Self-stabilizing Algorithms

Sébastien Tixeuil

Self-stabilizing Algorithms

Sébastien Tixeuil

The final version of this book chapter appears in [72].

1 Introduction

The study of distributed systems and algorithms helps in understanding the specific features of these systems compared to classic centralized systems: information is local (each element of the system only holds a fraction of the information, and must obtain more by communicating with other elements), and time is local (the elements of the system can run their instructions at different speeds). These two factors result in non-deterministic behaviors, as two consecutive executions of the same distributed system are likely to be different. The fact that certain elements of the system can become faulty increases even further this non-determinism and the difficulty of predicting the overall system's behavior.

When the number of components in a distributed system is increased, the possibility for one or several of these components to become faulty also increases. When the production costs of these components are reduced to achieve economies of scale, the rate of potential defects again increases. Finally, when the system's components are deployed in an environment that is not necessarily controlled, the risks of faults occurring become impossible to overlook.

1.1 Fault taxonomy in distributed systems

A first criterion for classifying faults in distributed systems is localization in *time*. Usually, three types of possible faults are distinguished:

1. *transient faults*: faults that are arbitrary in nature can strike the system, but there is a point in the execution beyond which these faults no longer occur;
2. *permanent faults*: faults that are arbitrary in nature can strike the system, but there is a point in the execution beyond which these faults always occur;
3. *intermittent faults*: faults that are arbitrary in nature can strike the system, at any moment in the execution.

Transient fault and permanent faults are, of course, specific cases of intermittent faults. However, with a system in which intermittent faults rarely occur, a system that tolerates transient faults can be useful, because the useful lifespan can be long enough.

A second criterion is the *nature* of the faults. An element of the distributed system can be represented by an automaton, whose states represent the possible values of the element's variables, and whose transitions represent the code run by the element. We can then distinguish the following faults depending on whether they involve the state or the code of the element:

1. *state related faults*: changes in an element's variables may be caused by disturbances in the environment (electromagnetic waves, for example), attacks (buffer overflow, for example) or simply faults on the part of the equipment used. For example, it is possible for some variables to have values that they are not supposed to have if the system is running normally;

2. *code-related faults*: an arbitrary change in an element's code is most often the result of an attack (the replacement, for example, of an element by a malicious opponent), but certain, less serious types correspond to bugs or a difficulty in handling the load. There are several different sub-categories of code-related faults:

   (a) *crashes*: at a given moment during the execution, an element stops its execution permanently and no longer performs any action;

   (b) *omissions*: at different moments during the execution, an element may omit to communicate with the other elements of the system, either in transmission, or in reception;

   (c) *duplications*: at different moments during the execution, an element may perform an action several times, even though its code states that this execution must be performed once;

   (d) *desequencing*: at different moments during the execution, an element may perform the right actions, but in the wrong order;

   (e) *Byzantine faults*: these simply correspond to an arbitrary type of fault, and are therefore the faults that cause the most harm.

Crashes are included in omissions (an element that no longer communicates is perceived by the rest of the system as an element that has ended its execution). Omissions are trivially included in Byzantine faults. Duplications and desequencing are also included in Byzantine faults, but are generally regarded as behaviors strictly related with communication capabilities.

A third criterion is the *extent* (or *span*) of the faults, *i.e.*, how many of the individual system components can be hit by faults or attacks.

1.2 Fault-tolerant algorithm categories

When faults occur on one or several of the elements that comprise a distributed system, it is essential to be able to deal with them. If a system tolerates no fault whatsoever, the failure of a single one of its elements can compromise the execution of the entire system: this is the case for a system in which an entity has a central role (such as the DNS). In order to preserve the system's useful lifespan, several ad hoc methods have been developed, which are usually specific to a particular type of fault that is likely to occur in the system in question. However, these solutions can be categorized depending on whether the effect is visible or not to an observer (a user, for example). A masking solution hides the occurrence of faults to the observer, whereas a non-masking solution does not present this characteristic: the effect of faults is visible over a certain period of time, then the system resumes behaving properly.

A masking approach may seem preferable at first, since it applies to a greater number of applications. Using a non-masking approach to regulate air traffic would make collisions possible following the occurrence of faults. However, a masking solution is usually more costly (in resources and in time) than a non-blocking solution, and can only tolerate faults

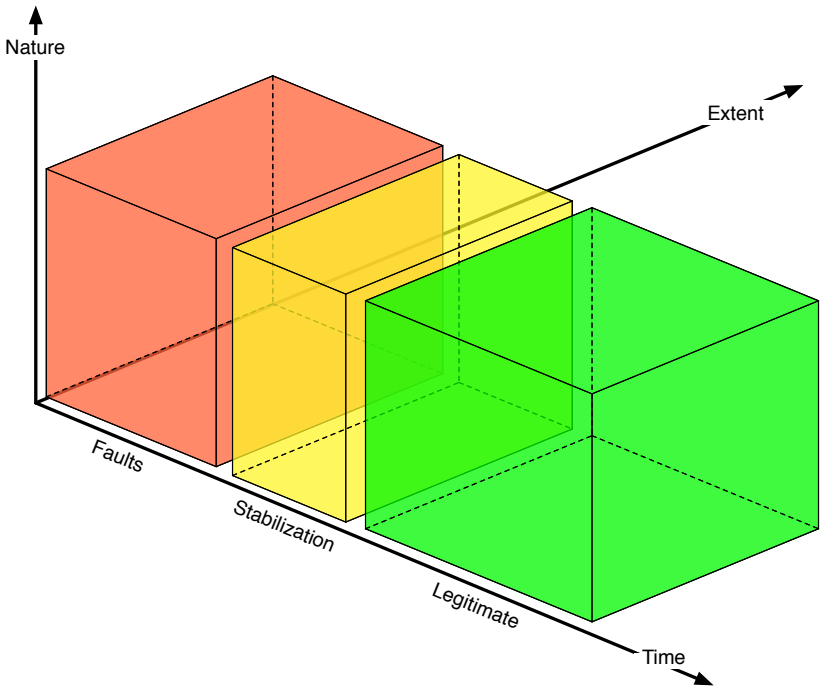
so long as they have been anticipated. For problems such as routing, where being unable to transport information for a few moments will not have catastrophic consequences, a non-masking approach is perfectly well-suited. Two major categories for fault-tolerant algorithms can be distinguished:

1. *robust algorithms*: these use redundancy on several levels of information, of communications, or of the system's nodes, in order to overlap to the extent that the rest of the code can safely be executed. They usually rely on the hypothesis that a limited number of faults will strike the system, so as to preserve at least a majority of correct elements (sometimes more if the faults are more severe). Typically, these are masking algorithms.

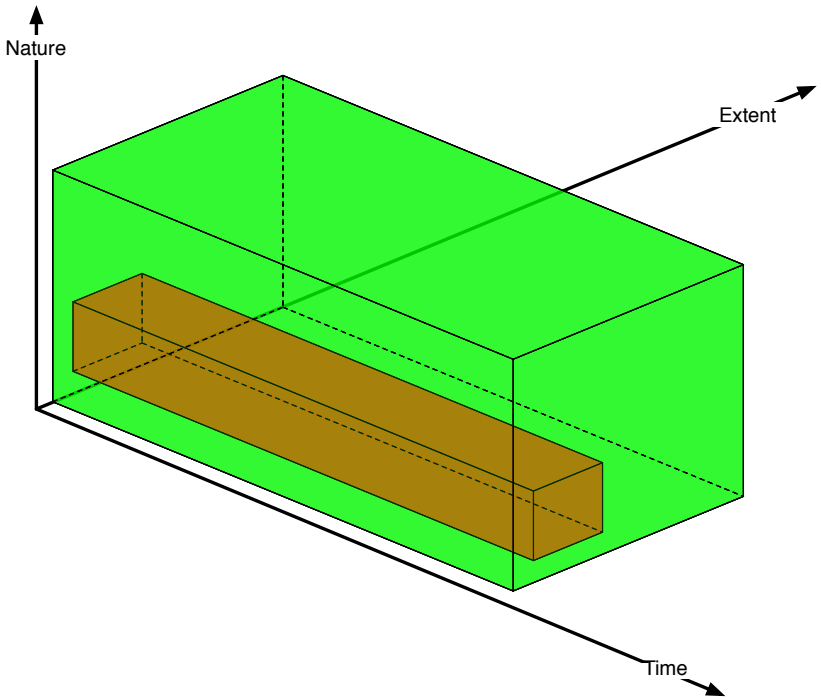
2. *self-stabilizing algorithms*: these rely on the hypothesis that the faults are transient (in other words, limited in time), but do not set constraints regarding the extent of the faults (which may involve all of the system's elements). An algorithm is self-stabilizing [21, 22] if it manages, in a finite time, to present an appropriate behavior independently from the initial state of its elements, meaning that the variables of the elements may exist in a state that is arbitrary (and impossible to achieve by running the application normally). Typically, self-stabilizing algorithms are non-masking, because between the moment when the faults cease and the moment when the system has stabilized to an appropriate behavior, the execution may turn out to be somewhat erratic.

Robust algorithms are quite close to what we conceive intuitively as fault-tolerance. If an element is susceptible to faults, then each element is replaced with three identical elements, and each time an action is undertaken, the action is performed three times by each of the elements, and the action actually undertaken is the one that corresponds to the majority of the three individual actions. Self-stabilization would seem to be related more to the concept of convergence in mathematics or control theory, where the objective is to reach a fixed point regardless of the initial position; the fixed point corresponds here to an appropriate execution. Being capable of starting with an arbitrary state may seem odd (since it would seem that the initial states of the elements are always well known), but studies [73] have shown that if a distributed system is subjected to stopping and restarting-type node failures (which correspond to a definite failure followed by a reinitialization), and communications cannot be totally reliable (some communications may be lost, duplicated or, desequenced), then an arbitrary state of the system can actually be achieved. Even if the probability of the execution that leads to this arbitrary state is negligible in normal conditions, it is not impossible for an attack on the system to attempt to reproduce such an execution. In any case, and regardless of the nature of what led the system to this arbitrary state, a self-stabilizing algorithm is capable of providing an appropriate behavior in a finite amount of time. In fact, self-stabilizing distributed algorithms are found in a number of protocols used in computer networks [48, 66].

Figure 1 sums up the relative capabilities of self-stabilizing and robust algorithms, respectively. The three axes take into account the three possibilities to classify faults in distributed systems that are described in Section 1.1. With a self-stabilizing algorithm, an external user may experience erratic behavior (the stabilizing phase) after the faults have actually ceased, while a robust algorithm will always appear as behaving properly. In contrast, a self-stabilizing algorithm makes no assumption about the extent or the nature of the faults, while robust systems will generally put constraints on those. The rest of the chapter is organized as follows: Section 2 presents the most common hypotheses that are made in the



(a) Self-stabilizing algorithms



(b) Robust algorithms

Figure 1: Self-stabilization *vs.* Robustness

self-stabilizing literature. Section 3 gives several examples of self-stabilizing algorithms, using various kinds of problems, hypotheses, and proof techniques. Section 4 presents the main variants of self-stabilization and concludes the chapter.

2 Models

Traditionally, a distributed system is usually represented by a graph, in which the nodes are the system's machines and the edges represent the ability for two machines to communicate. Thus, two machines are connected if they are capable of communicating information to one another (using a network connection for example). In some cases, the edges of the graph are oriented so as to represent the fact that the communication can only take place one way (for example, wireless communication from a satellite to an antenna on the ground). From now on, we will indiscriminately use the words machine, node or process depending on the context.

2.1 System hypotheses

In the context of self-stabilization, the hypotheses made for the system generally do not include, as with robust algorithms, conditions on the completeness or the globality of the communications. Many algorithms run on systems with nodes that only communicate locally. However, several hypotheses may be crucial for the algorithm to run properly, and involve the hypotheses made regarding the scheduling of the system:

1. *atomicity of the communications*: most of the self-stabilizing algorithms discussed in the literature use communication primitives with a high level of atomicity. At least three historic models are found in the literature:

   (a) *the state model* (or shared memory model [21]): in one atomic step, a node can read the state of each of the neighboring nodes, and update its own state;

   (b) *the shared register model* [26]: in one atomic step, a node can read the state of one of its neighboring nodes, or update its own state, but not both simultaneously;

   (c) *the message passing model* [1, 27, 52]: this is the classic model for distributed algorithms, for which in one atomic step, a node sends a message to one of the neighboring nodes, or receives a messages from one of the neighboring nodes, but not both simultaneously.

   With the recent study of the self-stabilization property in wireless and ad hoc sensor networks, several models for local diffusion with potential collisions have appeared. In the model that presents the highest degree of atomicity [55], a node can, in one atomic step, read its own state and partially write the state of each of the neighboring nodes. If two nodes simultaneously write the state of a common neighbor, a collision occurs and none of the information is written. A more realistic model [45] consists of defining two distinct and atomic actions for local diffusion on one hand and the reception of a locally diffused message on the other.

   In the case of bidirectional communications, it is possible to simulate a model using another model. For example, [22] shows how to transform the shared memory model into a message passing model. In the models that are specific to wireless networks, [54]

shows how to transform the local diffusion model with collisions into a shared memory model; in a similar fashion [43] shows that the model in [45] can be transformed into a shared memory model. There are two problems with these transformations:

(a) the transformation uses up resources (time, memory, energy in the case of sensors), which could be avoided using a direct solution in the model closest to the considered system;

(b) the transformation is only possible in systems with bidirectional communications: this is due to the fact that acknowledgments have to be sent regularly to ensure that the highest level model is properly simulated.

2. *spatial scheduling*: historically, self-stabilizing algorithms relied on the hypothesis that two neighboring nodes cannot execute their codes simultaneously. This makes it possible to break symmetry in certain configurations [4, 67]. Usually, three main possibilities are distinguished for scheduling, depending on which constraints are wanted:

   (a) *central scheduling*: at a given moment, only one of the system's nodes can run its code;

   (b) *global (or synchronous) scheduling*: at a given moment, all of the system's nodes run their codes;

   (c) *distributed scheduling*: at a given moment, an arbitrary subset of the system's nodes runs its code. This type of spatial scheduling is the most realistic.

   Other variations are also possible (for example, a k-locally central scheduling [62]: at a given moment, in each neighborhood of node at distance at most k, only one of the nodes executes it code), but in practice, they are equivalent to one of the three models above (see [69, 71]). The more constrained the spatial scheduling model is, the easier it is to solve problems. For example, [4] shows that it is impossible to color an arbitrary graph in a distributed and deterministic fashion. On the other hand, [38] shows that if the spatial scheduling is locally central, then such a solution is possible. Some algorithms, which rely on the hypothesis of one of these models, can be run in another model, at the price of a greater consumption of resources, as before. Because the most general model is the distributed model, it may be transformed into a more constrained model using a mutual exclusion algorithm [21, 14, 40] (for the central model), or a synchronization algorithm [2] (for the global model).

3. *temporal scheduling*: the first self-stabilizing [21] algorithms were independent of the concept of time, that is, they were written in a purely asynchronous model, where no hypothesis is stated regarding the relative speeds of the system's nodes. Later on, scheduling models with heavier constraints began to appear, particularly for the description of real systems. Schedulers are usually divided into three main types:

   (a) *arbitrary (*aka *unfair, adversarial) scheduling*: no hypothesis is made regarding the respective execution properties of the system's nodes, other than the simple progression (at each moment, at least one node executes some actions);

   (b) *fair scheduling*: each node runs local actions infinitely often;

   (c) *bounded scheduling*: between the executions of two actions for the same system node, each node executes a bounded number of actions.

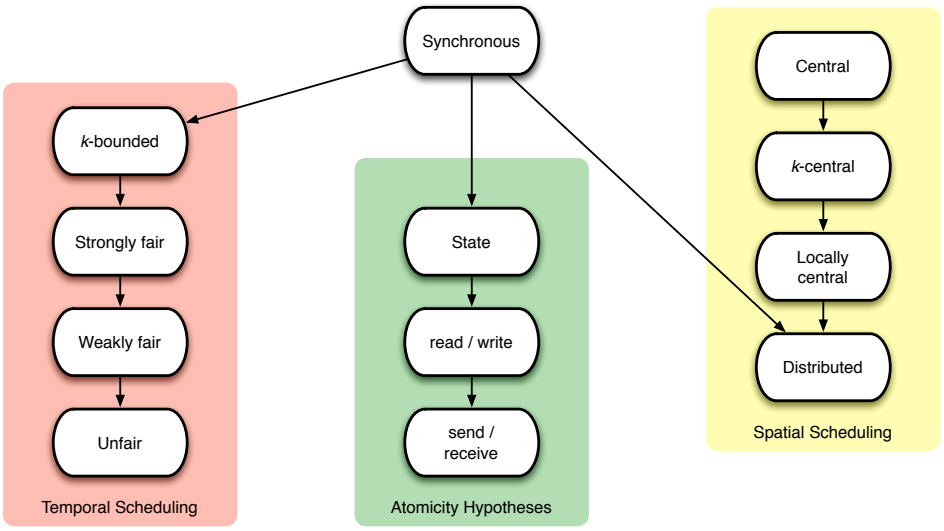


Figure 2: Taxonomy of system hypotheses in Self-stabilization

Other variations are possible (for example, [50] makes a distinction between weak fairness and strong fairness), but as before, the more constrained the temporal scheduling model is, the easier it is to solve problems. Bounded scheduling can be constrained further in order to obtain synchronous (or global) scheduling. As with the variations on the previous models, there are algorithms for transforming the execution from one model to another. For example, alternators [37, 47, 56] can be used to construct a bounded model based on a fair or arbitrary model. On the other hand, because of its unbounded nature, the strict fair model cannot be (strictly) constructed from the arbitrary model.

The general taxonomy of common system hypotheses made in self-stabilizing literature are presented in Figure 2. An arrow from a hypothesis a to a hypothesis b implies that a is stronger than b, that is an algorithm p_a assuming a is weaker than an algorithm p_b assuming b. That is, p_b will work under hypotheses b and a, but p_a will only work with hypothesis a. Since the kinds of scheduling presented in this section are quite different from the notion of scheduling used *e.g.* in parallel algorithms, the most used term to refer to those hypoteses in the self-stabilizing literature is the *daemon*. The two terms are used interchangeably in the remaining of the text.

