

**Safe and Eventually Safe:  
Comparing Stabilizing and non-Stabilizing Algorithms  
on a Common Ground**

by

Shlomi Dolev, Sylvie Delaët, and Olivier Peres

Technical Report #2009-05

August 2009

# Safe **and** Eventually Safe:

## Comparing Stabilizing and non-Stabilizing Algorithms on a Common Ground

Sylvie Delaët<sup>1</sup>      Shlomi Dolev<sup>2\*</sup>      Olivier Peres<sup>2†</sup>

### Abstract

Self-stabilizing systems can be started in any arbitrary state and converge to exhibit the desired behavior. However, self-stabilizing systems can be started in predefined initial states, in the same way as non-stabilizing systems. In this case, a self-stabilizing system can mask faults just like any other distributed system. Moreover, whenever faults overwhelm the systems beyond their capabilities to mask faults, the stabilizing system recovers to exhibit eventual safety and liveness, while the behavior of non-stabilizing systems is undefined and may well remain totally and permanently undesired. We demonstrate the importance of defining the initial state of a self-stabilizing system in a specific case of distributed reset over a system composed of several layers of self-stabilizing algorithms. A self-stabilizing stabilization detector ensures that, at first, only the very first layer(s) takes action, and that then higher levels are activated, ensuring smooth restarts, while preserving the stabilization property. The safety of initialized self-stabilizing systems, combined with their better ability to regain safety and liveness following severe conditions, is then demonstrated over the classical fault masking modular redundancy architecture.

**Keywords:** self-stabilization, safety.

## 1 Introduction

A distributed algorithm operates in a system consisting of several processes in order to perform a given task. For example, the mutual exclusion task is defined by a set of infinite executions in which at most one process executes the critical section in any configuration, and any process executes the critical section infinitely often.

A *self-stabilizing* algorithm [10, 11] has an additional property: it guarantees to eventually execute its task, by reaching a *legitimate configuration*,

---

<sup>1</sup>Univ Paris Sud; LRI; CNRS; Orsay F-91405.

<sup>2</sup>Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel 84105.

\*Partially supported by the ICT Programme of the European Union under contract number ICT-2008-215270, US Air Force, and Rita Altura chair in computer science.

†Partially supported by the ICT Programme of the European Union under contract number ICT-2008-215270.

regardless of the state in which the processes and communication links are started. Self-stabilization can thus be used to overcome any set of transient failures, e.g. arbitrary memory corruptions due to single upset events [21] caused by cosmic rays, or hardware malfunction in abnormal conditions, e.g. excessive heat.

Some algorithms are supposed to remain *safe* at all times while they carry out their task. For example, the part of the mutual exclusion task definition stating that no two processes may access the critical section simultaneously is a safety property. Safety, however, is impossible when very high levels of failures essentially make the system disappear.

The fact that self-stabilizing systems have the added ability to regain consistency when started in an arbitrary configuration indicates that the designers of self-stabilizing algorithms consider all configurations, including those where safety does not hold. Clearly, all configurations are not safe. However, a self-stabilizing algorithm can be initialized, just like any distributed algorithm, in a *initial configuration*. In this case, it can provide safety during its execution. If the system is initialized in any other configuration, it converges, without safety, to a legitimate configuration, which is already an improvement over non-stabilizing systems.

It is clear that in a self-stabilizing system, all starting configurations are not equally desirable. Likewise, not all illegitimate configurations are equally bad: some of them are close to a legitimate configuration, i.e. they only require a few steps of the algorithm to reach legitimacy, while others are much farther away.

Therefore, each self-stabilizing algorithm should have a set of *initial configurations*. These configurations must be safe, which adds the requirement of proving *initial safety*. This ensures that when the system is started in an initial configuration, there is no need to overcome any problem stemming from the starting configuration. Thus, safety is guaranteed throughout the execution, and eventually a legitimate configuration is attained.

However, initial configurations are not intended to provide a direct solution to the task. Rather than encoding a complete solution like a legitimate configuration, an initial configuration should not contain any wrong information. For example, an initial configuration for an algorithm that builds a breadth-first-search (BFS) tree should not itself describe a BFS tree. A good initial configuration would be one where every process has no parent and no child.

It is typically possible to reset the system into an initial configuration by having all the processes write predefined default values into their local variables, without requiring remote information. In conjunction with a self-stabilizing reset algorithm that resets layered algorithms, e.g. mutual exclusion algorithms using the output of a spanning tree algorithm [13], the use of initialized stabilization detectors allows fast and safe convergence.

