek, R. Ostrovsky, M. Yung, "Self-Stabilizing Symmetry Breaking in Constatnt-Space", *STOC’92*, 1992, pp.667-678.
51. J. Misra, "Detecting termination of distributed computation using markers", In *Principles of Distributed Computing*, Montreal, Ontario, 1983
52. M. Papatriantafilou, P. Tsigas, "On self-stabilizing wait-free clock synchronization", *Parallel Processing Letters*, 7(3), 1997, pp.321-328.
53. I. S. W. B. Prasetya, S. D. Swierstra, "Formal Design of Self-Stabilising Programs", Technical Report UU-CS-1995-07, Rijksuniversiteit Utrecht, 1995
54. M. Schneider, "Self stabilization", ACM Computing Surveys, 1992.
55. M. Schneider, "Self-stabilizing real-time decision systems", In *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, ed. D.S. Fussell and M. Malek, 1995, pp.19-44.
56. A. S. Tannenbaum, "Computer Networks", Prentice-Hall, Englewood Clifs, 1991.

57. N. Umemoto, H. Kakugawa, M. Yamashita, "A Self-Stabilizing Ring Orientation with a Smaller Number of Processor States", *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 6, 1998.
58. G. Tel, "Introduction to Distributed Algorithms", Cambridge University Press, 1994, pp.459-487.
59. N. Umemoto, H. Kakugawa, M. Yamashita, "A Self-Stabilizing Ring Orientation with a Smaller Number of Processor States", IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 6, 1998.
60. G. Varghese, "Self-Stabilization by local checking and correction", *Ph.D. thesis*, 1992.
61. G. Varghese, "Self Stabilization by Counter Flushing", *Proceedings of 13th ACM Symposium on Principles of Distributed Computing*, Los Angeles, 1994.
62. G. Varghese, "Dijkstra's Protocols as Examples of Local Checking and Counter Flushing", *based on Distributed Algorithms Course at MIT, 1992*, St. Louis, 1994.

nearly complete bibliography list is maintained by Ted Herman at http://www.cs.uiowa.edu/ftp/selfstab/bibliography/