2.2 Program model

For the formal description of our program we use simplified UNITY notation [35]. A program consists of a set of processes. A process contains a set of *constants* that it can read but not update. A process maintains a set of *variables*. Each variable ranges over a fixed domain of values. We use small case letters to denote singleton variables, and capital ones to denote sets. Some variables are persistent from one activation of the process to the next, and are called *state variables*, while some other are wiped out between two activations of a process, and are called *local variables*. When there is no ambiguity, the generic term *variable* refers to a state variable.

An action has the form $\langle name \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$. A *guard* is a Boolean predicate. In the shared memory model, this predicate is over the variables of the process and its

communication neighbors. In the shared register model, this predicate is over the variables of the process and a single input register. In the message passing model, this predicate is over the variables of the process and the **receive**(m) primitive. In this context, **receive**(m) evaluates to *true* if a message matching m has been received, and to *false* otherwise. A *command* is a sequence of statements assigning new values to the variables of the process (in all communication models) and sending messages (using the **send**(m) primitive) in the message passing model. Besides assignments, conditionals and loops are also available (with the **if-fi** and **do-od** constructions, respectively). We refer to a variable *var* and an action *ac* of process p as *var.p* and *ac.p* respectively. We make use of two operators: [] denotes *alternation* (that is $a[]b$ denotes the fact that either a or b is executed, but not both, and the choice is non-deterministic) while $*[]$ denotes *iteration* (that is, $*[a]$ denotes the fact that a is repetitively evaluated). A *parameter* is used to define a set of actions as one parametrized action. For example, let j be a parameter ranging over values 2, 5, and 9; then a parametrized action *ac.j* defines the set of actions: $ac.(j := 2)$ [] $ac.(j := 5)$ [] $ac.(j := 9)$.

A *configuration* of the system is the assignment of a value to every variable of each process from the variable's corresponding domain (and possibly a similar assignment of message contents in channels between processes in case of the message passing model). Each process contains a set of actions. An action is *enabled* in some configuration if its guard is *true* at this state. A *computation* is a maximal sequence of configurations such that for each configuration s_i, the next configuration s_{i+1} is obtained by executing the command of an action that is enabled in s_i. Maximality of a computation means that the computation is infinite or it terminates in a state where none of the actions are enabled. Such state is a *fix-point*.

A configuration *conforms* to a predicate if this predicate is **true** in this configuration; otherwise the state *violates* the predicate. By this definition every state conforms to predicate **true** and none conforms to **false**. Let R and S be predicates over the configuration of the system. Predicate R is *closed* with respect to the program actions if every configuration of the computation that starts in a configuration conforming to R also conforms to R. Predicate R *converges* to S if R and S are closed and any computation starting from a configuration conforming to R contains a configuration conforming to S. The program *stabilizes* to R if **true** converges to R.

Definition 1 (Self-stabilization) *Starting from an arbitrary initial configuration, any execution of a* self-stabilizing *algorithm contains a subsequent configuration from which every execution satisfies the specification.*

3 Designing self-stabilizing algorithms

In the context of self-stabilization, depending on the problem that we wish to solve, the minimum time required for going back to a correct configuration varies significantly. Problems are generally divided into two categories:

1. *static problems*: we wish to perform a task that consists of calculating a function that depends on the system in which it is assessed. For example, it can consist of coloring the nodes of a network so as to never have two adjacent nodes with the same color; another example is the calculation of the shortest paths to a destination [23].

2. *dynamic problems*: we wish to perform a task that performs a service for upper layer algorithms. The model transformation algorithms such as token passing fall into this category.

The example designs that are featured in this section are Maximal Matching (Section 3.1), Generalized Diners (Section 3.2), Census (Section 3.3), and Token Passing (Section 3.4). The first two are written for the state model (*a.k.a.* shared memory model) and the last two for the message passing model. Maximal Matching and Census are instances of static problems, while Generalized Diners and Token Passing are instances of dynamic ones. The proof techniques used to show the self-stabilization properties of the presented algorithms include attractors, potential functions, and Markov chains. The examples that are presented in this section previously appeared in [60, 12, 18, 29].

3.1 Maximal Matching

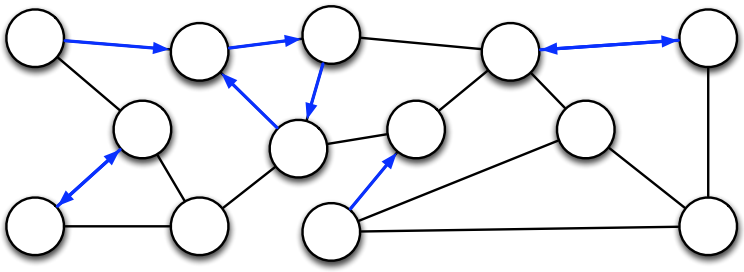
Algorithm. In the following we present and motivate our algorithm for computing a maximal matching. The algorithm is self-stabilizing and does not make any assumptions on the network topology. A set of edges $M \subseteq E$ is a *matching* if and only if $x, y \in M$ implies that x and y do not share a common end point. A matching M is *maximal* if no proper superset of M is also a matching. Note that a maximal matching differs from a *maximum* matching, that is required to have maximum cardinality among all possible maximal matchings. In the remaining of the section, n denotes the number of nodes and m denotes the number of edges, respectively.

Each process i has a variable *points_to*.i pointing to one of its neighbors or to *null*. We say that processes i and j *are married* to each other if and only if i and j are neighbors and their *points_to*-values point to each other. In this case we will also refer to i as being married without specifying j. However, we note that in this case j is unique. A process which is not married is *unmarried*. Figure 3 depicts two possible configurations of our algorithm: in Figure 3.(a), some nodes points at some other without the converse being true (leaving those nodes unmatched), while Figure 3.(b) depicts a maximal matching configuration.

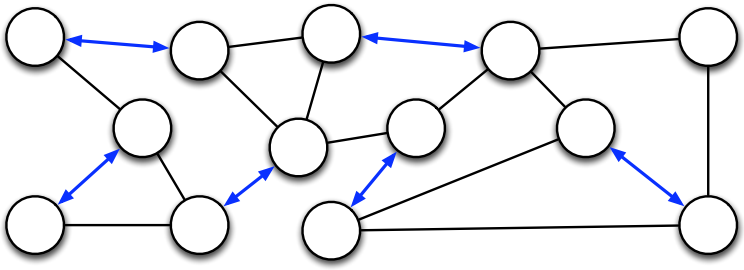
We also use a variable *matched*.i to let neighboring processes of i know if process i is married or not. To determine the value of *matched*.i we use a predicate *PRmarried(i)* which evaluates to true if and only if i is married. Thus predicate *PRmarried(i)* allows process i to know if it is currently married and the variable *matched*.i allows neighbors of i to know if i is married. Note that the value of *matched*.i is not necessarily equal to *PRmarried(i)*.

Our self-stabilizing scheme is given in Algorithm 1. It is composed of four mutually exclusive guarded rules as described below.

The *Update* rule updates the value of *matched*.i if it is necessary, while the three other rules can only be executed if the value of *matched*.i is correct. In the *Marriage* rule, an unmarried process that is currently being pointed to by a neighbor j tries to marry j by setting *points_to*.$i = j$. In the *Seduction* rule, an unmarried process that is not being pointed to by any neighbor, points to an unmarried neighbor with the objective of marriage. Note that the identifier of the chosen neighbor has to be larger than that of the current process. This is enforced to avoid the creation of cycles of pointer values. In the *Abandonment* rule, a process i resets its *points_to*.i value to *null*. This is done if the process j which it is pointing to does not point back at i and if either *(1)* j is married, or *(2)* j has a lower identifier than i. Condition *(1)* allows a process to stop waiting for an already married process while the purpose of Condition *(2)* is to break a possible initial cycle of *points_to*-values.



(a) A possible arbitrary initial configuration



(b) A legitimate maximal matching configuration

Figure 3: Possible configurations of Algorithm 1

We note that if *PRmarried(i)* holds at some point of time then from then on it will remain true throughout the execution of the algorithm. Moreover, the algorithm will never actively create a cycle of pointing values since the *Seduction* rule enforces that $j > i$ before process i will point to process j. Also, all initial cycles are eventually broken since the guard of the *Abandonment* rule requires that $j \leq i$.

Proof of Correctness. The proof of correctness presented in this section demonstrates how to cope with an adversarial daemon. Intuitively, we can only assume progress of the computation (*i.e.* at least one node that is activatable makes a move), so the main goal of the proof is to show that whatever the choices of the daemon are, only a finite number of moves can be made by each node before reaching a legitimate configuration, as there exists no chain of consecutive configurations that form a loop in the set of illegitimate configurations (see Figure 4). For the maximal matching problem, the situation is facilitated by the fact that a final configuration is eventually reached, in which a maximal matching is constructed.

In the following we will first show that when Algorithm 1 has reached a stable configuration it also defines a maximal matching. We will then bound the number of steps the algorithm needs to stabilize for the adversarial daemon. We now proceed to show that if Algorithm 1 reaches a stable configuration then the *points_to* and *matched*-values will define a maximal matching M where $(i,j) \in M$ if and only if $(i,j) \in E, points\_to.i = j$, and $points\_to.j = i$ while both $matched.i$ and $matched.j$ are true. A configuration is stable (or terminal) if no processes are eligible to execute a move. In order to perform the proof, we define the following five mutual exclusive predicates:

$$
\begin{aligned}
PRmarried(i) &\equiv \exists j \in N(i) : (points\_to.i = j \textbf{ and } points\_to.j = i) \\
PRwaiting(i) &\equiv \exists j \in N(i) : (points\_to.i = j \textbf{ and } points\_to.j \neq i \textbf{ and } \neg PRmarried(j)) \\
PRcondemned(i) &\equiv \exists j \in N(i) : (points\_to.i = j \textbf{ and } points\_to.j \neq i \textbf{ and } PRmarried(j))
\end{aligned}
$$

Algorithm 1 $\mathcal{MM}$: a self-stabilizing maximal matching algorithm

process i
const
N: communication neighbors of i
parameter
$r : N$
var
$matched.i : \{\mathbf{true}, \mathbf{false}\}$
$points\_to.i : \{\mathbf{null}\} \cup N$
predicate
$PRmarried(i) \equiv (points\_to.i = r \wedge points\_to.r = i)$

$*[$

update: $matched.i \neq PRmarried(i) \longrightarrow$
$matched.i := PRmarried(i)$

$[]$

marriage: $matched.i = PRmarried(i) \wedge points\_to.i = \mathbf{null} \wedge points\_to.r = i \longrightarrow$
$points\_to.i := r$

$[]$

seduction: $matched.i = PRmarried(i) \wedge points\_to.i = \mathbf{null} \wedge \forall k \in N : points\_to.k \neq i$
$\wedge(points\_to.r = \mathbf{null} \wedge r > i \wedge \neg matched.r) \longrightarrow$
$points\_to.i := Max\{j \in N : (points\_to.j = \mathbf{null} \wedge j > i \wedge \neg matched.j)\}$

$[]$

abandonment: $matched.i = PRmarried(i) \wedge points\_to.i = j \wedge points\_to.j \neq i$
$\wedge(matched.j \vee j \leq i) \longrightarrow$
$points\_to.i := \mathbf{null}$

$]$

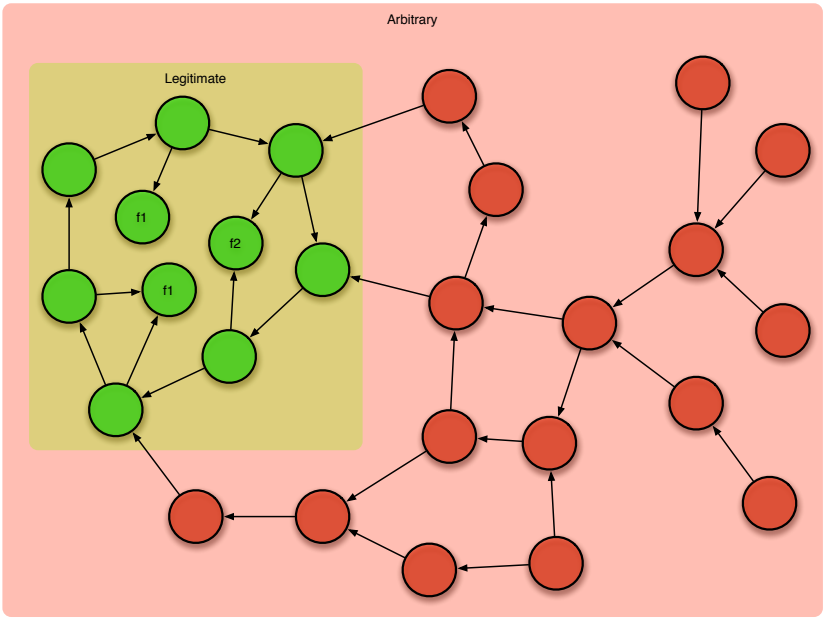


Figure 4: Proving Self-stabilization with unfair scheduling

$$PRdead(i) \equiv (points\_to.i = null) \textbf{ and } (\forall j \in N(i) : PRmarried(j))$$
$$PRfree(i) \equiv (points\_to.i = null) \textbf{ and } (\exists j \in N(i) : \neg PRmarried(j))$$

Note first that each process will evaluate exactly one of these predicates to true. Moreover, also note that *PRmarried(i)* is the same as in Algorithm 1. We now show that in a stable configuration each process i evaluates either *PRmarried(i)* or *PRdead(i)* to true, and when this is the case, the *points_to*-values define a maximal matching. To do so, we first note that in any stable configuration the *matched*-values reflects the current status of the process.

Lemma 1 *In a stable configuration we have* $matched.i = PRmarried(i)$ *for each* $i \in V$.

Proof: This follows directly since if $matched.i \neq PRmarried(i)$ then i is eligible to execute the *update* rule. □

We next show in the following three lemmas that no process will evaluate either $PRwaiting(i)$, $PRcondemned(i)$, or $PRfree(i)$ to true in a stable configuration.

Lemma 2 *In a stable configuration* $PRcondemned(i)$ *is false for each* $i \in V$.

Proof: If there exists at least one process i in the current configuration such that $PRcondemned(i)$ is true then $points\_to.i$ is pointing to a process $j \in N(i)$ that is married to a process k where $k \neq i$. From Lemma 1 it follows that in a stable configuration we have $matched.i = PRmarried(i)$ and $matched.j = PRmarried(j)$. Thus in a stable configuration the predicate $(matched.i = PRmarried(i) \wedge points\_to.i = j \wedge points\_to.j \neq i \wedge matched.j)$ evaluates to true. But then process i is eligible to execute the *Abandonment* rule contradicting that the current configuration is stable. □

Lemma 3 *In a stable configuration* $PRwaiting(i)$ *is false for each* $i \in V$.

Proof: Assume that the current configuration is stable and that there exists at least one process i such that $PRwaiting(i)$ is true. Then it follows that *points_to.i* is pointing to a process $j \in N(i)$ such that *points_to.j* $\neq i$ and j is unmarried. Note first that if *points_to.j* $=$ *null* then process j is eligible to execute a *marriage* move. Also, if $j < i$ then process i can execute an *abandonment* move. Assume therefore that *points_to.j* $\neq$ *null* and that $j > i$. It then follows from Lemma 2 that $\neg PRcondemned(j)$ is true and since j is not married we also have $\neg PRmarried(j)$. Thus $PRwaiting(j)$ must be true. Repeating the same argument for j as we just did for i it follows that if both i and j are ineligible for a move then there must exist a process k such that *points_to.j* $= k$, $k > j$, and $PRwaiting(k)$ also evaluates to true. This sequence of processes cannot be extended indefinitely since each process must have a higher identifier than the preceding one. Thus there must exist some process in V that is eligible for a move and the assumption that the current configuration is stable is incorrect. □

Lemma 4 *In a stable configuration $PRfree(i)$ is false for each $i \in V$.*

Proof: Assume that the current configuration is stable and that there exists at least one process i such that *PRfree(i)* is true. Then it follows that *points_to.i* $=$ *null* and that there exists at least one process $j \in N(i)$ such that j is not married.

Next, we look at the value of the different predicates for the process j. Since j is not married it follows that $PRmarried(j)$ evaluates to false. Also, from lemmas 2 and 3 we have that both $PRwaiting(j)$ and $PRcondemned(j)$ must evaluate to false. Finally, since i is not married we cannot have $PRdead(j)$. Thus we must have $PRfree(j)$. But then the process with the smaller identifier of i and j is eligible to propose to the other, contradicting the fact that the current configuration is stable. □

From lemmas 2 through 4 we immediately get the following corollary.

Corollary 1 *In a stable configuration either $PRmarried(i)$ or $PRdead(i)$ holds for every $i \in V$.*

We can now show that a stable configuration also defines a maximal matching.

Theorem 1 *In any stable configuration the matched and points_to-values define a maximal matching.*

Proof: From Corollary 1 we know that in a stable configuration either $PRmarried(i)$ or $PRdead(i)$ holds for every $i \in V$. Also, from Lemma 1 it follows that *matched.i* is true if and only if i is married. It is then straightforward to see that the *points_to*-values define a matching.

To see that this matching is maximal assume to the contrary that it is possible to add one more edge (i, j) to the matching so that it still remains a legal matching. To be able to do so we must have *points_to.i* $=$ *null* and *points_to.j* $=$ *null*. Thus we have $\neg PRmarried(i)$ and $\neg PRmarried(j)$ which again implies that both $PRdead(i)$ and $PRdead(j)$ evaluates to true. But according to the $PRdead$ predicate two adjacent processes cannot be dead at the same time. It follows that the current matching is maximal. □

In the following we will show that Algorithm 1 will reach a stable configuration after at most $3 \cdot n + 2 \cdot m$ steps under the distributed adversarial daemon.