## Related Works.

**Reset.** The goal of a *distributed reset algorithm* [2] is to place the whole system in a predefined configuration. Combined with the ability to detect that some unwanted property holds [1, 5, 17, 18], this enables resetting the system to a configuration that begins a recovery operation. Some self-stabilizing reset algorithms have been proposed [2, 3, 8]. They optimize time complexity [8] and/or space complexity [3].

**Safety and liveness.** The classical tool for reasoning about a distributed system is temporal logic, which considers separately two types of properties: safety and liveness. A safety property states that some bad event never happens. For example, in the *mutual exclusion* problem specification, it is guaranteed that no two processes access a critical section at the same time. A liveness property states that some good event will happen, no matter how the execution proceeds, e.g. all the processes access the critical section infinitely often.

Self-stabilizing algorithms are designed to be able to reach a legitimate configuration when started in any configuration. Thus, liveness properties, being *eventual* properties, hold. Safety properties, on the other hand, need a closer examination.

**Safety of self-stabilizing systems.** The design of safe self-stabilizing systems was studied by Ghosh and Bejan [15]. They presented a framework that not only captures the fact that a self-stabilizing system has safety properties, but also expresses how safe it is, in terms of a *safety margin* and a discrete metric, later extended, with Rao, to more realistic continuous metrics [6]. This framework is related to the safety of legitimate configurations, rather than on the definition of initial configurations and the initial safety of self-stabilizing algorithms.

Gouda, Cobb and Huang introduced *tri-redundant* systems [16], in which variables are replicated and whenever a process requests to read the value of a variable, the majority value among the replicas is returned. This helps building safe systems assuming faults occur during the execution, rather than before its beginning, as we assume.

Cournier, Datta, Petit and Villain introduced, in the context of snap-stabilization [7], a notion similar to that of initial configurations. Even then, initial configurations can improve the system, especially if another algorithm uses its output. Also, not all algorithms can be made snap-stabilizing, and we show in this paper that some existing self-stabilizing algorithms can exhibit safety properties without any modification; e.g. we show in this paper a routing table algorithm that, started in a configuration where all the tables are empty, is safe.

## System Settings.

An *algorithm* is a set of variables and a set of actions. A *process* is a state machine resulting from a given algorithm. A *system* is a set of processes linked by a communication medium. A *configuration* is the collection of the states of all the processes of a system and the state of the communication medium. A *step* is a state transition of one of the processes, including communication actions through the communication medium attached to the process. It takes a process and leaves it in a new state according to the state transition function of the process, executing the relevant communication operations with neighboring processes.

An *execution*  $(c_0, a_0, c_1, \dots, c_i, a_i, c_{i+1}, \dots)$  is an infinite alternate sequence of configurations and actions in a system such that for all  $i \in \mathbb{N}$ ,  $c_i$  is a configuration and  $a_i$  is a step that takes the system in configuration  $c_i$  and leaves it in  $c_{i+1}$ . We only consider *fair* executions, in which any step which can be taken an infinite number of times is taken at some point.

A *k-prefix* of an execution  $(c_0, a_0, c_1, \dots, c_{k-1}, a_{k-1})$  for some  $k \in \mathbb{N}$  is the first  $k + 1$  configurations and atomic steps of the execution. A *k-suffix* of an execution for some  $k \in \mathbb{N}$  is the infinite alternate sequence of configurations and atomic steps  $(c_k, a_k, c_{k+1}, a_{k+1}, \dots)$  that follows a *k-prefix* of the execution. A suffix of an execution is thus itself an execution.

A predicate  $P$  on the executions of a system  $S$  is *eventually* true for  $S$  if and only if any execution  $\mathcal{E}$  of  $S$  has a suffix in which  $P$  holds.

For a given system, a *safety* property is a predicate on executions that holds on all the executions of that system. A *liveness* property is a predicate on executions verified over any suffix of an execution of that system. An *eventual safety* property is a safety property for some suffix of any execution of that system. An *initial safety* property is a safety property for all the executions starting from an initial configuration.

**Safe self-stabilization.** Given a set  $\mathcal{I}$  of initial configurations and an initial safety property  $P$ , an algorithm is safely self-stabilizing to a set  $\mathcal{L}$  of *legitimate configurations* if and only if it satisfies the following requirements: *initial safety*: for any execution  $\mathcal{E}$  that starts in a configuration in  $\mathcal{I}$ ,  $P(\mathcal{E})$  is true ; *convergence*: the execution of the algorithm, started in any configuration, eventually leads to a configuration in  $\mathcal{L}$ ; *closure*: the execution of any step of the algorithm from any  $c \in \mathcal{L}$  yields a configuration  $c' \in \mathcal{L}$ .