First we note that as soon as two processes are married they will remain so for the rest of the execution of the algorithm.

Lemma 5 *If processes i and j are married in a configuration C, i.e. $points\_to.i = j$ and $points\_to.j = i$, then they will remain married in any ensuing configuration C'.*

Proof: Assume that $points\_to.i = j$ and $points\_to.j = i$ in some configuration C. Then process i can neither execute the *marriage* nor the *seduction* rule since these require that $points\_to.i = null$. Similarly, i cannot execute the *abandonment* rule since this requires that $points\_to.j \neq i$. The exact same argument for process j shows that j also cannot execute any of the three rules *marriage*, *seduction*, and *abandonment*. Thus the only rule that processes i and j can execute is *update* but this will not change the values of $points\_to.i$ or $points\_to.j$. □

A process discovers that it is married through executing the *update* rule. Thus this is the last rule a married process will execute in the algorithm. This is reflected in the following.

Corollary 2 *If a process i executes an* update *move and sets $matched.i = true$ then i will not move again.*

Proof: From the predicate of the *update* rule it follows that when process i sets $matched.i =$ true there must exist a process $j \in N(i)$ such that $points\_to.i = j$ and $points\_to.j = i$. Thus from Lemma 5 the only subsequent move i can make is an *update* move. But since the value of $matched.i$ is correct and $points\_to.i$ and $points\_to.j$ will not change again this will not happen. □

Since a married process cannot become "unmarried" we also have the following restriction on the number of times the *update* rule can be executed by any process.

Corollary 3 *Any process executes at most two* update *moves.*

We will now bound the number of moves from the set {*marriage*, *seduction*, *abandonment*}. Each such move is performed by a process i in relation to one of its neighbors j. We denote any such move made by either i or j with respect to the other as an *i, j-move*.

Lemma 6 *For any edge $(i, j) \in E$, there can at most be three steps in which an i, j-move is performed.*

Proof: Let $(i, j) \in E$ be an edge such that $i < j$. We then consider four different cases depending on the initial values of $points\_to.i$ and $points\_to.j$. Note from Algorithm 1 that the only values that $points\_to.i$ and $points\_to.j$ can take on are $points\_to.i \in \{null\} \cup N(i)$ and $points\_to.j \in \{null\} \cup N(j)$. For each case we will show that there can at most be three steps in which i, j-moves occur.

1. $points\_to.i \neq j$ and $points\_to.j \neq i$. Since $i < j$ the first i, j-move cannot be process j executing a *seduction* move. Also, as long as $points\_to.i \neq j$, process j cannot execute a *marriage* move. Thus process j cannot execute an i, j-move until after process i has first made an i, j-move. It follows that the first possible i, j-move is that i executes a *seduction* move simultaneously as j makes no i, j-move. Note that at the starting configuration of this move we must have $\neg matched.j$. If the next i, j-move is performed by j simultaneously as i performs no move then this must be a *marriage* move which results in $points\_to.i = j$ and $points\_to.j = i$. Then by Lemma 5 there will be no more i, j-moves. If process i makes the next i, j-move (independently of what process j does)

then this must be an *abandonment* move. But this requires that the value of $matched.j$ has changed from false to true. Then by Corollary 2 process j will not make any more i,j-moves and since $points\_to.j \neq null$ and $points\_to.j \neq i$ for the rest of the algorithm it follows that process i cannot execute any future i,j-move. Thus there can at most be two steps in which i,j-moves are performed.

2. $points\_to.i = j$ and $points\_to.j \neq i$. If the first i,j-move only involves process j then this must be a *marriage* move resulting in $points\_to.i = j$ and $points\_to.j = i$ and from Lemma 5 neither i nor j will make any future i,j-moves. If the first i,j-move involves process i then this must be an *abandonment* move. Thus in the configuration prior to this move we must have $matched.j =$ true. It follows that either $matched.j \neq PRmarried(j)$ or $points\_to.j \neq null$. In both cases process j cannot make an i,j-move simultaneously as i makes its move. Thus following the *abandonment* move by process i we are at Case *(1)* and there can at most be two more i,j-moves. Hence, there can at most be a total of three steps with i,j-moves.

3. $points\_to.i \neq j$ and $points\_to.j = i$. If the first i,j-move only involves process i then this must be a *marriage* move resulting in $points\_to.i = j$ and $points\_to.j = i$ and from Lemma 5 neither i nor j will make any future i,j-moves. If the first i,j-move involves process j then this must be an *abandonment* move. If process i does not make a simultaneous i,j-move then this will result in configuration (1) and there can at most be two more steps with i,j-moves for a total of three steps containing i,j-moves. If process i does make a simultaneous i,j-move with process j executing an abandonment move, then this must be a *marriage* move. We are now at a similar configuration as Case *(2)* but with $\neg matched.j$. If the second i,j-move involves process i then this must be an *abandonment* move implying that $matched.j$ has changed to true. It then follows from Corollary 2 that process j (and therefore also process i) will not make any future i,j-move leaving a total of two steps containing i,j-moves. If the second i,j-move does not involve i then this must be a *marriage* move performed by process j and resulting in $points\_to.i = j$ and $points\_to.j = i$ and from Lemma 5 neither i nor j will make any future i,j-moves.

4. $points\_to.i = j$ and $points\_to.j = i$. In this case it follows from Lemma 5 that neither process i nor process j will make any future i,j-moves.

□

It should be noted in the proof of Lemma 6 that only an edge (i,j) where we initially have either $points\_to.i = j$ or $points\_to.j = i$ (but not both) can result in three i,j-moves, otherwise the limit is two i,j-moves per edge. When we have three (i,j)-moves across an edge (i,j) we can charge these moves to the process that was initially pointing to the other. In this way each process will at most be incident on one edge which it is charged three moves for. From this observation we can now give the following bound on the total number of steps needed to obtain a stable solution.

Theorem 2 *Algorithm 1 will stabilize after at most* $3 \cdot n + 2 \cdot m$ *steps under the distributed adversarial daemon.*

Proof: From Corollary 2 we know that there can be at most $2n$ *update* moves, each which can occur in a separate step. From Lemma 6 it follows that there can at most be three i,j-moves

per edge. But as observed, there is at most one such edge incident on each process i for which process i is charged for, otherwise the limit is two i, j-moves. Thus the total number of i, j-moves is at most $n + 2 \cdot m$ and the result follows. □

From Theorem 2 it follows that Algorithm 1 will use $O(m)$ moves on any connected system when assuming a distributed daemon. Since the distributed daemon encompasses the sequential daemon this result also holds for the sequential daemon. To see that this is a tight bound for the stabilization time, consider a complete graph in which each process $i_n, i_{n-1}, ..., i_1$ has a unique identifier such that $i_n > i_{n-1} > ... > i_1$. We will now show that there exists an initial configuration and a sequence of moves such that $\Omega(m)$ moves are executed before the system reaches a stable configuration. Consider that if initially every process is unmarried and not pointing to anyone. Then the processes $i_{n-1}, ..., i_1$ will be eligible to execute *seduction* moves and point to i_n. Following this, i_n may now execute a *marriage* move, and become married to i_{n-1}. Thus the processes $i_{n-2}, ..., i_1$ are now eligible to execute *abandonment* moves. Observe that following these moves, two moves have been executed for every edge incident to i_n, and the processes $i_{n-2}, ..., i_1$ are once again not pointing to any other process. Furthermore, by Lemma 5 we know that neither i_n nor i_{n-1} will execute any further non-*update* moves (note that no moves were executed for any edge incident on i_{n-1}, with the exception of the edge (i_n, i_{n-1})). In the same manner, we can now reason that in addition to the above, two moves are executed for every edge incident on i_{n-2} (with the exception of those incident on i_n or i_{n-1}). Repeating this argument gives that $\Omega(m)$ non-*update* moves are executed before the system reaches a stable configuration.

3.2 Generalized diners

Algorithm. An instance of the generalized diners problem defines for each process p a set of *communication neighbors* $N.p$ and a set of *conflict neighbors* $M.p$. Both relations are symmetric. That is, for any two processes p and q if $p \in N.q$ then $q \in N.p$. The same applies to $M.p$. Throughout the computation each process requests critical section (CS for short) access an arbitrary number of times: from zero to infinity. A program that solves the generalized diners satisfies the following two properties for each process p:

safety — if the action that executes the CS is enabled in p, it is disabled in all processes of $M.p$;

liveness — if p wishes to execute the CS, it is eventually allowed to do so.

In this section it is assumed that in any computation, the action execution is *weakly fair*. That is, if an action is enabled in all but finitely many states of an infinite computation then this action is executed infinitely often.

K-hop diners is a restriction of generalized diners. In k-hop diners, for each process p, $M.p$ contains all processes whose distance to p in the graph formed by the communication topology is no more than k.

The main idea of the algorithm is to coordinate CS request notifications between multiple conflict neighbors of the same process. We assume that for each process p there is a tree that spans $M.p$. This tree is rooted in p. A stabilizing breadth-first construction of a spanning tree is a relatively simple task [22].

The processes in this tree propagate CS request of its root. The request reflects from the leaves and informs the root that its conflict neighbors are notified. This mechanism resembles information propagation with feedback [9].

Algorithm 2 $\mathcal{KDP}$: a self-stabilizing k-hops diners algorithm

process p
const
M: k-hop conflict neighbors of p
N: communication neighbors of p
$(\forall q : q \in M : dad.p.q \in N, KIDS.p.q \subset N)$
parent id and set of children ids for each k-hop neighbor
parameter
$r : M$
var
$state.p.p : \{\mathbf{idle}, \mathbf{req}\}$,
$(\forall q : q \in M : state.p.q : \{\mathbf{idle}, \mathbf{req}, \mathbf{rep}\})$,
$YIELD : \{\forall q : q \in M : q > p\}$ lower priority processes to wait for
$needcs$: **boolean**, application variable to request the CS

$*[$

join: $needcs \wedge state.p.p = \mathbf{idle} \wedge YIELD = \varnothing \wedge$
$(\forall q : q \in KIDS.p.p : state.q.p = \mathbf{idle}) \longrightarrow$
$state.p.p := \mathbf{req}$

$[]$

enter: $state.p.p = \mathbf{req} \wedge$
$(\forall q : q \in KIDS.p.p : state.q.p = \mathbf{rep}) \wedge$
$(\forall q : q \in M \wedge q < p : state.p.q = \mathbf{idle}) \longrightarrow$
/* Critical Section */
$YIELD := \{\forall q : q \in M \wedge q > p : state.p.q = \mathbf{rep}\}$,
$state.p.p := \mathbf{idle}$

$[]$

forward: $state.p.r = \mathbf{idle} \wedge state.(dad.p.r).r = \mathbf{req} \wedge$
$((KIDS.p.r = \varnothing) \vee (\forall q : q \in KIDS.p.r : state.q.r = \mathbf{idle})) \longrightarrow$
$state.p.r := \mathbf{req}$

$[]$

back: $state.p.r = \mathbf{req} \wedge state.(dad.p.r).r = \mathbf{req} \wedge$
$((KIDS.p.r = \varnothing) \vee (\forall q : q \in KIDS.p.r : state.q.r = \mathbf{rep})) \vee$
$state.p.r \neq \mathbf{rep} \wedge state.(dad.p.r).r = \mathbf{rep} \longrightarrow$
$state.p.r := \mathbf{rep}$

$[]$

stop: $(state.p.r \neq \mathbf{idle} \vee r \in YIELD) \wedge$
$state.(dad.p.r).r = \mathbf{idle} \longrightarrow$
$YIELD := YIELD \setminus \{r\}$,
$state.p.r := \mathbf{idle}$

$]$

The access to the CS is granted on the basis of the priority of the requesting process. Each process has an identifier that is unique throughout the system. A process with lower identifier has higher priority. To ensure liveness, when executing the CS, each process p records the identifiers of its lower priority conflict neighbors that also request the CS. Before requesting it again, p then waits until all these processes access the CS.

Each process p has access to a number of constants. The set of identifiers of its communication neighbors is N, and its conflict neighbors is M. For each of its conflict neighbors r, p knows the appropriate spanning tree information: the parent identifier — $dad.p.r$, and a set of ids of its children — $KIDS.p.r$.

Process p stores its own request state in variable $state.p.p$ and the state of each of its conflict neighbors in $state.p.r$. Notice that p's own state can be only **idle** or **req**, while for its conflict neighbors p also has **rep**. To simplify the description, depending on the state, we refer to the process as being idle, requesting or replying. In $YIELD$, process p maintains the ids of its lower priority conflict neighbors that should be allowed to enter the CS before p requests it again. Variable *needcs* is an external Boolean variable that indicates if CS access is desired. Notice that CS entry is guaranteed only if *needcs* remains **true** until p requests the CS.

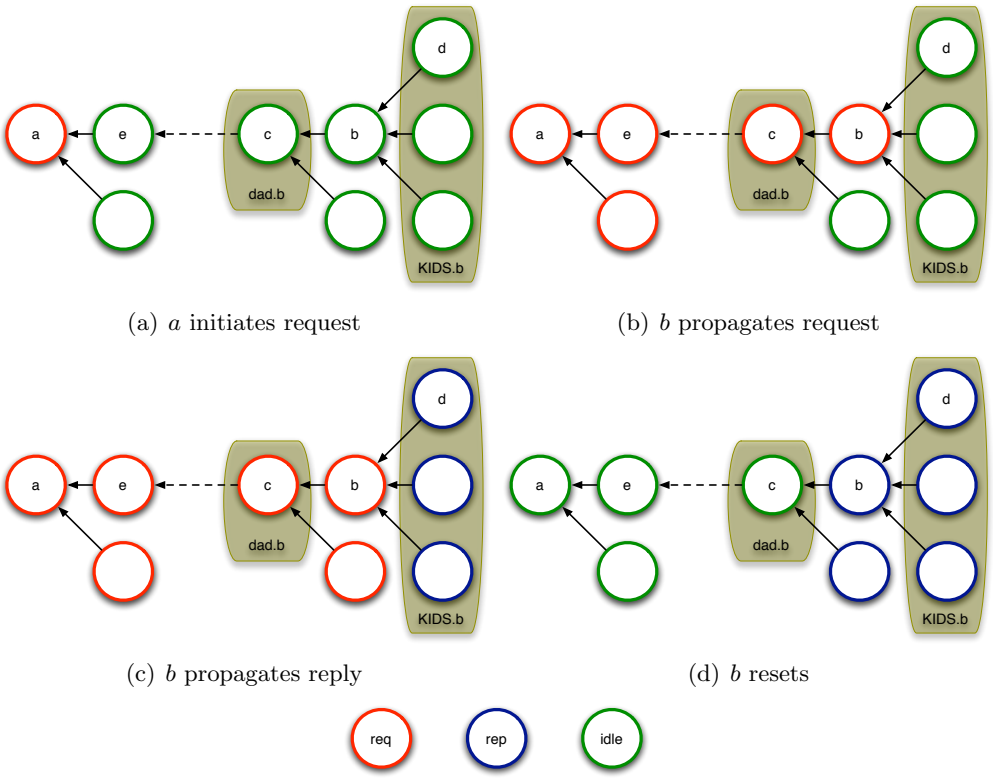
There are five actions in the algorithm. The first two: *join* and *enter* manage CS entry of p itself. The remaining three: *forward*, *back* and *stop* — propagate CS request information along the tree. Notice that the latter three actions are parametrized over the set of p's conflict neighbors.

Action *join* states that p requests the CS when the application variable *needcs* is **true**, p itself, as well as its children in its own spanning tree, is idle and there are no lower priority conflict neighbors to wait for. As action *enter* describes, p enters the CS when its children reply and the higher priority processes do not request the CS themselves. To simplify the presentation, we describe the CS execution as a single action.

Action *forward* describes the propagation of a request of a conflict neighbor r of p along r's tree. Process p propagates the request when p's parent — $dad.p.r$ is requesting and p's children are idle. Similarly, *back* describes the propagation of a reply back to r. Process p propagates the reply either if its parent is requesting and p is the leaf in r's tree or all p's children are replying. The second disjunct of *back* is to expedite the stabilization of Algorithm 2. Action *stop* resets the state of p in r's tree to idle when its parent is idle. This action removes r from the set of lower-priority processes to await before initiating another request.

The operation of Algorithm 2 in legitimate states is illustrated in Figure 5. We focus on the conflict neighborhood $M.a$ of a certain node a. We consider representative nodes in the spanning tree of $M.a$. Specifically, we consider one of a's children — e, a descendant — b, b's parent — c and one of b's children — d.

Initially, the states of all processes in $M.a$ are idle. Then, a executes *join* and sets $state.a.a$ to **req** (see Figure 5, (a)). This request propagates to process b, which executes *forward* and sets $state.b.a$ to **req** as well (Figure 5, (b)). The request reaches the leaves and bounces back as the leaves change their state to **rep**. Process b then executes *back* and changes its state to **rep** as well (Figure 5, (c)). After the reply reaches a and if none of the higher priority processes are requesting the CS, a executes *enter*. This action resets $state.a.a$ to **idle**. This reset propagates to b which executes *stop* and also changes $state.b.a$ to **idle** (Figure 5, (d)).



(a) a initiates request

(b) b propagates request

(c) b propagates reply

(d) b resets

(e) color code

Figure 5: Phases of Algorithm 2 operation

Proof of Correctness. The proof of correctness that is presented in this section demonstrates the use of *attractors* (see Figure 6). An attractor a is a predicate on configurations that is closed for any subsequent execution, and that attracts any execution starting from any arbitrary initial state. In other words, any execution of the system has a configuration that satisfies a. Usually, attractors are nested as presented in Figure 6, meaning that each nested attractor is a refinement of the previous one. Figure 6.(a) presents the notion of nested attractors that do not require a fairness condition: starting from any state, only a finite number of steps may be executed before a particular intermediate attractor is satisfied. In contrast, Figure 6.(b) presents some paths that are cycling, *i.e.* there exists an execution such that the attractor is never reached (*e.g.* the cycle of red nodes with f label). However, for every configuration in a cyclic path, there exists another path that leads to a configuration that satisfies the attractor. Assuming *weak fairness* (*i.e.* in all but finitely many configurations, if it is possible that a particular action makes the configuration satisfy the attractor, then this action is eventually executed), we are able to prove that the attractor is eventually satisfied. Of course, there can be nested such attractors as well.

We present Algorithm 2 correctness proof as follows. We first state a predicate we call $InvK$ and demonstrate that Algorithm 2 stabilizes to it in Theorem 3. We then proceed to show that if $InvK$ holds, then Algorithm 2 satisfies the safety and liveness properties of the k-hop diners in Theorems 4 and 5 respectively.

Throughout this section, unless otherwise specified, we consider the conflict neighbors of a certain node a (see Figure 5). That is, we implicitly assume that a is universally quantified over all processes in the system. We focus on the following nodes: $e \in KIDS.a.a$, $b \in M.a$, $c \equiv dad.b.a$ and $d \in KIDS.b.a$.

Since we discuss the states of e, b, c and d in the spanning tree of a, when it is clear from the context, we omit the specifier of the conflict neighborhood. For example, we use $state.b$ for $state.b.a$. Also, for clarity, we attach the identifier of the process to the actions it contains. For example, *forward.b* is the *forward* action of process b.

Our global predicate consists of the following predicates that constrain the states of each individual process and the states of its communication neighbors. The predicate below relates the states of the root of the tree a to the states of its children.

$$(state.a = \textbf{idle}) \Rightarrow (\forall e : e \in KIDS.a : state.e \neq \textbf{req}) \qquad (Inv.a)$$

The following sequence of predicates relates the state of b to the state of its neighbors.

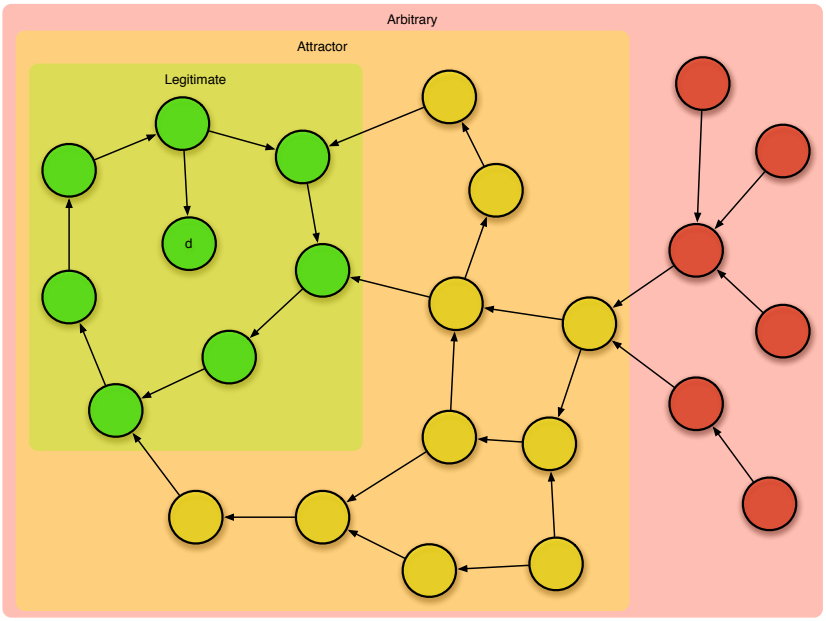
$$state.b = \textbf{idle} \wedge state.c \neq \textbf{rep} \wedge (\forall d : d \in KIDS.b : state.d \neq \textbf{req}) \qquad (I.b.a)$$
$$state.b = \textbf{req} \wedge state.c = \textbf{req} \qquad (R.b.a)$$
$$state.b = \textbf{rep} \wedge (\forall d : d \in KIDS.b : state.d = \textbf{rep}) \qquad (P.b.a)$$

We denote the disjunction of the above three predicates as follows:

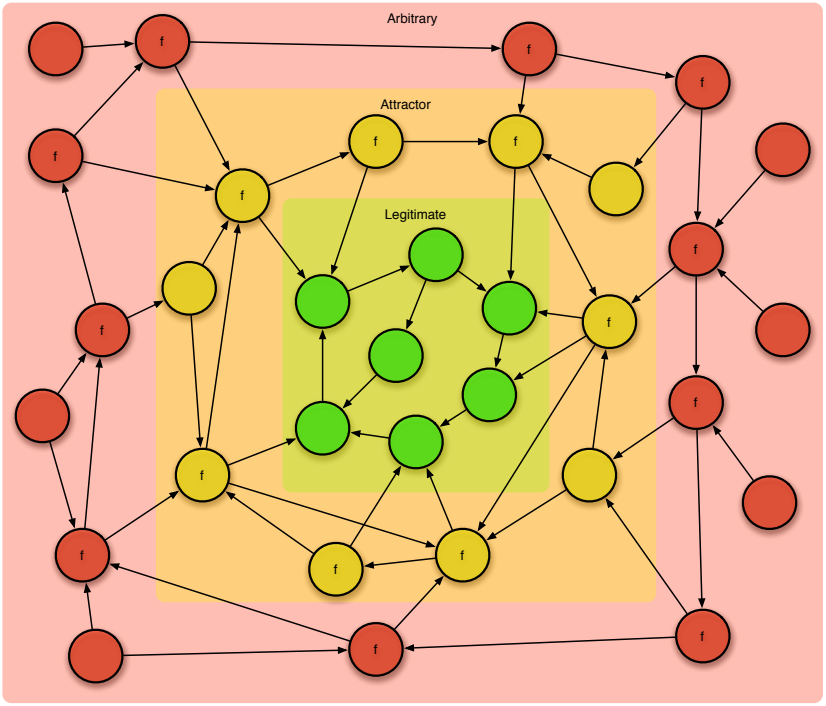
$$I.b.a \vee R.b.a \vee P.b.a \qquad (Inv.b.a)$$

The following predicate relates the states of all processes in $M.a$.

$$(\forall a :: Inv.a \wedge (\forall b : b \in M.a : Inv.b.a)) \qquad (InvK)$$

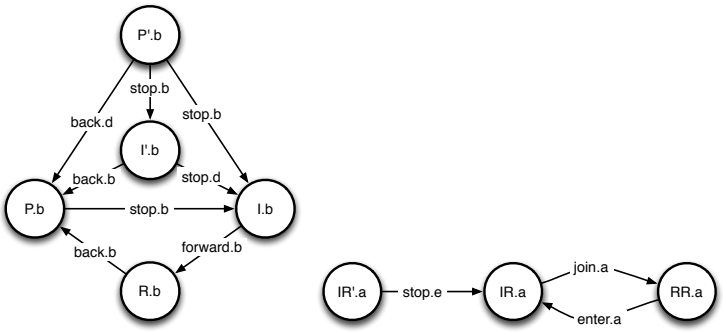


(a) without fairness condition



(b) with fairness condition

Figure 6: Proving Self-stabilization with attractors



(a) intermediate process b if Inv holds for ancestors

(b) root process a

Figure 7: State transitions for an individual process

To aid in exposition, we mapped the states and transitions for individual processes in Figure 7. Note that to simplify the picture, for the intermediate process b we only show the states and transitions if $Inv.f.a$ holds for each ancestor f of b. For b, the $I.b$, $R.b$ and $P.b$ denote the states conforming to the respective predicates, while the primed versions $I'.b$ and $P'.b$ signify the states where b is respectively idle and replying but $Inv.b.a$ does not hold. Notice that if $Inv.c.a$ holds for b's parent c, the primed version of R does not exist. Indeed, to violate R, b should be requesting while c is either idle or replying. However, if $Inv.c.a$ holds and c is in either of these two states, b cannot be requesting.

For a, $IR.a$ and $RR.a$ denote the states where a is respectively idle and requesting while $Inv.a$ holds. In states $IR'.a$, a is idle while $Inv.a$ does not hold. Notice that since $state =$ **req** falsifies the antecedent of $Inv.a$, the predicate always holds if a is requesting. The state transitions in Figure 7 are labeled by actions whose execution effects them. Loopback transitions are not shown.

Theorem 3 *Algorithm 2 stabilizes to $InvK$.*

Proof: By the definition of stabilization, $InvK$ should be closed with respect to the execution of the actions of Algorithm 2, and Algorithm 2 should converge to $InvK$. We prove the closure first.

Closure. To aid in the subsequent convergence proof, we show a property that is stronger than just the closure of $InvK$. We demonstrate the closure of the following conjunction of predicates: $Inv.a$ and $Inv.b.a$ for a set of descendants of a up to a certain depth of the tree. To put another way, in showing the closure of $Inv.b.a$ for b we assume that the appropriate predicates hold for all its ancestors. Naturally, the closure of $InvK$ follows. By definition of predicate closure, we need to demonstrate that if the predicate holds in a certain state, the execution of any action in this state does not violate the predicate.

Let us consider $Inv.a$ and the root process a first. Notice that the only two actions that can potentially violate $Inv.a$ are *enter.a* and *forward.e*. Let us examine each action individually. If *enter.a* is enabled, each child of a is replying. Hence, when it is executed and it changes the state of a to **idle**, $Inv.a$ holds. If *forward.e* is enabled, a is requesting. Thus, executing the action and setting the state of e to **req** does not violate $Inv.a$.

Let us now consider $Inv.b.a$ for an intermediate process $b \in M.a$. We examine the effect of the actions of b, b's parent — c, and one of b's children — d in this sequence.

We start with the actions of b. If $I.b$ holds, *forward.b* is the only action that can be enabled. If it is enabled, c is requesting. Thus, if it is executed, $R.b$ holds and $Inv.b.a$ is not violated. If $R.b$ holds then *back.b* is the only action that can be enabled. However, if *back.b* is enabled and $R.b$ holds, then all children of b are replying. If *back.b* is executed, the resultant state conforms to $P.b$. If $P.b$ holds, then *stop.b* can exclusively be enabled. If $P.b$ holds and *stop.b* is enabled, then c is idle and all children of b are replying. The execution of *back.b* sets the state of b to **idle**. The resulting state conforms to $I.b$ and $Inv.b.a$ is not violated.

Let us examine the actions of c. Recall that we are assuming that $Inv.c$ and the respective invariants of all of b's ancestors hold. If $I.b$ holds, *forward.c* and *join.c* (in case b is a child of a) are the actions that can possibly be enabled. If either is enabled, b is idle. The execution of either action changes the state of c to **req**. $I.b$ and $Inv.b.a$ still hold. If $R.b$ holds, none of the actions of c are enabled. Indeed, actions *forward.c*, *back.c*, *join.c* and *enter.c* are disabled. Moreover, if $R.b$ holds, c is requesting: since $Inv.c$ holds, c must be in $R.c$. Which means that c's parent is not idle. Hence, *stop.c* is also disabled. Since $P.b$ does not mention the state of c, the execution of c's actions does not affect the validity of $P.b$.

Let us now examine the actions of d. If $I.b$ holds, the only possibly enabled action is *stop.d*. The execution of this action changes the state of d to **idle**, which does not violate $I.b$. $R.b$ does not mention the state of d. Hence, its action execution does not affect $R.b$. If $P.b$ holds, all actions of d are disabled. This concludes the closure proof of $InvK$.

Convergence. We prove convergence by induction on the depth of the tree rooted in a. Let us show convergence of a. The only illegitimate set of states is $IR'.a$. When a conforms to $IR'.a$, a is idle and at least one child e is requesting. In such state, all actions of a that affect its state are disabled. Moreover, for every child of a that is idle, all relevant actions are disabled as well. For the child of a that is not idle, the only enabled action is *stop.e*. After this action is executed, e is idle. Thus, eventually $IR.a$ holds.

Let a conform to $Inv.a$. Also, let every descendant process f of a up to depth i conform to $Inv.f.a$. Let the distance from a to b be $i+1$. We shall show that $Inv.b.a$ eventually holds. Notice that according to the preceding closure proof, the conjunction of $Inv.a$ and $Inv.f.a$ for each process f in the distance no more than i is closed.

Note that according to Figure 7, there is no loop in the state transitions containing primed states. Hence, to prove that b eventually satisfies $Inv.b.a$ we need to show that b does not remain in a single primed set of states indefinitely. Process b can satisfy either $I'.b$ or $P'.b$. Let us examine these cases individually.

Let $b \in I'.b$. Since $Inv.c.a$ holds, if b is idle, c cannot satisfy $P.c$. Thus, for b to satisfy $I'.b$, at least one child d of b must be requesting. However, if b is idle then *stop.d* is enabled. Notice that when b is idle, none of its non-requesting children can start to request. Thus, when this *stop* is executed for every requesting child of b, b leaves $I'.b$.

Suppose $b \in P'.b$. This means that there exists at least one child d of b that is not replying. However, for every such process d, *back.d* is enabled. Notice that when b is replying, none of its replying children can change state. Thus, when *back* is executed for every non-replying child of b, b leaves $P'.b$.

Hence, Algorithm 2 converges to $InvK$. □

Theorem 4 *If $InvK$ holds and enter.a is enabled, then for every process $b \in M.a$, enter.b is disabled.*

Proof: If *enter.a* is enabled, every child of a is replying. Due to $InvK$, this means that every descendant of a is also replying. Thus, for every process $x \in M.a$ whose priority is

lower than a's priority, *enter.x* is disabled. Note also, that since *enter.a* is enabled, for every process $y \in M.a$ whose priority is higher than a's, *state.a.y* is **idle**. According to $InvK$, none of the ancestors of a in y's tree, including y's children, are replying. Thus, *enter.y* is disabled. In short, when *enter.a* is enabled, neither higher nor lower priority processes of $M.a$ have *enter* enabled. The theorem follows. □

Lemma 7 *If $InvK$ holds and some process a is requesting, then eventually either a stops requesting or none of its descendants are idle.*

Proof: Notice that the lemma trivially holds if a stops requesting. Thus, we focus on proving the second claim of the lemma. We prove it by induction on the depth of a's tree. Process a is requesting and so it is not idle. By the assumption of the lemma, a will not be idle. Now let us assume that this lemma holds for all its descendants up to distance i. Let b be a descendant of a whose distance from a is $i + 1$. And let b be idle.

By inductive assumption, b's parent c is not idle. Due to $InvK$, if b is idle, c is not replying. Hence, c is requesting. If there exists a child d of b that is not idle, then *stop.d* is enabled at d. When *stop.d* is executed, d is idle. Notice that when b and d are idle, all actions of d are disabled. Thus, d continues to be idle. When all children of b are idle and its parent is requesting, *forward.b* is enabled. When it is executed, b is not idle. Notice, that the only way for b to become idle again is to execute *stop.b*. However, by inductive assumption c is not idle. This means that *stop.b* is disabled. The lemma follows. □

Lemma 8 *If $InvK$ holds and some process a is requesting, then eventually all its children in $M.a$ are replying.*

Proof: Notice that when a is requesting, the conditions of Lemma 7 are satisfied. Thus, eventually, none of the descendants of a are idle. Notice that if a process is replying, it does not start requesting without being idle first (see Figure 7). Thus, we have to prove that each individual process is eventually replying. We prove it by induction on the height of a's tree.

If a leaf node b is requesting and its parent is not idle, *back.b* is enabled. When it is executed, b is replying. Assume that each node whose longest distance to a leaf of a's tree is i is replying. Let b's longest distance to a leaf be $i + 1$. By assumption, all its children are replying. Due to Lemma 7, its parent is not idle. In this case *back.b* is enabled. After it is executed, b is replying. By induction, the lemma holds. □

Lemma 9 *If $InvK$ holds and the computation contains infinitely many states where a is idle, then for every descendant in a's tree there are infinitely many states where it is idle as well.*

Proof: We first consider the case where the computation contains a suffix where a is idle in every state. In this case we prove the lemma by induction on the depth of a's tree with a itself as a base case. Assume that there is a suffix where all descendants of a up to depth i are idle. Let us consider process b whose distance to a is $i + 1$ and this suffix. Notice that this means that c remains idle in every state of this suffix. If b is not idle, *stop.b* is enabled. Once it is executed, no relevant actions are enabled at b and it remains idle afterwards. By induction, the lemma holds.

Let us now consider the case where no computation suffix of continuously idle a exists. Yet, there are infinitely many states where a is idle. Thus, a leaves the idle state and returns to it infinitely often. We prove by induction on the depth of the tree that every descendant

of a behaves similarly. Assume that this claim holds for the descendants up to depth i. Let b's distance to a be $i+1$.

When $InvK$ holds, the only way for b's parent c to leave **idle** is to execute *forward.c* (see Figure 7). Similarly, the only way for c to return to **idle** is to execute *stop.c* while c is replying [1]. However, *forward.c* is enabled only when b is idle. Also, according to $InvK$ when c is requesting, b is not idle. Thus, b leaves **idle** and returns to it infinitely many times as well. By induction, the lemma follows. □

Lemma 10 *If $InvK$ holds and process a is requesting such that and a's priority is the highest among the processes that ever request the CS in $M.a$, then a eventually executes the CS.*

Proof: If a is requesting, then, by Lemma 8, all its children are eventually replying. Therefore, the first and second conjuncts of the guard of *enter.a* are **true**. If a's priority is the highest among all the requesting processes in $M.a$, then each process z, whose priority is higher than that of a is idle. According to Lemma 9, *state.a.z* is eventually **idle**. Thus, the third and last conjunct of *enter.a* is enabled. This allows a to execute the CS. □

Lemma 11 *If $InvK$ holds and process a is requesting, a eventually executes the CS.*

Proof: Notice that by Lemma 8, for every requesting process, the children are eventually replying. According to $InvK$, this implies that all the descendants of the requesting process are also replying. For the remainder of the proof we assume that this condition holds.

We prove this lemma by induction on the priority of the requesting processes. According to Lemma 10, the requesting process with the highest priority eventually executes the CS. Thus, if process a is requesting and there is no higher priority process $b \in M.a$ which is also requesting then, by Lemma 10, a eventually enters the CS.