**Failures.** A *stopping failure*, or a *crash*, happens when a process stops taking steps. Depending on the context, the process can be allowed to *restart*, i.e. resume taking steps, or not. A *transient failure* is an arbitrary modification of a configuration. It can affect either one or several processes, or the communication medium, causing them to be set to an arbitrary state from their state space. As a convention, we do not allow an execution to contain

transient failures, but consider that the starting configuration of any execution occurs after the last transient failure. Self-stabilizing algorithms are expected to converge to a legitimate configuration regardless of their starting configuration, which effectively captures all possible transient failures in the past and their consequences.

The rest of the paper is organized as follows. In Section 2, we give a formal framework for analyzing self-stabilizing algorithms with reference to their starting configurations, using the Update protocol as an example. This framework helps understanding the self-stabilizing stabilization detectors which we introduce in Section 3. Composed with self-stabilizing algorithms started in initial configurations, they assist in obtaining smooth restarts. Section 4 is devoted to the modular redundancy case study: how a self-stabilizing algorithm designed with initial safety in mind provides more safety than an ordinary redundant system, especially in the presence of unexpectedly severe failures. We conclude our paper in Section 5.

## 2 Initial, Reacheable, Legitimate, Corrupted Configurations

We introduce a framework that assigns a *weight* to each process and each configuration of an *information gathering algorithm*. Typical routing and census algorithms are in this category. We extend the definition given by Delaët and Tixeuil for their census algorithm [9] as follows.

The weight of a process is *zero* if the process has gathered all the information it can obtain, *one* if the information it has is *reacheable*, i.e. partial but correct, *two* otherwise. Since, in the algorithms that we consider, processes do not delete complete correct information or accept wrong information, the weight of each process can only decrease monotonically toward *zero*.

The weight of a configuration is a string of all the weights of the processes in decreasing order (first *twos*, then *ones*, and lastly *zeros*). A string of *zeros* denotes a legitimate configuration. A string of *ones* and *zeros* denotes a configuration that does not contain any wrong information. Any weight containing a *two* denotes a configuration reached by a transient failure and therefore may violate the *initial safety* requirement. The weight of a process can never increase in these algorithms, combined with the convergence property of self-stabilizing system, forces that weight to eventually decrease.

The weight framework assists in proving the correctness of the algorithm and helps in characterizing convergence by comparing the strings. For two strings on a given number of processes, the string that has the greatest number of *twos* is heavier. If both strings have the same number of *twos*, then the string that has the greater number of *ones* is heavier. Thus, a higher weight directly denotes a configuration that is farther from a legitimate configuration.

C1	$A, B := \emptyset, \emptyset$
C2	<b>forall</b> $q \in N_p$ <b>do</b> $\text{read}(q); A := A \cup e_q$ <b>od</b>
C3	$A := A \setminus \langle p, *, * \rangle; A := A++\langle *, *, 1 \rangle$
C4	<b>forall</b> $q \in \text{processors}(A)$ <b>do</b> $B := B \cup \{\text{mindist}(q, A)\}$ <b>od</b>
C5	$B := B \cup \{p, x_p, 0\}; e_p := \text{initseq}(B)$
C6	<b>write</b>

Figure 1: Update protocol for process  $p$ .

This framework is particularly well adapted to fixed output algorithms that work by accumulating information. For example, in a census algorithm, each process gradually learns new process identifiers until it knows them all, but the algorithm has to make sure that if a process knows an identifier that does not exist, this identifier is eventually removed.

### The Update Protocol Example.

To show in details how initial configurations facilitate convergence, we now consider Dolev and Herman's *update protocol* [12], a minimum distance vector algorithm for asynchronous shared register systems, given in Figure 1. In this algorithm, each process  $p$  maintains a routing table  $e_p$  that contains, at most, one entry per process in the system. An entry consists of a process identifier, an originator process identifier and an integer distance. Each process periodically scans its neighbors' routing tables, merges them all in a set  $A$ , and extracts from  $A$  the shortest path to each process in  $A$ , which yields  $p$ 's new routing table.

We define the set  $\mathcal{I}$  of initial configurations for this algorithm as the configurations where all the processes have an empty routing table, in accordance with the definition of an initial configuration.