Suppose, on the contrary, that there exists a requesting process $b \in M.a$ whose priority is higher than a's. If every such process b enters the CS finitely many times, then, by repeated application of Lemma 10, there is a suffix of the computation where all processes with priority higher than a's are idle. Then, by Lemma 10, a enters the CS. Suppose there exists a higher priority process b that enters the CS infinitely often. Since a is requesting, $state.b.a = \textbf{rep}$. When b executes the CS, it enters a into *YIELD.b*. We assume that b enters the CS infinitely often. However, b can request the CS again only if *YIELD.b* is empty. The only action that takes a out of *YIELD.b* is *stop.b*. However, this action is enabled if *state.b.a* is **idle**. Notice that, if $InvK$ holds, the only way for the descendants of a to move from replying to idle is if a itself moves from requesting to idle. That is a executes the CS. Thus, each process a requesting the CS eventually executes it. □

Lemma 12 *If $InvK$ holds and process a wishes to enter the CS, a eventually requests.*

Proof: We show that a wishing to enter the CS eventually executes *join.a*. We assume that a is idle and *needcs.a* is **true**. Then, *join.a* is enabled if $YIELD.a$ is empty. Note that a adds a process to $YIELD$ only when it executes the CS. Thus, as a remains idle, processes can only be removed from $YIELD.a$.

Let us consider a process $b \in YIELD.a$. If b executes the CS finitely many times, then there is a suffix of the computation where b is idle. According to Lemma 9, for all descendants

[1] The argument is slightly different for $c = a$ as it executes *join.a* and *enter.a* instead.

of b, including a, *state.a.b* is idle. If this is the case *stop.a* is enabled. When it is executed b is removed from $YIELD.a$.

Let us consider the case, where b executes the CS infinitely often. In this case, b enters and leaves **idle** infinitely often. According to Lemma 9, *state.a.b* is idle infinitely often. Moreover, a moves to idle by executing *stop.a*, which removes b from $YIELD.a$. The lemma follows. □

The theorem below follows from Lemmas 11 and 12.

Theorem 5 *If InvK holds, a process wishing to enter the CS is eventually allowed to do so.*

We draw the following corollary from Theorems 3, 4 and 5.

Corollary 4 *Algorithm 2 is a self-stabilizing solution to the k-hop diners problem.*

3.3 Census

Algorithm. Each node i has a unique identifier and is aware of its input degree $\delta^-.i$ (the number of its incident arcs), which is also placed in non corruptible memory. A node i arbitrarily numbers its incident arcs using the first $\delta^-.i$ natural numbers. When receiving a message, the node i knows the number of the corresponding incoming link (that varies from 1 to $\delta^-.i$).

Each node maintains a local memory. The local memory of i is represented by a list denoted by $(i_1; i_2; \ldots; i_k)$. Each i_α is a non-empty list of pairs $\langle$*identifier*,*colors*$\rangle$, where *identifier* is a node identifier, and where *colors* is an array of booleans of size $\delta-.i$. Each boolean in the *colors* array is either *true* (denoted by $\bullet$) or *false* (denoted by $\circ$). We assume that natural operations on boolean arrays, such as unary *not*(denoted by $\neg$), binary *and* (denoted by $\wedge$) and binary *or* (denoted by $\vee$) are available.

The goal of the Census algorithm is to guarantee that the local memory of each node contains the list of lists of identifiers (whatever the *colors* value in each pair $\langle$*identifier*,*colors*$\rangle$) that are predecessors of i in the communication graph. Each predecessor of i is present only once in the list. For the Census task to be satisfied, we must ensure that the local memory of each node i can contain as many lists of pairs as necessary. We assume that a minimum of

$$(n-1) \times \left(\log_2(k) + \delta^-.i\right)$$

bits space is available at each node i, where n is the number of nodes in the system and k is the number of possible identifiers in the system.

For example,

$$((j, [\bullet \circ \circ]; q, [\circ \bullet \circ]; t, [\circ \circ \bullet])(z, [\bullet \bullet \bullet]))$$

is a possible local memory for node i, assuming that $\delta^-.i$ equals 3. From the local memory of node i, it is possible to deduce the knowledge that node i has about its ancestors. With the previous example, node j is a direct ancestor of i (it is in the first list of the local memory of i) and this information was carried through incoming channel number 1 (only the first position of the *colors* array relative to node j is *true*). Similarly, nodes q and t are direct ancestors of i and this information was obtained through incoming links 2 and 3, respectively. Then, node z is at distance 2 from i, and this information was received through incoming links numbered 1, 2, and 3.

Each node sends and receives messages. The contents of a message is represented by a list denoted by $(i_1; i_2; \ldots; i_k)$. Each i_α is a non-empty list of *identifiers*.

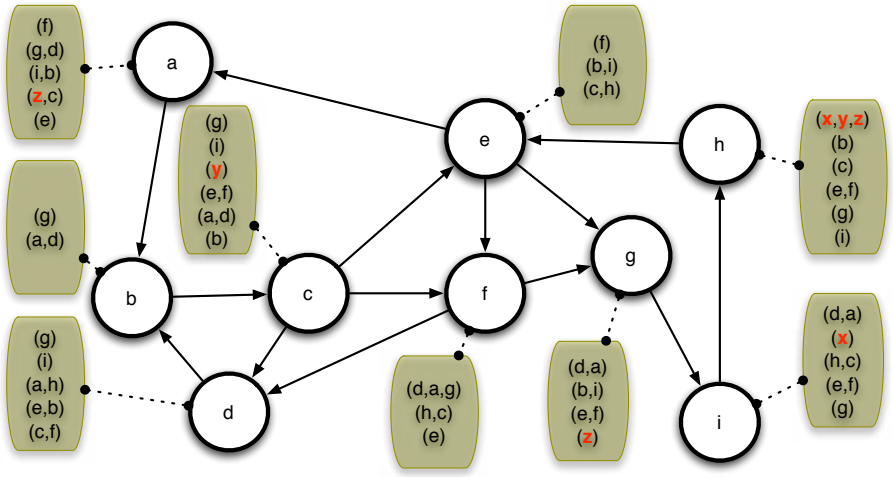
For example,

$$((i)(j;q;t)(z))$$

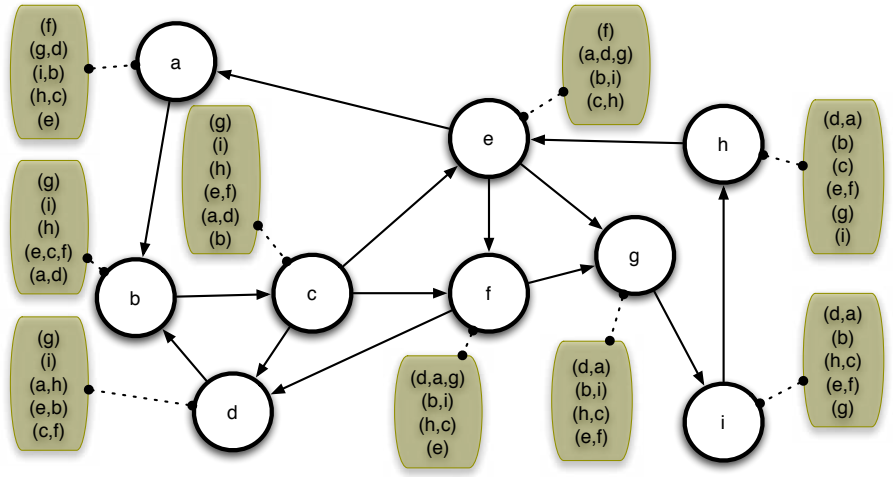
is a possible contents of a message. It means that i sent the message (since it appears first in the message), that i believes that j, q, and t are the direct ancestors of i, and that z is an ancestor at distance 2 of i.

The *distance* from i to j is denoted by $d.i.j$, which is the minimal number of arcs from i to j. We assume that the graph is *strongly connected*, so the distance from i to j is always defined. Yet, since the graph may not be bidirectional, $d.i.j$ may be different from $d.j.i$. The *age* of i, denoted by $\chi.i$, is the greatest distance $d.j.i$ for any j in the graph. The network *diameter* is then equal to

$$\max_i \chi.i = D$$



(a) A possible corrupted initial configuration



(b) A legitimate configuration (messages not displayed)

Figure 8: Two possible configurations for Algorithm 3

Our algorithm can be seen as a knowledge collector on the network. The local memory of

a node then represents the current knowledge of this node about the whole network. The only certain knowledge a node may have about the network is local: its identifier, its incoming degree, the respective numbers of its incoming channels. This is the only information that is stored in non-corruptible memory. In a nutshell, Figure 8 provides a possible corrupted initial configuration (in subfigure (a), where some information is missing and some information is incorrect, *e.g.* nodes x, y, and z are non existent) and a legitimate configuration (in subfigure (b), where the link contents are not displayed).

The algorithm for each node consists in updating in a coherent way its knowledge upon receipt of other process' messages, and communicating this knowledge to other processes after adding its constant information about the network. More precisely, each information placed in a local memory is related to the local name of the incoming channel that transmitted this information. For example, node i would only emit messages starting with singleton list (i) and then not containing i since it is trivially an ancestor of i at distance 0. Coherent update consists in three kinds of actions: the first two being trivial coherence checks on messages and local memory, respectively.

Check Message Coherence Since all nodes have the same behavior, when a node receives a message that does not start with a singleton list, the message is trivially considered as erroneous and is ignored. For example, messages of the form $(\underline{(j;q;t)}(k)(m;y)(p;z))$ are ignored.

Check Local Coherence Regularly and at each message receiving, a node checks for local coherence. We only check here for trivial inconsistencies (see the *problem*() helper function below): a node is incoherent if there exist at least one pair $\langle \mathit{identifier}, \mathit{colors} \rangle$ such that $\mathit{colors} = [\circ \cdots \circ]$ (which means that some information in the local memory was not obtained from any of the input channels). If a problem is detected upon time-out, then the local memory is reinitialized, else if a problem is detected upon a message receipt, the local memory is completely replaced by the information contained in the message.

Trust Most Recent Information When a node receives a message through an incoming channel, this message is expected to contain more recent and thus more reliable information. The node removes all previous information obtained through this channel from its local memory. Then it integrates new information and only keeps old information (from its other incoming channels) that does not clash with new information.

For example, assume that a message $\mathit{mess} = ((j)(k;\, l)(m)(p;\, q\,;r;\, i))$ is received by node i through its incoming link 1 and that $\delta_i^- = 2$. The following informations can be deduced:

1. j is a direct ancestor of i (it appears first in the message),
2. k and l are ancestors at distance 2 of i and may transmit messages through node j,
3. m is an ancestor at distance 3 of i,
4. p, q and r are ancestors at distance 4 of i, j obtained this information through m.

These informations are compatible with a local memory of i such as:

$$((j, [\bullet\circ]; q, [\circ\bullet])(\, k, [\bullet\circ];\, e, [\circ\bullet];\, w, [\circ\bullet])(m, [\circ\bullet]; y, [\bullet\bullet])(p, [\bullet\circ]; z, [\circ\bullet]; h, [\bullet\circ]))$$

Upon receipt of message *mess* at i, the following operations take place: *(i)* the local memory of i is cleared from previous information coming from link 1, *(ii)* the incoming message is "colored" by the number of the link (here each identifier α in the message becomes a pair $\alpha, [\bullet\circ]$ since it is received by link number 1 and *not* by link number 2), and *(iii)* the local memory is enriched as in the following (where "$\leftarrow$" denotes information that was acquired upon receipt of a message, and where "$\rightarrow$" denotes information that is to be forwarded to the node output links):

$$\begin{array}{lcccccc}
(& (q,[\circ\bullet]) & (e,[\circ\bullet];w,[\circ\bullet]) & (m,[\circ\bullet];y,[\circ\bullet]) & (z,[\circ\bullet]) &) \\
\leftarrow (& (j,[\bullet\circ]) & (k,[\bullet\circ];l,[\bullet\circ]) & (m,[\bullet\circ]) & \begin{pmatrix} p,[\bullet\circ];q,[\bullet\circ]; \\ r,[\bullet\circ];i,[\bullet\circ] \end{pmatrix} &) \\
\rightarrow (& (j,[\bullet\circ];q,[\circ\bullet]) & \begin{pmatrix} k,[\bullet\circ];e,[\circ\bullet]; \\ w,[\circ\bullet];l,[\bullet\circ] \end{pmatrix} & (m,[\bullet\bullet];y,[\circ\bullet]) & \begin{pmatrix} p,[\bullet\circ];z,[\circ\bullet]; \\ q,[\bullet\circ];r,[\bullet\circ] \end{pmatrix} &)
\end{array}$$

This message enabled the modification of the local memory of node i in the following way: l is a new ancestor at distance 2. This was acquired through incoming link number 1 (thus through node j). Nodes m and y are confirmed to be ancestors at distance 3, but *mess* sends information *via* nodes j and q, while y only transmits its informations *via* node q. Moreover, q and r are part of the new knowledge of ancestors at distance 4. Finally, although i had information about h $(h,[\bullet\circ])$ before receiving *mess*, it knows now that the information about h is obsolete.

The property of resilience to intermittent link failures of our algorithm is mainly due to the fact that each message is self-contained and independently moves towards a complete correct knowledge about the network. More specifically:

1. The fair loss of messages is compensated by the infinite spontaneous retransmission by each process of their current knowledge.

2. The finite duplication tolerance is due to the fact that our algorithm is *idempotent* in the following sense: if a process receives the same message twice from the same incoming link, the second message does not modify the knowledge of the node.

3. The desequencing can be considered as a change in the relative speeds of two messages towards a complete knowledge about the network. Each message independently gets more accurate and complete, so that their relative order is insignificant. A formal treatment of this last and most important part can be found in the later part of this section..

We now describe helper functions that will enhance readability of our algorithm. Those functions operate on lists, integers and pairs $\langle identifier, colors \rangle$. The specifications of those functions use the following notations: l denotes a list of identifiers, p denotes an integer, lc denotes a list of pair $\langle identifier, colors \rangle$, Ll denotes a list of lists of identifiers, and Llc denotes a list of lists of pairs $\langle identifier, colors \rangle$.

We assume that classical operations on generic lists are available: $\setminus$ denotes the binary operator "minus" (and returns the first list from which the elements of the second have been removed), $\cup$ denotes the binary operator "union" (and returns the list without duplicates of elements of both lists), $+$ denotes the binary operator "concatenate" (and returns the list

resulting from concatenation of both lists), $\sharp$ denotes the unary operator that returns the number of elements contained in the list, and [] takes an integer parameter p so that $l[p]$ returns a reference to the p^{th} element of the list l if $p \leq \sharp l$ (in order that it can be used on the left part of an assignment operator ":="), or expand l with $p - \sharp l$ empty lists and returns a reference to the p^{th} element of the updated list if $p > \sharp l$.

colors$(p) \rightarrow$ **array of booleans** returns the array of booleans that correspond to the p^{th} incoming link, *i.e.* the array that is such that $[\underbrace{\circ \cdots \circ}_{p-1\text{times}} \bullet \underbrace{\circ \cdots \circ}_{\delta^-.i-p\text{times}}]$.

clean$(lc, p) \rightarrow$ **list of couples** returns the empty list if lc is empty and a list of pairs lc_2 such that for each $\langle identifier_{lc}, colors_{lc} \rangle \in lc$, if $colors_{lc} \wedge \neg colors(p) \neq [\circ \cdots \circ]$, then $\langle identifier_{lc}, colors_{lc} \wedge \neg colors(p) \rangle$ is in lc_2, else $\langle identifier_{lc}, * \rangle$ is not in lc_2.

emit(i, Llc) sends the message resulting from $(i)+identifiers(Llc)$ on every outgoing link of i.

identifiers$(Llc) \rightarrow$**list of list of identifiers** returns the empty list if Llc is empty and returns a list Ll of list of identifiers (such that each pair $\langle identifier, colors \rangle$ in Llc becomes $identifier$ in Ll) otherwise.

merge$(lc, l, p) \rightarrow$ **list of couples** returns the empty list if lc and l are both empty and

$$\bigcup_{\substack{\langle i,c \rangle \in lc \\ i \in l}} (\langle i, c \vee colors(p) \rangle) \cup \bigcup_{\substack{\langle i,* \rangle \notin lc \\ i \in l}} (\langle i, colors(p) \rangle)$$

otherwise.

new$(lc, l) \rightarrow$ **list of couples** returns the empty list if lc is empty and the list of pairs $\langle identifier, colors \rangle$ whose $identifier$ is in lc but not in l otherwise.

problem$(Llc) \rightarrow$**boolean** returns *true* if there exist two integers p and q such that $p \leq \sharp(Llc)$ and $q \leq \sharp(Llc[p])$ and $Llc[p][q]$ is of the form $\langle identifier, colors \rangle$ and all booleans in $colors$ are *false* ($\circ$). Otherwise, this function returns *false*.

In addition to its local memory, each node makes use of the following local variables when processing messages: α is the current index in the local memory and message main list, *i_pertinent* is a boolean that is *true* if the α^{th} element of the local memory main list contains pertinent information, *m_pertinent* is a boolean that is *true* if the α^{th} element of the message main list contains pertinent information, *known* is the list of all identifiers found in the local memory and message found up to index α, *temp* is a temporary list that stores the updated α^{th} element of the local memory main list.

We are now ready to present our Census Algorithm (noted Algorithm 3 in the remaining of the paper). This algorithm is message driven: processes execute their code when they receive an incoming message. In order to perform correctly in configurations where no messages are present, Algorithm Algorithm 3 also uses a spontaneous action that will emit a message.

Algorithm 3 $\mathcal{CA}$: a self-stabilizing census algorithm

process i
const
  $\delta^-.i$: input degree of i
parameter
  message: list of lists of identifiers
var
  local_memory.i: list of lists of pairs
lvars
  α: integer
  i_pertinent, *m_pertinent*: boolean
  known, *temp*: list of identifiers

$*[$

problem: $problem(local\_memory.i) \longrightarrow$
  $local\_memory.i := ()$
  $emit(i, local\_memory.i)$

$[]$

update: **receive**$(message)$ **from link** $p \longrightarrow$
  $i\_pertinent := \neg\ problem(local\_memory.i)$
  $m\_pertinent := (\sharp(message[1]) = 1)$
  $\alpha := 0; known := (i);$
  do $m\_pertinent \vee i\_pertinent \longrightarrow$
    $\alpha := \alpha + 1; temp := ()$
    $local\_memory.i[\alpha] := clean(local\_memory.i[\alpha], p)$
    if $i\_pertinent \longrightarrow$
      $temp := new(local\_memory.i[\alpha], known)$
      $temp = () \longrightarrow i\_pertinent := false$
    fi
    if $m\_pertinent \longrightarrow$
      if $message[\alpha] \backslash known = () \longrightarrow$
        $m\_pertinent := false$
      $[]$ $temp := merge(temp, message[\alpha] \backslash known, p)$
      fi
    fi
    if $temp \neq () \longrightarrow$
      $local\_memory.i[\alpha] := temp$
      $known := known \cup identifiers(temp)$
    fi
  od
  $local\_memory.i := (local\_memory.i[1], \ldots, local\_memory.i[\alpha])$

$[]$

resend: $true \longrightarrow$
  $emit(i, local\_memory.i)$

$]$

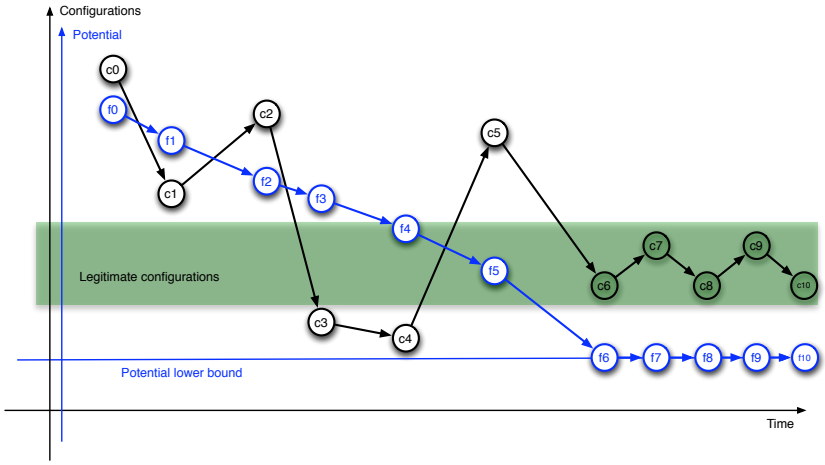


Figure 9: Proving self-stabilization with potential functions

Proof of Correctness. The intuitive reason for self-stabilization of the protocol is as follows: anytime a node sends a message, it adds up its identifier and pushes further in the list of lists included in every message potential fake identifiers. As the network is strongly connected, the message eventually goes through every possible cycle in the network, and every node on every such cycle removes its identifier from a list at a certain index (corresponding to the length of the cycle). So, after a message has visited all nodes, there exists at least one empty list in the list of lists of the message, and all fake identifiers are all included beyond this empty list. Since every outgoing message is truncated from the empty list onwards, no fake identifier remains in the system forever.

The proof of correctness that is presented in this section demonstrates the use of *potential functions* (see Figure 9, where each configuration $c_1, c_2, |dots$ is associated with a number $f_1, f_2, |dots$ obtained through the potential function for this configuration). Intuitively, a potential function maps configurations into a finite set endowed with a total order (typically the set of natural integers), and exhibit the following property: when an action of the distributed system is executed to move from one configuration to the other, the potential function decreases. Since the set is finite and the order well founded, the lowest element is eventually reached. Now if the lowest element of the potential function matches the set of legitimate configuration of the problem specification, this implies stabilization. Of course, the difficult part of the proof lies in finding a suitable such potential function.

In this section, we show that Algorithm 3 is a self-stabilizing Census algorithm. In more details, independently of the initial configuration of network channels (non infinitely full) and of the initial configuration of local memories of nodes, every node ends up with a local memory that reflect the contents of the network, even if unreliable communication media is used for the underlying communication between nodes.

First, we define a formal measure on messages that circulate in the network and on local memories of the nodes. This measure is either the distance between the current form of the message and its canonical form (that denotes optimal knowledge about the network), or between the current value of the local memories and their canonical form (when a node has a perfect knowledge about the network). We use this measure to compute the potential function result of a configuration (in the sequel, the result of this potential function is called

the configuration *weight*).

Then, we show that after a set of emissions and receptions of messages, the weight of a configuration decreases. An induction shows that this phenomenon continue to appear and that the weight of a configuration reaches 0, *i.e.* a configuration where each message is correct and where each node has an optimal knowledge about the network. We also show that such a configuration (whose weight is 0) is stable when a message is emitted or received. According to the previous definitions, a configuration of weight 0 is a *legitimate configuration* after finite time.

These two parts prove respectively the convergence and closure of our algorithm, and establish its self-stabilizing property.

The Census problem being static and deterministic, when we only consider node local memories, there is a single legitimate configuration. This legitimate configuration is when each node has a global correct knowledge about the network. It is also the stable configuration the system would reach had it been started from a zero knowledge configuration (where the local memory of each node is null, and where no messages are in transit in the system).

In this legitimate configuration, all circulating messages are of the same kind. Moreover, on every particular link, all messages have the same contents. The *canonical form of a message* circulating on a link between nodes j and i is the list of lists starting with the singleton list (j) followed by the $\chi.j$ lists of ancestors of j at distance between 1 and $\chi.j$. The *canonical form of node* i's local memory is the list of lists of pairs *Llc* of the $\chi.j$ lists of pairs $\langle$ *identifier*,*colors*$\rangle$ such that: *(i) identifiers*(*Llc*) is the list of the $\chi.i$ lists of ancestors of i at distance 1 to $\chi.i$, and *(ii)* if a shortest path from node j to node i passes through the p^{th} input channel of i, then the boolean array *colors* associated to node j in *Llc* has $colors[p] = \bullet$. For the sake of simplicity, we will also call the α^{th} list of a canonical message or a canonical local memory a *canonical list.*

Proposition 1 *The canonical form of node i's local memory and that of its incoming and outgoing channels are coherent.*

Proof: If node i's local memory is in canonical form, then the *emit* action trivially produces a canonical message.

Conversely, upon receipt by node i of a canonical message through incoming link j, the local memory of i is replaced by a new identical canonical memory. Indeed, *clean* first removes from the α^{th} list of i's local memory all pairs $\langle identifier, colors\rangle$ such that *colors*= *colors*(p), yet by definition of canonical memory, each such *identifier* is that of a node such that the shortest path from *identifier* to i is of length α and passes through j. Moreover, the list l used by *merge* is the list of nodes at distance $\alpha - 1$ of node i, so for any *identifier* appearing in l, two cases may occur:

1. There exists a path from *identifier* to i that is of length $< \alpha$, then *identifiers*$\in$ *known* and it does not appear in the new list of length α,

2. There exists a shorter path from *identifier* to i through j of length α, then $\langle identifier, colors(p)\rangle$ is one of the elements that were removed by *clean* and this information is put back into node i's local memory.

□

Corollary 5 *The set of legitimate configurations is closed.*

Proof: Starting from a configuration where every message and every local memory is canonical, none of the local memories is modified, and none of the emitted message is non-canonical. □

We define a *weight* on configurations as a function on system configurations that returns a positive integer. As configurations of weight zero are legitimate, the weight of a configuration c denotes the "distance" from c towards a legitimate configuration.

In order to evaluate the weight of configurations, we define a measure on messages and local memory of nodes as an integer written using $D+2$ digits in base 3 (where D denotes the graph diameter). The weight of a configuration is then the pair of the maximum weight of local memories, and the maximum weight of circulating messages. For sake of clarity, a single integer will denote the weight of the configuration when both values are equal. Let m be a circulating message on a communication link whose canonical message is denoted by $\tilde{m}$. Note that since a canonical message is of size $\leq D+1$, we have $\tilde{m}[D+2] = ()$. The weight of m is the integer written using $D+2$ base 3 digits and whose α^{th} digit is: *(i)* 0, if $m[\alpha] = \tilde{m}[\alpha]$, *(ii)* 1, if $m[\alpha] \subsetneq \tilde{m}[\alpha]$, and *(iii)* 2, if $m[\alpha] \not\subseteq \tilde{m}[\alpha]$. Then, $3^{D+2}-1$ is the biggest weight for a message, and corresponds to a message that is totally erroneous. At the opposite, 0 is the smallest weight for a message, and corresponds to a canonical message, or to a message that begins with a canonical message.

Let m be the local memory of a node i whose canonical local memory is $\tilde{m}$. The weight of m is the integer written using $D+1$ digits (in base 3) and whose α^{th} digit is: *(i)* 0, if $m[\alpha] = \tilde{m}[\alpha]$, *(ii)* 1, if $m[\alpha] \neq \tilde{m}[\alpha]$ and $identifiers(m[\alpha]) \subseteq identifiers(\tilde{m}[\alpha])$ and for any $\langle$ *identifier*, $colors_1$ $\rangle$ of $m[\alpha]$, the associated $\langle$ *identifier*,$colors_2\rangle$ in $\tilde{m}[\alpha]$ satisfies: $(colors_1 \wedge colors_2) = colors_1$, and *(iii)* 2, otherwise. Then $3^{D+1}-1$ is the biggest weight of a local memory and denotes a totally erroneous local memory. At the opposite, 0 is the smallest weight and denotes a canonical local memory.

Let us notice that in both cases (weight of circulating messages and of nodes local memories), the α^{th} digit 0 associated to the α^{th} list denotes that this particular list is in its final form (the canonical form). The α^{th} digit 1 means that the α^{th} list is coherent with the α^{th} canonical list, but still lacks some information. On the contrary, the α^{th} digit 2 signals that the related α^{th} position contains informations that shall not persist and that are thus unreliable. The weight of a message indicates how much of the information it contains is pertinent. After defining message weight and, by extension, configuration weights, we first prove that starting from an arbitrary initial configuration, only messages of weight lower or equal to $3^{D+1}-1$ are emitted, which stands for the base case for our induction proof.

Lemma 13 *In any configuration, only messages of weight lower than* 3^{D+1} *may be emitted.*

Proof: Any message that is emitted from a node i on a link from i to j is by function *emit*. This function ensures that this message starts with the singleton list (i). This singleton list is also the first element of the canonical message for this channel. Consequently, the biggest number that may be associated to a message emitted by node i starts with a 0 and is followed by $D+1$ digits equal to 2. Its overall weight is at most $3^{D+1}-1$. □

Lemma 14 *Assume* $\alpha \geq 1$. *The set of configurations whose weight is strictly lower than* $3^{\alpha-1}$ *is an attractor for the set of configuration whose weight is strictly lower than* 3^{α}.

Proof: A local memory of weight strictly lower that 3^α contains at most α erroneous lists, and it is granted that it starts with $D+2-\alpha$ canonical lists.

By definition of the *emit* function, each node i that owns a local memory of weight strictly below 3^α shall emit the singleton list (i) followed by $D+2-\alpha$ canonical lists. Since canonical messages sent by a node and its canonical local memory are coherent, it must emit messages that contain at least $D+2-\alpha+1$ canonical lists, which means at worst $\alpha-1$ erroneous lists. The weight of any message emitted in such a configuration is then strictly lower than $3^{\alpha-1}$.

It follows that messages of weight exactly 3^α which remain are those from the initially considered configuration. Hence they are in finite number. Such messages are either lost or received by some node in a finite time. The first configuration that immediately follows the receiving or loss of those initial messages is of weight (3^α (local memory), $3^{\alpha-1}$ (messages)).

The receiving by each node of at least one message from any incoming channel occurs in finite time. By the time each node receives a message, and according to the local memory maintenance algorithm, each node would have been updated. Indeed, the receiving of a message from an input channel implies the cleaning of all previous information obtained from this channel. Consequently, in the considered configuration, all lists in the local memory result from corresponding lists in the latest messages sent through each channel. Yet, all these latest messages have a weight strictly lower than $3^{\alpha-1}$ and by the coherence property on canonical forms, they present information that are compatible with the node canonical local memory, up to index $D+3-\alpha$. By the same property, and since all input channels contribute to this information, it is complete. In the new configuration, each node i maintains a local memory whose first $D+3-\alpha$ lists are canonical, and thus the weight of its local memory is $3^{\alpha-1}$. Such a configuration is reached within finite time and its weight is ($3^{\alpha-1}$ (local memory), $3^{\alpha-1}$ (messages)). □

Proposition 2 *The set of configurations whose weight is* 0 *is an attractor for the set of all possible configurations.*

Proof: By induction on the maximum degree of the weight on configurations. The base case is proved by Lemma 13, and the induction step is proved by Lemma 14. Starting from any initial configuration whose weight is greater that 1, a configuration whose weight is strictly inferior is eventually reached. Since the weight of a configuration is positive or zero, and that the order defined on configurations weights is total, eventually a configuration whose weight is zero is eventually reached. By definition, this configuration is legitimate. □

Theorem 6 *Algorithm 3 is self-stabilizing.*

Proof: Consider a message m of weight 0. Two cases may occur: *(i)* m is canonical, or *(ii)* m starts with a canonical message, followed by at least one empty list, (possibly) followed by several erroneous lists. Assume that m is *not* canonical, then it is impossible that m was emitted, since the **truncate** part of Algorithm 3 ensures that no message having an empty list can be emitted; then m is an erroneous message that was present in the initial configuration.

Similarly, the only local memories that may contain an empty list are those initially present (*e.g.* due to a transient failure).

As a consequence, after receipt of a message by each node and after receipt of all initial messages, all configurations of weight 0 are legitimate (they only contain canonical messages and canonical local memories).

By Proposition 2, the set of legitimate configurations is an attractor for the set of all possible configurations, and Corollary 5 proves closure of the set of legitimate configurations. Therefore, Algorithm 3 is self-stabilizing. □

In the convergence part of the proof, we only assumed that computations were maximal, and that message loss, duplication and desequencing could occur. In order to provide an upper bound on the stabilization time for our algorithm, we assume strong synchrony between nodes and a reliable communication medium between nodes. Note that these assumptions are used for complexity results only, since our algorithm was proven correct even in the case of asynchronous unfair computations with link intermittent failures. In the following D denotes the network diameter.

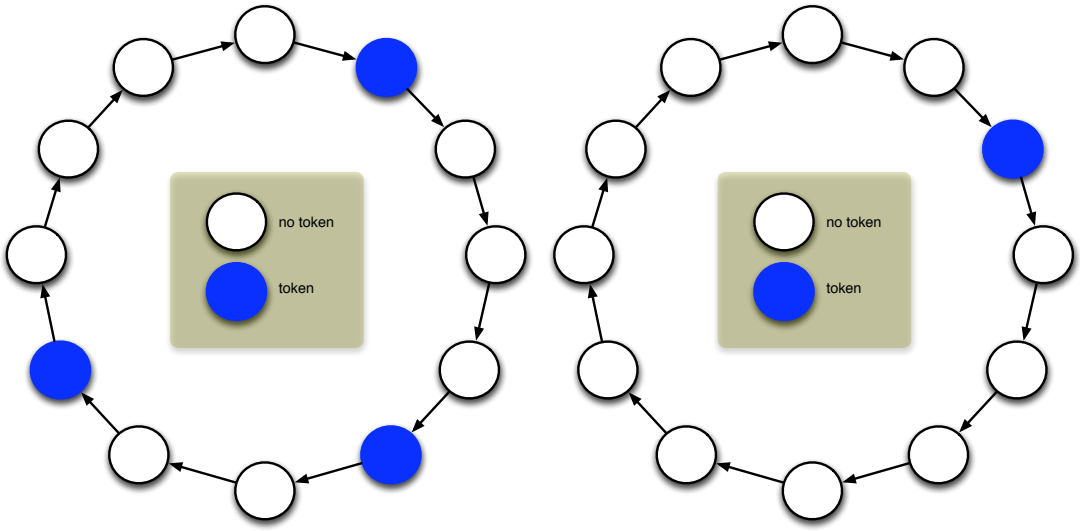
Lemma 15 *Assuming a synchronous reliable system $\mathcal{S}$, the stabilization time of Algorithm 3 is $O(D)$.*

Proof: Since the network is synchronous, we consider *system steps* as: *(i)* each node receives all messages that are located at each of its incoming links and updates its local memory according to the received information, and *(ii)* each node sends as many messages as received on each of its outgoing links. Intuitively, within one system step, each message is received by one process and sent back. Within one system step, all messages are received, and messages of weight strictly inferior to that of the previous step are emitted (see the proof of Lemma 14). In the same time, when a process has received messages from each of its incoming links, its weights is bounded by $3^{D+1-\alpha}$, where D is the network diameter, and α is the number of the system step (see the proof of Lemma 14). Since the maximal initial weight of a message and of a local memory is 3^{D+2}, after $O(D)$ system steps, the weight of each message and of each local memory is 0, and the system has stabilized. □

3.4 Token Passing

Algorithm. The "benchmark" self-stabilizing algorithm for dynamic problems (and the first published algorithm [21]) is the mutual exclusion algorithm on a unidirectional ring. Mutual exclusion is ensured by circulating a "token" (a local predicate at a given node) fairly in the system. In a self-stabilizing context, it is necessary to recover both from configuration where there is no initial token and where there exists several superfluous ones (see *e.g.* Figure 3.4.(a) for a possible inital configuration with several tokens, and Figure 3.4.(b) for a legitimate configuration with a single token). In practice, several criteria need to be taken into account: the stabilization time, the service time (maximum time between two token passings on a given node), the memory used (in bits) and the transparency with respect to underlying communication algorithms.

Due to impossibility results in uniform networks (unidirectional rings where nodes can not be distinguished from one another), probabilistic self-stabilization was introduced [41]. A probabilistic self-stabilizing algorithm has the property of reaching a legitimate configuration in finite time with probability 1. The algorithm of [41] can intuitively be viewed as follows: each process i maintains a local predicate that states whether it has a token, and executes the following code: at each pulse, if i is currently holding a token (*i.e.* the token predicate is true), it transmits it to its successor with probability p, and keeps it with probability $1 - p$. The original paper [41] performs in synchronous odd-sized networks and uses a combinatorial trick to guarantee the presence of at least one token (this trick was further refined to handle the case of arbitrary sized networks [6] and arbitrary minimal number of tokens [39]).