Indeed, a process can easily delete its routing table when a reset occurs. It is also clear that an empty routing table does not contain any wrong information. Throughout any execution started in a configuration of  $\mathcal{I}$ , entries are added to the routing tables in a way that preserves the following invariants:

- **Subset validity:** only correct information is obtained. If the routing table of process  $p$  contains  $\langle q, x_q, d \rangle$ , then there is a path from  $p$  to  $q$  whose first component is  $x_q$  and whose length is  $d$ .
- **Distance monotonicity:** the information remains transitively distributed. If the routing table of process  $p$  contains  $\langle q, x_q, d \rangle$  with  $d > 1$ , then there exists a neighbor  $r$  of  $p$  such that the routing table of  $r$  contains  $\langle q, x_q, d' \rangle, d' < d$ .

The closure and convergence properties of the Update algorithm have already been proven [12]. In Lemmas 1 and 2, we prove its initial safety, i.e. that the system never acquires wrong information.

**Lemma 1.** *In any failure-free execution starting in an initial configuration, all the routing table entries preserve the subset validity invariant.*

*Proof.* The only line in which the routing table of  $p$  is modified is C5, where it is given the value of  $A$ . In turn,  $A$  only contains a reference to  $p$  itself (C5) and all the other information is extracted from the neighbors' routing tables (C2). Since the `initseq` operator (C5) only deletes suboptimal paths, only paths that really exist in the network topology are kept.  $\square$

**Lemma 2.** *In any failure-free execution starting in an initial configuration, all the routing table entries preserve the distance monotonicity invariants.*

*Proof.* The only line in which the routing table of  $p$  is modified is C5, where it is given the value of  $A$ . In turn,  $A$  only contains a reference to  $p$  itself with distance 0 (C5) and all the other information is extracted from the neighbors' routing tables (C4), adding 1 to all the distances. Since the `initseq` operator only deletes suboptimal paths, applying it cannot break distance monotonicity.  $\square$

### 3 Detecting Stabilization

Initial configurations can also be used to design *self-stabilizing stabilization detectors* that provide a boolean output over an algorithm  $\mathcal{A}$  to estimate whether  $\mathcal{A}$  is in a legitimate configuration. When the system is started in an arbitrary configuration, this detector typically can output wrong answers for an unknown number of steps before eventually giving the right answer forever. We show in this section that starting the system in an initial configuration allows to overcome this drawback and obtain a stabilization detector that outputs *false* as long as the system has not converged and eventually outputs *true* forever when the system has converged. In other words, if the system is started in an initial configuration, the detector is reliable, thereby increasing the safety of the system.

In many self-stabilizing algorithms, the state of each process progresses until it becomes *canonical*, as described by Delaët and Tixeuil [9], such that the system is in a safe configuration when all the processes are in a canonical state. This property was used, for example, by Dolev and Tzachar [14] to design a method that, with certain assumptions, transforms probabilistic self-stabilizing algorithms that use an infinite number of random bits so that they use a bounded number of random bits. We made use of this property to define the weight framework, and our stabilization detector is built on the same grounds.

$\begin{aligned} \text{stabilized} &= \forall q \in \text{processors}(e_p), \forall n \in \text{neighbors}(q), \langle n, x_n, d \rangle \in e_p \text{ s.t.} \\ &\quad d = \text{mindist}_{\text{id}}(n, e_p) \wedge x_p = \text{minproc}_{\text{id}}(n, e_p) \\ \\ \text{legitimate} &= \text{stabilized}_p \bigwedge_{q \in \text{processors}(e_p)} \text{stabilized}_q \end{aligned}$
---

Figure 2: Stabilization Detector algorithm.

### Fair Composition.

Let  $\mathcal{A}$  be a self-stabilizing algorithm, let  $\mathcal{B}$  be an algorithm that has a read-only access to the variables of  $\mathcal{A}$  such that if  $\mathcal{A}$  is self-stabilizing, then  $\mathcal{B}$  is self-stabilizing. Then, as shown by Dolev, Israeli and Moran [13], the *fair composition*  $\mathcal{A} \oplus \mathcal{B}$  obtained by the concatenation of the code and variables of  $\mathcal{A}$  and  $\mathcal{B}$  is a self-stabilizing algorithm. This property is useful for writing modular algorithms and reusing previously written code.

The definition of weights can be extended to composed self-stabilizing algorithms. Intuitively, it is easy to see that a global legitimate configuration requires all the composed systems to be in a legitimate configuration. Similarly, the global system contains only correct information if and only if each individual process only contains correct information.