(a) A possible arbitrary initial configuration (b) A possible legitimate configuration

Figure 10: possible configurations of a self-stabilizing token passing algorithm

While the original version of the algorithm is written for the shared memory model, we present a version of the algorithm that is written for the message passing model. So, we have to rely on timeouts to recover from configurations with no tokens. We consider that the protocol has a parameter k that is used as a timeout value; k should be big enough that the timeout is not triggered if a token is already present, but given the probabilistic nature of the token propagation, this is guaranteed only with high probability. Our protocol is presented as Algorithm 3.4.

Proof of correctness. To show stabilization, it is sufficient to prove that starting from a configuration with several tokens, every execution ends up in a configuration with exactly one token. The proof is by showing that in any configuration with two tokens (or more) there is a positive probability that the two tokens merge (the "last" one always get a "transmit" to the successor, while the "first" one always get a "keep" at the current node). When two tokens merge, they stay so. Overall, a configuration with m tokens is always reacheable from a configuration with $m+1$ tokens with a strictly positive probability, while a configuration with $m+1$ tokens is *never* reachable from a configuration with m tokens. As a result, a configuration with exactly one token is eventually reached.

The intuitive proof argument being settled, we focus here on the complexity of the stabilization time. Since the scheduler is synchronous, there is no non-deterministic choice other than probabilistic in every execution, and Markov chains are an attractive tool to compute expected bounds for the stabilization time. We follow the terminology of [65] about Markov Chains. The classical hypothesis can be used since the network has a synchronous behaviour; for an asynchronous setting, see [30]. Let $P_{n\times n}$ be a stochastic matrix, that is the sum of every line is equal to 1. A *discrete Markov Chain*, denoted by $(X_t)_{t\leq 0}$ on a set of states X is a sequence of random variables $X_0, X_1, \ldots$ with $X_i \in X$ and so that X_{i+1} only depends on X_i and $\mathbb{P}r(X_{i+1} = y | X_i = x) = p_{x,y}$. The matrix P is called the *transition probability matrix*. A node x *leads* to a node y if $\exists j \geq i, \mathbb{P}r(X_j = y | X_i = x) > 0$. A state y is an *absorbing* state if

Algorithm 4 $\mathcal{TP}$: a probabilistically self-stabilizing token passing algorithm

process i
const
  $p.i$: predecessor of i on the ring
  $s.i$: successor of i on the ring
parameter
  k: integer
  p: probability to pass the token
var
  $token.i$: $\{\mathbf{null}, \mathbf{token}\}$
  $timeout.i$: integer
function
  $random(x)$: draws a random number z in [0..1], returns true if $z \leq x$, false otherwise.
macro
  $send\_token \equiv$
    if $token.i = \mathbf{token} \wedge random(p) \longrightarrow$
      send(**token**) **to** $s.i$
      $token.i := \mathbf{null}$
    fi

$*[$
$token$: $timeout.i > 0 \wedge \mathbf{receive}(\mathbf{token})$ **from** $p.i \longrightarrow$
    $token.i := \mathbf{token}$
    $timeout.i := k$
    $send\_token$
$[]$
$elapse$: $timeout.i > 0 \longrightarrow$
    $timeout.i := timeout.i - 1$
    $send\_token$
$[]$
$timeout$: $timeout.i = 0 \longrightarrow$
    $token.i := \mathbf{token}$
$]$

y does not lead to any other state. The *expected hitting time* or *hitting time* $\mathbb{E}_x^y$ is the average number of steps starting from node x to reach node y for the first time. We will make use of the following theorem for Markov chains :

Theorem 7 *The vector of hitting times* $\mathbb{E}^t = (\mathbb{E}_x^t : x \in V)$ *is the minimal non-negative solution of the following system of linear equations :*

$$\begin{cases} \mathbb{E}_t^t = 0 \\ \mathbb{E}_x^t = 1 + \sum_{y \neq t} p_{x,y} \mathbb{E}_y^t \text{ for } x \in V \end{cases}$$

Applying Theorem 7 to a specific Markov chain, we obtain a useful Lemma for the analysis of Algorithm 3.4 :

Lemma 16 *Let* C_d *be a chain of* $d+1$ *states* $0, 1, \ldots, d$ *and* $q \in]0, 1/2]$. *If state* 0 *is absorbing and the transition matrix is of the form :*

$$\begin{cases} p_{i,i-1} = p_{i,i+1} = q \text{ for } 1 \leq i \leq d-1 \\ p_{i,i} = 1 - 2q \text{ for } 1 \leq i \leq d-1 \\ p_{d,d} = 1 - q \end{cases}$$

then the hitting time to state 0 *starting from state* i *is* $\mathbb{E}_i^0 = \frac{i}{2q}(2d - i + 1)$.

Proof: We make a use of Theorem 7 for the computation of $\mathbb{E}_i^0$. We have

$$\begin{cases} \mathbb{E}_1^0 = 1 + (1-2q)\mathbb{E}_1^0 + q\mathbb{E}_2^0 \\ \mathbb{E}_i^0 = 1 + q\mathbb{E}_{i-1}^0 + (1-2q)\mathbb{E}_i^0 + q\mathbb{E}_{i+1}^0 \text{ for } 2 \leq i \leq d-1 \\ \mathbb{E}_d^0 = 1 + (1-q)\mathbb{E}_d^0 + q\mathbb{E}_{d-1}^0 \end{cases}$$

Noting that $\mathbb{E}_i^0 = \sum_{j=1}^{i} \mathbb{E}_j^{j-1}$, we are interested by $\mathbb{E}_j^{j-1}$ for $1 \leq j \leq d$. Therefore, $\mathbb{E}_d^{d-1} = 1 + (1-q)\mathbb{E}_d^{d-1} = 1/q$ and

$$\begin{aligned} \mathbb{E}_j^{j-1} &= 1 + (1-2q)\mathbb{E}_j^{j-1} + q\mathbb{E}_{j+1}^{j-1} \\ &= 1 + (1-2q)\mathbb{E}_j^{j-1} + q(\mathbb{E}_{j+1}^{j} + \mathbb{E}_j^{j-1}) \\ &= 1/q + \mathbb{E}_{j+1}^{j} \\ &= \frac{d-j}{q} \end{aligned}$$

This implies that $\mathbb{E}_i^0 = \sum_{j=1}^{i}(d-j)/q = \frac{1}{q}(di - \frac{i(i-1)}{2})$. □

Theorem 8 *In a unidirectional* n*-sized ring containing an arbitrary number* k *of tokens* ($k \geq 2$)*, the stabilization time of Algorithm 3.4 is* $\frac{n^2}{8p(1-p)} \leq \mathbb{E}(T) \leq \frac{n^2}{2p(1-p)}(\frac{\pi^2}{6} - 1) + \frac{n \log n}{p(1-p)}$. *For constant* p, $\mathbb{E}(T) = \Theta(n^2)$.

Proof: For any $k \geq 2$, the evolution of the ring with exactly k tokens under Algorithm 3.4 can be described by a Markov chain $\mathcal{S}_k$ whose state space is the set of k-tuples of positive integers whose sum is equal to n (these integers represent the distances between successive tokens on the ring), with an additional state $\delta = (0, \ldots, 0)$ to represent transitions to a configuration with fewer than k tokens. To prove the upper bound of the theorem, we will

prove an upper bound on the hitting time of this state δ, independently of the initial state. Consider two successive tokens on the ring. On any given round, each will move forward, independently of the other, with probability p, and stay in place with probability $1-p$. Thus, with probability $p(1-p)$, the distance between them will decrease by 1; with the same probability, it will increase by 1; and, with probability $1-2p(1-p)$, the distance will remain the same. Thus, locally, the distance between consecutive tokens follows the same evolution rule as that of the chain $\mathcal{C}_n$ of Lemma 16.

What follows is a formal proof, using the technique of *couplings* of Markov chains, that the expected time it takes for two tokens among k to collide is no longer than the expected time for $\mathcal{C}_{n/k}$ to reach state 0.

For any state $\mathbf{x} = (x^1, \ldots, x^k)$ of $\mathcal{S}_k$, let $m(\mathbf{x}) = \min_i x^i$ denote the minimum distance between two successive tokens, and let $i(\mathbf{x}) = \min\{j : x^j = m(\mathbf{x})\}$ denote the smallest index where this minimum is realized. Let $(X_t)_{t\geq 0}$ denote a realization of the Markov chain $\mathcal{S}_k$. We define a *coupling* $(X_t, Y_t)_{t\geq 0}$ of the Markov chains $\mathcal{S}_k$ and $\mathcal{C}_d$, where $d = \lfloor n/k \rfloor$ and $q = p(1-p)$, as follows :

- $Y_0 = m(X_0)$;
- $Y_{t+1} = \min\{d, Y_t + (X_{t+1}^{i(X_t)} - X_t^{i(X_t)})\}$

In other words, the evolution of Y_t is determined by selecting two tokens that are separated by the minimum distance in X_t, and making the change in Y_t reflect the change in distance between these two tokens (while capping Y_t at d).

A trivial induction on t shows that $Y_t \geq m(X_t)$ holds for all t, so that (X_t) will reach state δ no later than (Y_t) reaches 0. Thus, *the time for $\mathcal{S}_k$ to reach δ (that is, the time during which the ring has exactly k tokens) is stochastically dominated by the time for $\mathcal{C}_d$ to reach* 0. By Lemma 16, the expectation of this time is no longer than

$$\frac{d(d+1)}{2q} \leq \frac{1}{2q}\left(\frac{n^2}{k^2} + \frac{n}{k}\right)$$

Summing over all values of k from 2 to n, we get, for the expected stabilization time T,

$$\begin{aligned}
\mathbb{E}(T) &\leq \frac{1}{2p(1-p)} \sum_{k=2}^{n} \left[\frac{n^2}{k^2} + \frac{n}{k}\right] \\
&\leq \frac{1}{2p(1-p)} \left(\left(\frac{\pi^2}{6} - 1\right) n^2 + n\ln(n)\right)
\end{aligned}$$

The lower bound comes from the fact that, when $k = 2$, the expected time for $\mathcal{C}_{n/2}$ to reach state 0 from state $n/2$ is at least $\frac{n^2}{8p(1-p)}$. □

Remark 1 *Our upper bound on the expected convergence time is minimal for $p = 1/2$. The precise study of Algorithm 3.4 show that the convergence time hardly depends on the initial number of tokens: for n high enough and $p = 1/2$, $\mathbb{E}(T) > n^2/2$ for two initial tokens at distance $n/2$, and $\mathbb{E}(T) < 1.3n^2$ for n tokens.*

4 Research Issues and Summary

In its core formulation, self-stabilization is a useful paradigm for forward recovery in distributed systems and networks. Because self-stabilization only considers the effect of faults, there is no assumption about the nature or the extent of faults. In practise, when a faulty component is diagnosed in a self-stabilizing network (*e.g.* because it exhibits erratic behavior), it is sufficient to remove this component from the system to recover proper behavior automatically [66], as a self-stabilizing system does not require any kind of initialization.

On the negative side, "eventually" does not give any complexity guarantee on the stabilization time, and in some cases, it is possible that a single hazard triggers a correction wave in the whole network. Also, the fact that a system participant is not able to detect stabilization (arbitrary memory corruption could lead the participants believe that the system is stabilized when it is not) prevents safety-related specifications from having self-stabilizing solutions. Those limitations led to defining new forms of self-stabilization, that are presented in Figure 11 and detailed in the remaining of the section. In Figure 11, each arrow denotes the (transitive) relation "provides weaker guarantees than" between two variants of self-stabilization.

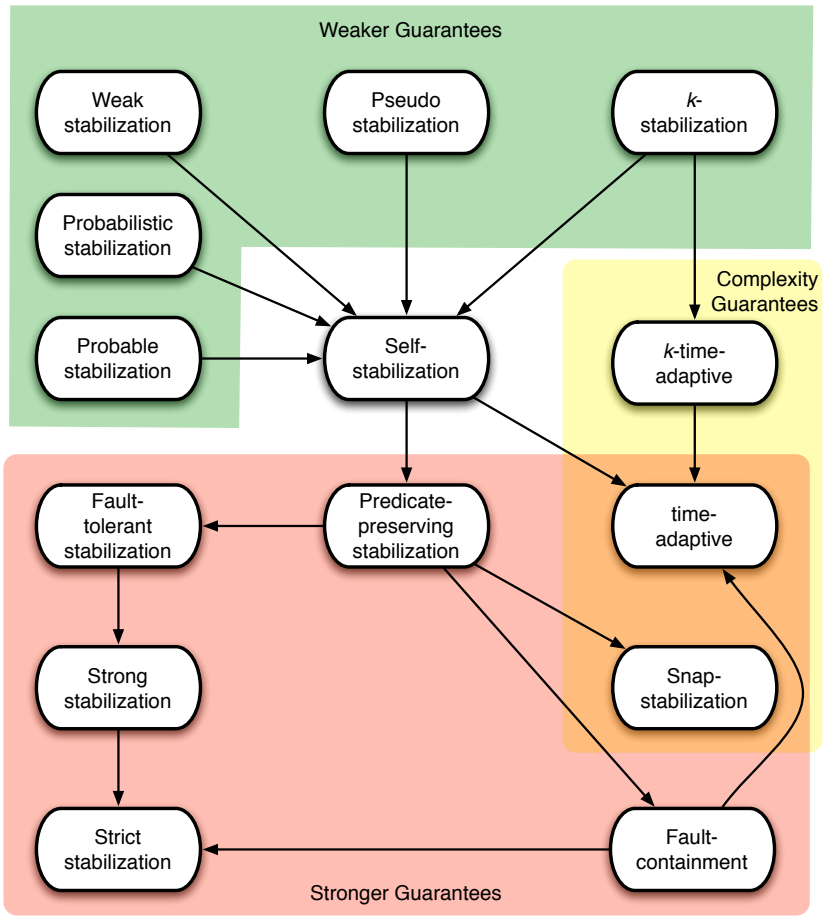


Figure 11: Taxonomy of Self-stabilization main variants

Weaker than self-stabilization. The guarantees that are given are weaker than that of self-stabilization, and this permits in general to solve strictly more difficult specifications than those than can be solved by a self-stabilizing system. Thus problems that are provably intractable in a strict self-stabilizing setting may be come solvable. This permits to widen the scope of self-stabilization to new applications, while maintaining attractive fault tolerance properties to the developped applications.

1. **Restricting the nature of the faults**. This approach, that we denote by *probable stabilization* consists of considering that truly arbitrary memory corruptions are highly unlikely. Probabilistic arguments are used to establish that, in general, memory corruptions that result from faults can be detected using traditional techniques from information theory, such as data redundancy or error detection and correction codes. In particular, in [44], error detection codes are used to determine that memory corruption has occurred, with a high probability. If the article only considers the case where a single corruption arises (in other words, only one node in the system is affected by this corruption), it makes it possible to return to a normal behavior in a single correction step. For a system, even a large-scale one, where memory corruptions are localized in each neighborhood and are not malicious (that is, they can be detected using techniques such a cyclic redundancy checks), this approach is well indicated.

2. **Restricting the extent of the faults**. If we assume that the faults that can occur only ever concern a very small part of the network, it is possible to design algorithms that converge more quickly than traditional self-stabilizing algorithms. In order to have a formal framework, we consider that the distance to a legitimate configuration is equal to the number of nodes whose memories have to be changed in order to achieve a legitimate configuration (as with a Hamming distance). Of course, it is possible that even if we are at a distance k from a legitimate configuration, more than k nodes have, in fact, corrupted memories. From the perspective of returning to a normal state, only the closest legitimate configuration is considered. Studies that attempt to minimize the stabilization time in context where few faults occur usually divide stabilization into two levels [57]:

   (a) *"visible" stabilization*: here, only the output variables of the algorithm are involved. The output variables are typically used by the system's user. For example, if we consider a tree construction algorithm, only the pointer oriented toward the parent node is included in the output variables;

   (b) *"internal" stabilization*: here, all of the algorithm's variables are involved. This type of stabilization corresponds to the traditional concept of self-stabilization;

   In many studies, only the "visible" stabilization is performed quickly (that is, in time relative to the number of faults that strike the system, rather than in time relative to the size of said system), while the "internal" stabilization most of the time remains proportional to the network's size. Algorithms that present this constraint are not capable of tolerating a high frequency of faults. Consider an algorithm whose time for visible stabilization depends on k (the number of faults) and whose internal stabilization depends on n (the size of the system). Now, if a new fault occurs while the visible stabilization is being performed, but not during the internal stabilization, this can lead to a global state containing a number of faults greater than k, and there is no longer

any guarantee on the new time for visible stabilization. The notion of *k-stabilization* [7] is defined as self-stabilization, when restricting the starting configurations to those configurations that are at a distance of k or less from a legitimate configuration. Because of the less hostile environment, it is possible to solve problems that are impossible in the case of general self-stabilization [46, 71], and to offer reduced visible stabilization times. A particular instance of k-stabilization is *node-stabilization* [70]: in the initial configuration, links do not contain any message, yet the memory of the nodes may be arbitrarily corrupted, as this would happen if the networking equipments of a system were shut down but not the computing devices.

3. **Restricting the stabilization guarantees**.

   (a) Section 3.4 already presented the concept of *probabilistic stabilization*. Probabilistic stabilization weakens the convergence requirement by only requiring expected finite time convergence with probability 1. In [13, 14], weak and strong variants of probabilistic stabilization are presented: a *weak* probabilistic algorithm only guarantees correctness with probability 1, while a *strong* probabilistic algorithm guarantees *certain* correctness (that is, when a legitimate configuration is reached, all subsequet executions from this configuration conform to the specification). For example, the algorithm presented in Section 3.4 falls in the category of weak probabilistic algorithms [29].