Formally, let  $P = \oplus_{i \in [1, m]} P_i$ . A configuration  $c$  of  $P$  is an initial configuration if and only if in  $c$ , for all  $i$ ,  $P_i$  is in an initial configuration. The weight of a configuration  $c$  of  $P$  is a vector of  $w_j$  such that for each  $j$ ,  $w_j(P) = \max_{i \in [1, m]} w_j(P_i)$ .

### Self-stabilizing Stabilization Detector.

We assume that processes acquire the neighbor list of any process along with its identifier. This means that knowing a process implies also knowing the identifiers of all its neighbors.

The stabilizing stabilization detector (SD) is specified to work as follows:

- when the system is started in an arbitrary configuration, the SD eventually outputs *true* forever after the update algorithm has reached a legitimate configuration;
- when the system is started in an initial configuration, the output of the SD is always false until a legitimate configuration is reached, then eventually true.

The stabilization detector for the update algorithm is implemented as shown in Figure 2. The *minproc* function returns the process to contact in order to reach a given process, taking the implementation of the set functions into account. Each process has a *stabilized* register that indicates whether the local state of the process is the one that belongs in a legitimate configuration.

Then, the *legitimate* predicate returns whether the process estimates that the global configuration is legitimate. It does so by checking that all the local *stabilized* predicates are true. It is trivial to see that the stabilization detector returns *true* in a legitimate configuration, since the detector essentially checks all the criteria for a configuration to be legitimate.

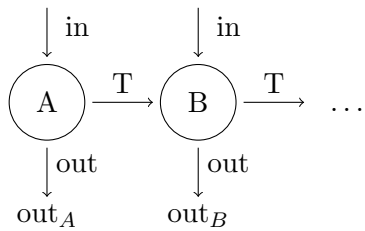
The stabilization detector also returns true in configurations where the routing tables contain wrong information about their neighbors. Since this wrong information is eliminated during convergence, the output can only be wrong for a limited number of steps.

As we showed, if the system is started in an initial configuration, then no wrong information is ever acquired. More precisely, no nonexistent process is ever added to a routing table and no distance can be underestimated. Due to the asynchronous scheduling, distances can be overestimated, but this does not make the stabilization detector output a wrong value. Indeed, the stabilized predicate only evaluates to true if all the routing table entries satisfy the minimal distance condition. Suppose that there is a path  $p, p_1, p_2, \dots, q$  from  $p$  to  $q$  with a lower distance than the current value in the routing table. Then, this path is partially present in the routing table of  $p$ . At the very least,  $p$  is present in its own routing table and knows that  $p_1$  is one of its neighbors. Hence, for some  $i$ , there is a  $p_i$  in the minimal path such that at least one neighbor of  $p_i$  is not in the routing table of  $p$  or does not satisfy the minimum distance requirement. As a consequence, if the system is started in an initial configuration, the output can never be wrong.

**Fair Composition with a Stabilization Detector.** In the general case of two composed algorithms,  $\mathcal{A}$  and  $\mathcal{B}$ , both started in an initial configuration, the system as a whole benefits from the added safety. The point is that  $\mathcal{B}$  can trust the values of the variables of  $\mathcal{A}$ . For example, if  $\mathcal{A}$  provides a broadcast service using a topology that it builds,  $\mathcal{B}$  can use it knowing that messages may not reach all the processes, but will not be duplicated, as could happen if the topology were, at first, arbitrary. If  $\mathcal{A}$  is a graph coloring algorithm, then  $\mathcal{B}$  knows that the color provided by  $\mathcal{A}$  is different from that of the neighbors, even though there is no optimality guarantee at that point, and thus the color might change later.

## 4 Safer than Safe: the Modular Redundancy Case Study

So far, we described the properties of self-stabilizing algorithms with reference to their initial configurations. We now show that taking initial safety and initial configurations into account when designing a new algorithm results in better fault tolerance. Specifically, our algorithm combines the fault-masking properties of a redundant system with the recovery abilities, even



(a) Principle of a state machine.

#### Variables

state: process state  
 r:  $\llbracket 0, m \rrbracket$  (\*  $m > n$  \*)

#### Code

```

r ← maj(mem.r) + 1
if (r mod n = 0) then begin
  state ← maj(states)
  state ← T(state,in)
end
mem[p + (r mod n)].out ← out(state)
mem[p + (r mod n)].r ← r
  
```

(b)  $n$ -modular redundancy algorithm.

Figure 3: State Machine, Modular Redundancy Algorithm

in a system overwhelmed with failures, provided by self-stabilization.

In this section, we present a classical masking fault-tolerant algorithm that is not self-stabilizing and we demonstrate its lack of safety in the presence of an unexpected number of stopping failures. We then present our self-stabilizing version of this algorithm and show that, while still masking the same classes of faults, it comes with added safety in the presence of more serious failures: the system can withstand them as long as some correct information remains, and if there is no more correct information, at least it is guaranteed that the system will converge to some consistent state.

The  $n$ -MR principle was identified very early as a fundamental tool for fault tolerance [20]. Nowadays, it is used both in hardware and in software. In hardware, it is used to compensate for very harsh, high-noise working conditions, e.g. in satellites [4]. In software, it can be used in setups where one or several machines can experience temporary failures, including large scale systems [19]. A typical value for  $n$  is 3, hence the common name *triple modular redundancy* (TMR). In the rest of the section, we use  $n$  in definitions and algorithms and assume that  $n = 3$  in examples.

The same redundancy principle can be applied *inside* each process, as in tri-redundancy [16]. This yields a system in which each component attempts to compensate for failures by itself, whereas in modular redundancy, the system globally masks the failures of individual components. Both approaches have advantages of their own: tri-redundant systems, as a fine-grained approach, works well on parallel systems, while the coarser-grained modular redundancy is better suited for distributed systems.

The  $n$ -MR algorithm transforms a regular state machine, as shown in Figure 3(a), into a replicated state machine. A state machine can receive input data and change state accordingly, e.g. from A to B in Figure 3(a). Its *transition function*  $T$  maps each state and input to a new state (possibly the same). An output function, called *out*, maps states to output data.

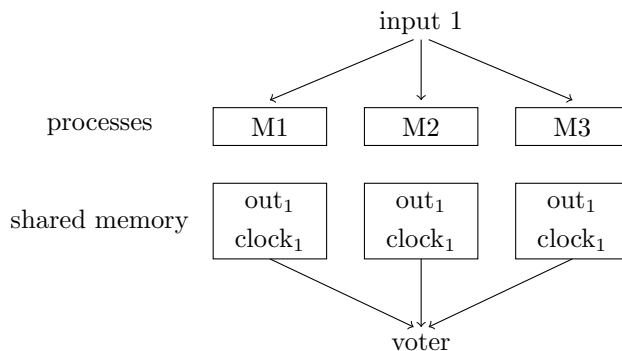


Figure 4:  $n$ -modular redundancy system.

The  $n$ -MR system, shown in Figure 4, is synchronous. It consists of an *input*, common to all the processes, an odd number  $n$  of computing processes, a *voter* that outputs the majority value among the  $n$  processes, and a shared memory consisting of  $n$  cells, each one containing an output value and a clock value in  $\llbracket 0, m \rrbracket$ , where  $m > n$ . Any process can read and modify the contents of any cell of the shared memory, but if two concurrent write operations occur simultaneously, the result is undefined.

Classically, this system works as follows. Consider a state machine with transition function  $T$  and a function  $out$  that maps states to outputs. The  $n$ -MR system replicates this state machine  $n$  times and allows to tolerate faults in a minority of the machines.

Each transition of the original state machine becomes a *cycle* of the  $n$ -MR system. During each cycle, an input value  $\mathbf{in}$  is received. Each of the processes changes its internal state according to  $T$  and calculates a new output value. The voter then outputs the majority value. If one of the computing processes fails in any way, providing a majority of the others do not fail, their correct output provides a sufficient majority. Thus,  $n$ -MR is a *masking* fault tolerance technique, in that under the minimal functionality assumption that a majority of processes never fails, the user connected to the output of the voter is in no way aware of the failures. The input circuit and the voter circuit are very simple and robust, and thus are assumed not to fail.

This setup is clearly meant to add safety to the original state machine. The property that should be enforced throughout the execution is that the voter produces the output that would be given by the original state machine in the absence of failure.