   (b) The notion of *pseudo-stabilization* [11] removes the guarantee of reaching a *configuration* from which the behavior satisfies the specification. Instead, pseudo-stabilization guarantees that every execution has a suffix that satisfies the specification. The main weakening here is that there is no guarantee that the system ever stabilizes (as it may never reach a legitimate configuration). However, for an external observer, the behavior of the system is eventually correct.

   (c) *Weak stabilization* [36] breaks the requirement that *every* execution reaches a legitimate configuration. Instead, weak stabilization guarantees that from every possible initial configuration c, there *exists* at least one execution starting from c that reaches a legitimate configuration. Recently, a strong connection between weak stabilization and probabilistic stabilization was demonstrated [20]: essentially a weak stabilizing protocol can be turned into a probabilistic stabilizing algorithm that operates under a probabilistic daemon (*i.e.* a distributed scheduler whose choices are probabilistic).

Stronger than self-stabilization. The guarantees that are given are stronger than that of self-stabilization, and this permits in general to solve strictly less difficult specifications than those than can be solved by a self-stabilizing system. That is, the set of problems than can be solved is strictly smaller than the set of problems that can be solved by striclty self-stabilizing algorithms, yet the guarantees are stronger and may even match those of robust algorithms.

1. **Stronger safety guarantees**.

   (a) *Predicate-preserving* stabilization refers to the fact that in addition to being self-stabilizing, the algorithm also preserves some distributed predicate on configurations, either in the stabilizing phase or in the stabilized phase, in spite of the occurrence of new faults (of limited nature). One such instance is the algorithm described

in Section 3.3, where message losses, duplications, and desequencings are tolerated both in the stabilizing and stabilized phases. Another instance is that of *route-preserving* stabilization [49]: the algorithm maintains a shortest path spanning tree in a self-stabilizing way, and has an additional property of path preservation, meaning that if a tree is initially constructed toward a destination, any message transmitted towards that destination reaches it in a finite time, even if the cost of every edge in the system continuously changes. *Super-stabilization* [24, 42, 51] is a special instance of predicate-preserving stabilization. This property states that a super-stabilizing algorithm is self-stabilizing on one hand and, on the other, preserves a predicate (typically a safety predicate) when changes in topology occur in a legitimate configuration. Thus, changes in topology are limited: if these changes occur during the stabilization phase, the system can never stabilize. On the other hand, if they occur only after a correct global state is achieved, the system remains stable.

(b) *Fault-tolerant self-stabilization* is characteristic of algorithms that aim at providing tolerance to both arbitrary transient faults (self-stabilization) and permanent ones (robustness), the two main trends in distributed fault-tolerance (see Section 1). While mostly impossibility results were obtained in this context [3, 8] except for some particular problems such as time synchronization [28], recent results [19] hint that the ultimate properties found in fault-tolerant algorithms are more related to pseudo-stabilization than to self-stabilization.

(c) *Strict stabilization* [64] refers to a different scheme to tolerate both transient and permanent Byzantine faults. The Byzantine contamination radius is defined as the maximum distance from which the effect of Byzantine nodes can be felt. This contamination radius must obviously be as small as possible. A problem is r-restrictive if its specification prohibits combinations of states in a configuration for nodes at a distance of r at the most. For example, the problem of coloring nodes in a network is 1-restrictive, since two neighboring nodes cannot have the same color. On the other hand, the tree construction problem is r-restrictive (for any r between 1 and $n-1$) because the correction implies that all of the parents that are chosen must form a tree. The main theorem in [64] states that if a problem is r-restrictive, the best contamination radius that can be obtained is r. The follow-up paper [63] provides a *Byzantine-insensitive* link coloring algorithm: the subset of correct nodes, once stabilized, can not be influenced again by Byzantine processes.

(d) For problems such as tree construction that have r-restrictive specifications for some arbitrary r, the weaker notion of *Strong stabilization* has been introduced [61]. While strict stabilization contains the action of Byzantine processes in *space*, strong stabilization contains the action of Byzantine processes in *time*: even if a Byzantine process may execute an infinite number of malicious actions in a infinite execution, the correct processes may only be impacted a finite number of times. However, and similarly to pseudo-stabilization, there is no bound on the time after which Byzantine actions are harmless (a Byzantine process may execute only correct actions over a long - unbounded - period of time, and then arbitrarily behave for some time, making correct processes execute a finite (bounded) number of corrective actions).

2. **Stronger complexity guarantees**.

(a) For certain problems, a memory corruption can cause a cascade of corrections in the entire system [5], yet it would be natural for the stabilization to be quicker when the number of failures that strike the system is smaller. This is the principle behind *time-adaptive* self-stabilization [10, 25, 58], also known as scalable stabilization [34] and fault local stabilization [59]. Note that time adaptive protocols have the property of fault containment [33] in the sense that a visible fault can not spread on the while network: it is contained near to its initial location before it disappears. It is possible to arrange self-stabilizing, k-stabilizing and time-adaptive algorithms into classes, depending on the difficulty in solving problems that can be solved in each case. For example, if it is possible to solve a problem in a self-stabilizing way, it is also possible to solve it in a k-stabilizing way (if you can do more, you can do less). Likewise, if it is possible to solve a problem in a time-adaptive way, it is also possible to solve it without trying to constrain the visible stabilization time. Thus, the class of problems that can be solved in a time-adaptive way is a subset of the class of problems that can be solved in a self-stabilizing way, which is itself a subset of the class of problems that have k-stabilizing solutions. These inclusions are strict: some problems can be solved in a k-stabilizing way, but not in a self-stabilizing [71], others can be solved in a self-stabilizing way, but not in a time-adaptive way [31].

(b) Given a problem specification, a *snap-stabilizing* system [9] is guaranteed to perform according to this specification regardless of the initial state. That is, a snap-stabilizing system has a stabilization time of 0. It is important to note that a snap-stabilizing protocol does not guarantee that the system never works in a fuzzy manner. Actually, the main idea behind the snap-stabilization is the following: the protocol is seen as a function and the function ensures two properties despite the arbitrary initial configuration of the system:

   i. Upon an external (*w.r.t.* the protocol) request at a process p, the process p (called the initiator) starts a computation of the function in finite time using special actions called starting actions;

   ii. If the process p starts an computation, then the computation performs an expected task.

   With such properties, the protocol always satisfies its specifications. Indeed, when the protocol receives a request, this means that an external application (or a user) requests the computation of a specific task provided by the protocol. In this case, a snap-stabilizing protocol guarantees that the requested task is executed as expected. On the contrary, when there is no request, there is nothing to guarantee. Due to the "start" and "correctness" properties it has to ensure, snap-stabilization requires specifications based on a sequence of actions ("request", "start",...) rather than a particular subset of configurations (*e.g.*, the legitimate configurations). Most of the literature on snap-stabilization deals with the shared memory model only, with the notable recent exception of [17]. Due to the 0 stabilization time complexity, snap-stabilization actually also guarantees stronger safety properties (*e.g.* mutual exclusion in [17]) than those of self-stabilization.

There remains the special case of k-time adaptive stabilization. In our classification, it is both a weakening of self-stabilization in the sense that fewer faults are allowed, and a

strenghening of self-stabilization in the sense that the stabilization time complexity guarantee is proportional to the number of faults that hit the network.

References

[1] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

[2] Luc Onana Alima, Joffroy Beauquier, Ajoy Kumar Datta, and Sébastien Tixeuil. Self-stabilization with global rooted synchronizers. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 102–109, Amsterdam, The Netherlands, May 1998. IEEE Press.

[3] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In André Schiper, editor, *WDAG*, volume 725 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 1993.

[4] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *STOC '80: Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 82–93, New York, NY, USA, 1980. ACM Press.

[5] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset (extended abstract). In Gerard Tel and Paul M. B. Vitányi, editors, *Distributed Algorithms, 8th International Workshop, WDAG '94*, volume 857 of *Lecture Notes in Computer Science*, pages 326–339. Springer, 1994.

[6] Joffroy Beauquier, Stéphane Cordier, and Sylvie Delaët. Optimum probabilistic self-stabilization on uniform rings. In *Proceedings on the Workshop on Self-stabilizing Systems*, pages 15.1–15.15, 1995.

[7] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. Optimal reactive k-stabilization: The case of mutual exclusion. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, 1999.

[8] Joffroy Beauquier and Synnöve Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *Int. J. Systems Science*, 28(11):1177–1187, 1997.

[9] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1):3–19, 2007.

[10] Janna Burman, Ted Herman, Shay Kutten, and Boaz Patt-Shamir. Asynchronous and fully self-stabilizing time-adaptive majority consensus. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *OPODIS*, volume 3974 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2005.

[11] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[12] Praveen Danturi, Mikhail Nesterenko, and Sébastien Tixeuil. Self-stabilizing philosophers with generic conflicts. In Datta and Gradinariu [15], pages 214–230.

[13] Ajoy K Datta, Maria Gradinariu, and Sébastien Tixeuil. Self-stabilizing mutual exclusion using unfair distributed scheduler. In *IEEE International Parallel and Distributed Processing SYmposium (IPDPS'2000)*, pages 465–470, Cancun, Mexico, May 2000. IEEE Press.

[14] Ajoy K. Datta, Maria Gradinariu, and Sébastien Tixeuil. Self-stabilizing mutual exclusion with arbitrary scheduler. *The Computer Journal*, 47(3):289–298, October 2004.

[15] Ajoy Kumar Datta and Maria Gradinariu, editors. *Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006, Dallas, TX, USA, November 17-19, 2006, Proceedings*, volume 4280 of *Lecture Notes in Computer Science*. Springer, 2006.

[16] Ajoy Kumar Datta and Ted Herman, editors. *Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1-2, 2001, Proceedings*, volume 2194 of *Lecture Notes in Computer Science*. Springer, 2001.

[17] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. In *International Conference on Distributed Systems and Networks (ICDCN 2009)*, number 5404 in LNCS, pages 281–286, January 2009.

[18] Sylvie Delaët and Sébastien Tixeuil. Tolerating transient and intermittent failures. *Journal of Parallel and Distributed Computing (JPDC)*, 62(5):961–981, May 2002.

[19] Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. Robust stabilizing leader election. In Toshimitsu Masuzawa and Sébastien Tixeuil, editors, *SSS*, volume 4838 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2007.

[20] Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, Beijin, China, June 2008.

[21] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[22] S. Dolev. *Self-stabilization*. MIT Press, March 2000.

[23] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999.

[24] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[25] Shlomi Dolev and Ted Herman. Parallel composition for time-to-fault adaptive stabilization. *Distributed Computing*, 20(1):29–38, 2007.

[26] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[27] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997.

[28] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5):780–799, 2004.

[29] Philippe Duchon, Nicolas Hanusse, and Sébastien Tixeuil. Optimal randomized self-stabilizing mutual exclusion in synchronous rings. In *Proceedings of the 18th Symposium on Distributed Computing (DISC 2004)*, number 3274 in Lecture Notes in Computer Science, pages 216–229, Amsterdam, The Nederlands, October 2004. Springer Verlag.

[30] Marie Duflot, Laurent Fribourg, and Claudine Picaronny. Randomized finite-state distributed algorithms as markov chains. In Jennifer L. Welch, editor, *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2001.

[31] Christophe Genolini and Sébastien Tixeuil. A lower bound on k-stabilization in asynchronous systems. In *Proceedings of IEEE 21st Symposium on Reliable Distributed Systems (SRDS'2002)*, Osaka, Japan, October 2002.

[32] Sukumar Ghosh. *Distributed Systems (Computer and Information Sciences)*. Chapman & Hall/CRC, 2006.

[33] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing distributed protocols. *Distributed Computing*, 20(1):53–73, 2007.

[34] Sukumar Ghosh and Xin He. Scalable self-stabilization. *J. Parallel Distrib. Comput.*, 62(5):945–960, 2002.

[35] Mohamed G. Gouda. *Elements of network protocol design.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[36] Mohamed G. Gouda. The theory of weak stabilization. In Datta and Herman [16], pages 114–123.

[37] Mohamed G. Gouda and F. Furman Haddix. The alternator. *Distributed Computing*, 20(1):21–28, 2007.

[38] Maria Gradinariu and Sébastien Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *International Conference on Principles of Distributed Systems (OPODIS'2000)*, pages 55–70, Paris, France, December 2000.

[39] Maria Gradinariu and Sébastien Tixeuil. Tight space uniform self-stabilizing l-mutual exclusion. In *IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, pages 83–90, Phoenix, Arizona, May 2001. IEEE Press.

[40] Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS 2007)*, page 46. IEEE, June 2007.

[41] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[42] Ted Herman. Superstabilizing mutual exclusion. *Distributed Computing*, 13(1):1–17, 2000.

[43] Ted Herman. Models of self-stabilization and sensor networks. In Samir R. Das and Sajal K. Das, editors, *Distributed Computing - IWDC 2003, 5th International Workshop*, volume 2918 of *Lecture Notes in Computer Science*, pages 205–214. Springer, 2003.

[44] Ted Herman and Sriram V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2):41–46, 2000.

[45] Ted Herman and Sébastien Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. In *Proceedings of the First Workshop on Algorithmic Aspects of Wireless Sensor Networks (AlgoSensors'2004)*, number 3121 in Lecture Notes in Computer Science, pages 45–58, Turku, Finland, July 2004. Springer-Verlag.

[46] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 119–131, 1990.

[47] Colette Johnen, Luc Alima, Ajoy K. Datta, and Sébastien Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters (PPL)*, 12(3-4):327–340, 2002.

[48] Colette Johnen, Franck Petit, and Sébastien Tixeuil. Auto-stabilisation et protocoles réseaux. *Technique et Science Informatiques*, 23(8):1027–1056, 2004.

[49] Colette Johnen and Sébastien Tixeuil. Route preserving stabilization. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems*, volume 2704 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2003.

[50] Mehmet Hakan Karaata. Self-stabilizing strong fairness under weak fairness. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):337–345, 2001.

[51] Yoshiaki Katayama, Eiichiro Ueda, Hideo Fujiwara, and Toshimitsu Masuzawa. A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. *J. Parallel Distrib. Comput.*, 62(5):865–884, 2002.

[52] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[53] A.D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[54] Sandeep S. Kulkarni and Mahesh Arumugam. Transformations for write-all-with-collision model' . *Computer Communications*, 29(2):183–199, 2006.

[55] Sandeep S. Kulkarni and Umamaheswaran Arumugam. Collision-free communication in sensor networks. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003*, volume 2704 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2003.

[56] Sandeep S. Kulkarni, Chase Bolen, John Oleszkiewicz, and Andrew Robinson. Alternators in read/write atomicity. *Inf. Process. Lett.*, 93(5):207–215, 2005.

[57] Shay Kutten and Toshimitsu Masuzawa. Output stability versus time till output. In Andrzej Pelc, editor, *DISC*, volume 4731 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2007.

[58] Shay Kutten and Boaz Patt-Shamir. Stabilizing time-adaptive protocols. *Theor. Comput. Sci.*, 220(1):93–111, 1999.

[59] Shay Kutten and David Peleg. Fault-local distributed mending. *J. Algorithms*, 30(1):144–165, 1999.

[60] Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A new self-stabilizing maximal matching algorithm. In *Proceedings of the 14^{th} International Colloquium on Structural Information and Communication Complexity (Sirocco 2007)*, volume 4474, pages 96–108. Springer Verlag, June 2007.

[61] Toshimitsu Masuzawa and Sébastien Tixeuil. Bounding the impact of unbounded attacks in stabilization. In Datta and Gradinariu [15], pages 440–453.

[62] Toshimitsu Masuzawa and Sébastien Tixeuil. On bootstrapping topology knowledge in anonymous networks. In Datta and Gradinariu [15], pages 454–468.

[63] Toshimitsu Masuzawa and Sébastien Tixeuil. Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. *International Journal of Principles and Applications of Information Science and Technology (PAIST)*, 1(1):1–13, December 2007.

[64] Mikhail Nesterenko and Anish Arora. Tolerance to unbounded byzantine faults. In *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 22–. IEEE Computer Society, 2002.

[65] J. R. Norris. *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, New York, NY, first edition, 1997.

[66] R. Perlman. *Interconnexion Networks*. Addison-Wesley, 2000.

[67] Naoshi Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *PODC*, pages 173–179, 1999.

[68] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2001.

[69] Sébastien Tixeuil. *Auto-stabilisation Efficace*. PhD thesis, University of Paris Sud XI, January 2000.

[70] Sébastien Tixeuil. On a space-optimal distributed traversal algorithm. In Datta and Herman [16], pages 216–228.

[71] Sébastien Tixeuil. *Wireless Ad Hoc and Sensor Networks*, chapter Fault-tolerant distributed algorithms for scalable systems, pages 225–256. ISTE, October 2007.

[72] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.

[73] George Varghese and Mahesh Jayaram. The fault span of crash failures. *J. ACM*, 47(2):244–293, 2000.

5 Defining Terms

Configuration : global state of the system at a particular time

Execution : maximal sequence of configurations

Specification : predicate on executions

Daemon : predicate on executions, used to abstract system hypotheses

Self-stabilization : property of a distributed system to eventually converge to a configuration from which every execution assuming a particular daemon conforms to a particular specification

6 Further Information

Advances on all aspects of self-stabilization are reported in the annual Symposium on Stabilization, Safety, and Security of distributed systems (SSS). Theoretical aspects of self-stabilization are also covered by theoretical distributed computing conferences such as PODC (Principles of Distributed Computing), DISC (Distributed Computing), OPODIS (On Principles of Distributed Systems), and Sirocco (International Colloquium on Structural Information and Communication Complexity). Practical aspects of self-stabilization are covered by ICDCS (International Conference on Distributed Computing Systems), DSN (Dependable Systems and Networks), SRDS (Symposium on Reliable Distributed Systems), and the many conferences dedicated to sensor networks and autonomic computing.

A book [22] published in 2000 is dedicated to self-stabilization, and [68, 32, 53] all include a chapter on self-stabilization. [48] surveys self-stabilization in network protocols, while [71] describes self-stabilization with respect to scalability properties.