Let us now consider a self-stabilizing version of this algorithm, as shown in Figure 3(b). Every execution is organized in cycles of  $n$  rounds. Each round consists of loading the most common clock value (an arbitrary rule breaks ties) and incrementing it modulo  $m$ , an integer greater than  $n$ . Then, if the round is the first of a new cycle, each process  $p$  loads a state consistent

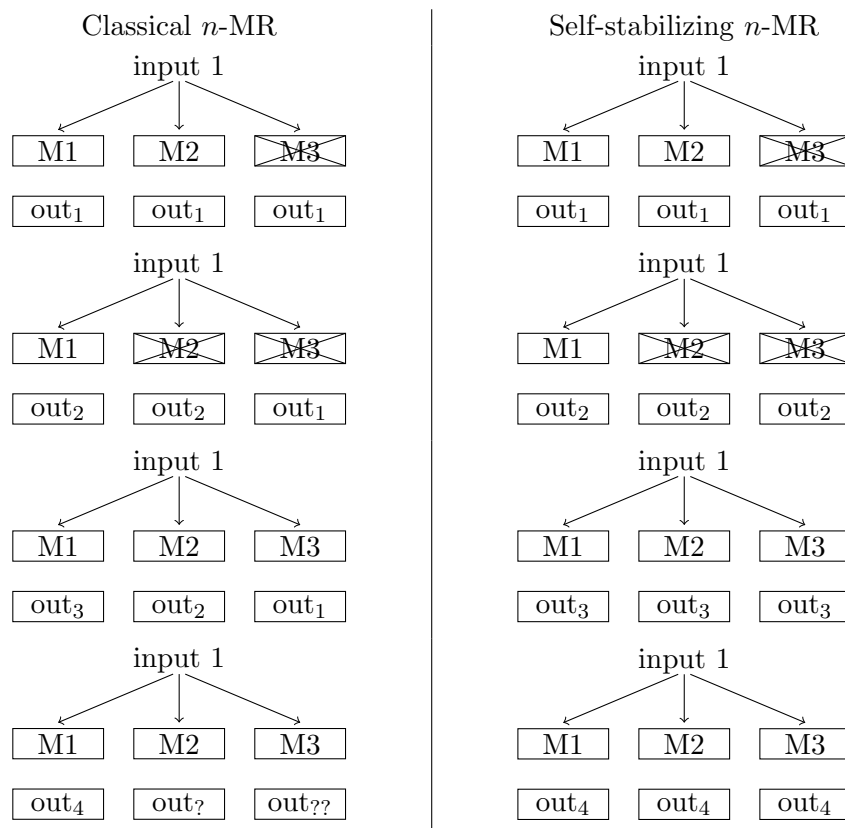


Figure 5: Classical  $n$ -MR versus self-stabilizing  $n$ -MR

with the other processes. Again, the most common state is chosen. In case more than one value is most common, a deterministic rule is used to choose one of them. Then,  $p$  reads its input and changes its state accordingly, thus calculating its new output,  $\mathbf{out}(\mathbf{state})$ . The round ends with  $p$  writing its output and its clock value in one memory cell determined uniquely by the identifier of  $p$  and the round number, thereby ensuring, in the absence of failures, that exactly one process writes in each memory cell.

As an immediate consequence, it is enough that one process remains alive for at least  $\lceil \frac{n}{2} \rceil$  rounds for the voter to output the right answer. It is also possible for a collection of processes to write the right outputs in the shared memory in different rounds, if some processes crash and others recover: since all the processes choose the same state from which to resume, they cannot erase a correct value. This improves over the classical modular redundancy, which does not consider recoveries after crashes. Moreover, even if consistency is violated, leading to a wrong majority vote, the transition function is eventually respected so that the sequence of states forms an execution.

An execution is given in Figure 5. The left column shows the self-stabilizing algorithm, while the right column shows the classical algorithm. In both cases, M3 stops in the first step, M2 stops in the second step, and both recover in the third step. In the case of the self-stabilizing algorithm, since all the processes load the same stored state in the beginning of each round, this results in a configuration in which all the processes are in the same state. In other words, the system has fully recovered from these failures. The classical algorithm, on the other hand, enters an inconsistent configuration in which no two processes have the same state. Since, in the absence of failures, all the processes will now apply their transition function to identical inputs, the system may not recover forever.

**Proof sketch for self-stabilization.** For initial safety, it is easy to see that a system started with all the processes in the same state behaves like a regular modular redundancy system. Convergence is achieved as follows: at latest, the next time that the value of the round counter ( $r \bmod n$ ) is 0, all the live processes load the same clock value and the same state at the same time. This yields a configuration where all the processes are in the same state, i.e. a legitimate configuration. Closure is an immediate consequence of the specification of the state machines: since all the processes have the same transition function and the system is synchronous, in all subsequent configurations, all the processes are in the same state.

**Benefits of this approach.** Suppose that this algorithm is started in an initial configuration, and crash/recovery failures may occur. It then provides the same service as the non-stabilizing algorithm, tolerating  $n - 1$  stopping failures. If transient failures change the state of any minority of the processes, then there remains a majority of correct outputs. In addition, since the system is self-stabilizing, it provides a protection against any combination of transient failures, even transient failures that corrupt the state of all the processes. All the processes eventually load the same arbitrary state simultaneously, thus recovering from these failures in the sense that they start to implement an execution of the simulated machine from an arbitrary state. In case all the processes fail, at least they eventually become synchronized again, which is better than behaving erratically, as does a non-stabilizing system.

## 5 Conclusion

We devoted this paper to a study of the safety of self-stabilizing systems. Our main contribution is the definition of *initial configurations*, i.e. preferred starting configurations, from which a self-stabilizing algorithm can safely

converge toward a legitimate configuration. Evidently, being self-stabilizing, the algorithm converges if initialized in any other configuration too.

This allows to compare self-stabilizing and non-stabilizing algorithms on an equal grounds. Non-stabilizing algorithms take advantage of their initial state to provide safety, so can self-stabilizing algorithms.

We gave a generic framework for analyzing self-stabilizing information gathering algorithms with reference to their initial configurations and we applied this framework to composed algorithms and stabilization detection.

As an example, we introduce self-stabilizing modular redundancy. This system is safer than usual NMR, since our system not only masks failures just like a normal NMR, it also adds eventual safety if faults that no algorithm could mask overwhelm the system.

We believe that specifying initial configurations with self-stabilizing algorithms should be standard. It provides the user who wants to run the algorithm with instructions on how to start the system in order to obtain the best possible conditions, combining safety, in normal conditions, and eventual safety, in the presence of a very high number of failures.

To sum up, we argue that self-stabilizing systems should no longer be regarded as only eventually safe, but as safe **and** eventually safe.

## References

- [1] Y. Afek and S. Dolev. Local stabilizer. *J. Parallel Distrib. Comput.*, 62(5):745–765, 2002.
- [2] A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:316–331, 1990.
- [3] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network RESET (extended abstract). In *PODC*, pages 254–263, 1994.
- [4] R. Banu and T. Vladimirova. On-board encryption in earth observation small satellites. In *40th Annual IEEE International Carnahan Conference on Security Technology*, pages 203–208, 2006.
- [5] J. Beauquier, S. Delaët, S. Dolev, and S. Tixeuil. Transient fault detectors. *Distributed Computing*, 20(1):39–51, 2007.
- [6] A. Bejan, S. Ghosh, and S. Rao. An extended framework of safe stabilization. In D. J. Jackson, editor, *Computers and Their Applications*, pages 276–282. ISCA, 2006.
- [7] A. Cournier, A. K. Datta, F. Petit, and V. Villain. Enabling snap-stabilization. In *ICDCS*, pages 12–19. IEEE Computer Society, 2003.
- [8] A. Cournier, S. Devismes, and V. Villain. From self- to snap- stabilization. In *SSS*, pages 199–213, 2006.
- [9] S. Delaët and S. Tixeuil. Tolerating transient and intermittent failures. *Journal of Parallel and Distributed Computing*, 62(5):961–981, 2002.

- [10] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17(11):643–644, 1974.
- [11] S. Dolev. *Self-Stabilization*. MIT Press, 2000.
- [12] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.
- [13] S. Dolev, A. Israeli, and S. Moran. Self stabilization of dynamic systems. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.
- [14] S. Dolev and N. Tzachar. Randomization adaptive self-stabilization. *CoRR*, abs/0810.4440, 2008.
- [15] S. Ghosh and A. Bejan. A framework of safe stabilization. In S.-T. Huang and T. Herman, editors, *Self-Stabilizing Systems*, volume 2704 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 2003.
- [16] M. G. Gouda, J. A. Cobb, and C.-T. Huang. Fault masking in tri-redundant systems. In A. K. Datta and M. Gradinariu, editors, *SSS*, volume 4280 of *Lecture Notes in Computer Science*, pages 304–313. Springer, 2006.
- [17] T. Herman and S. V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2):41–46, 2000.
- [18] C.-T. Huang and M. G. Gouda. State checksum and its role in system stabilization. In *ICDCS Workshops*, pages 29–34. IEEE Computer Society, 2005.
- [19] I. ling Yen. Specialized n-modular redundant processors in large-scale distributed systems. In *Proceedings of the 1996 15 th Symposium on Reliable Distributed Systems*, pages 12–21, 1996.
- [20] R. E. Lyons and W. Vandervulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, pages 200–209, 1962.
- [21] E. Normand. Single event upset at ground level. *IEEE Trans. Nuclear Science*, 43:2742–2751, 1996.