Self-Stabilizing Byzantine Asynchronous Unison

Swan Dubois[1,3] Maria Gradinariu Potop-Butucaru[1,4] Mikhail Nesterenko[2,5]
Sébastien Tixeuil[1,6]

Abstract

We explore asynchronous unison in the presence of systemic transient and permanent Byzantine faults in shared memory. We observe that the problem is not solvable under less than strongly fair scheduler or for system topologies with maximum node degree greater than two. We present a self-stabilizing Byzantine-tolerant solution to asynchronous unison for chain and ring topologies. Our algorithm has minimum possible containment radius and optimal stabilization time.

1 Introduction

Asynchronous unison *[22] requires processors to maintain synchronization between their counters called* clocks. *Specifically, each processor has to increment its clock indefinitely while the clock* drift *from its neighbors should not exceed* 1. *Asynchronous unison is a fundamental building block for a number of principal tasks in distributed systems such as distributed snapshots [6] and synchronization [1, 2].*

A practical large-scale distributed system must counter a variety of transient and permanent faults. A systemic transient fault may perturb the configuration of the system and leave it in the arbitrary configuration. Self-stabilization *[10, 12, 25] is a versatile technique for transient fault forward recovery.* Byzantine fault *[18] is the most generic permanent fault model: a faulty processor may behave arbitrarily. However, designing distributed systems that handle both transient and permanent faults proved to be rather difficult [8, 13, 23]. Some of the difficulty is due to the inability of the system to counter Byzantine behavior by relying on the information encoded in the global system configuration: a transient fault may place the system in an arbitrary configuration.*

In this context considering joint Byzantine and systemic transient fault tolerance for asynchronous unison appears futile. Indeed, the Byzantine processor may keep setting its clock to an arbitrary value while the clocks of the correct processors are completely out of synchrony. Hence, we are happy to report that the problem is solvable. In this paper we present a shared-memory Byzantine-tolerant self-stabilizing asynchronous unison algorithm that operates chain and ring system topologies. The algorithm operates under a strongly fair scheduler. We show that the problem is unsolvable for any other topology or for less stringent scheduler. Our algorithm achieves minimal fault-containment radius: each correct processor eventually synchronizes with its correct neighbors. We prove our algorithm correct and demonstrate that its stabilization time is asymptotically optimal.

[1]The author is with Université Pierre & Marie Curie and INRIA, France.
[2]The author is with Kent State University, USA.
[3]swan.dubois@lip6.fr
[4]maria.gradinariu@lip6.fr
[5]mikhail@cs.kent.edu
[6]sebastien.tixeuil@lip6.fr

Related work. *The impetus of this work is the study by Dubois et al [14]. They consider joint tolerance to crash faults and systemic transient faults. The key observation that enables this avenue of research is that the definition of asynchronous unison does not preclude the correct processors from decrementing their clocks. This allows the processors to synchronize and maintain unison even while their neighbors may crash or behave arbitrarily.*

There are several pure self-stabilizing solutions to the unison problem [4, 5, 7, 15]. None of those tolerate Byzantine faults.

Classic Byzantine fault tolerance focuses on masking the fault. There are self-stabilizing Byzantine-tolerant clock synchronization algorithms for completely connected synchronous systems both probabilistic [3, 13] and deterministic [11, 17]. The probabilistic and deterministic solutions tolerate up to one-third and one-fourth of faulty processors respectively.

Another approach to joint transient and Byzantine tolerance is containment*. For tasks whose correctness can be checked locally, such as vertex coloring, link coloring or dining philosophers, the fault may be isolated within a region of the system.* Strict-stabilization *guarantees that there exists a containment radius outside of which the processors are not affected by the fault [20, 23, 24]. Yet some problems are not local and do not admit strict stabilization. However, the tolerance requirements may weakened to* strong-stabilization *[19, 21] which allows the processors arbitrarily far from the faulty processor to be affected. The faulty processor can affect the correct processors only a finite number of times. Strong-stabilization enables solution to several problems, such as tree orientation and tree construction.*

2 Model, Definitions and Notation

Program syntax and semantics. *A distributed system consists of n processors that form a communication graph. The processors are nodes in this graph. The edges of this graph are pairs of processors that can communicate with each other. Such pairs are* neighbors*. A* distance *between two processors is the length of the shortest path between them in this communication graph. Each processor contains variables and rules. A variable ranges over a fixed domain of values. A rule is of the form* $\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$*. A* guard *is a boolean predicate over processor variables. A* command *is a sequence of assignment statements. Processor p may mention its variables anywhere in its guards and commands. That is, p can read and update its variables. However, p may not mention the variables of its neighbors on the left-hand-sides of the assignment statements of its commands. That is, p may only read the variables of its neighbors.*

A processor is either correct *or* faulty*. In this paper we consider* crash faults *and* Byzantine faults*. A crashed processor stop the execution of its rules for the remainder of the run. A processor affected by Byzantine fault disregards its program and it may write arbitrary values to variables. Note that, in a given state, a Byzantine processor exhibits the same state to all its neighbors. When the fault type is not explicitly mentioned, the fault is Byzantine.*

An assignment of values to all variables of the system is configuration*. A rule whose guard is* ***true*** *in some system configuration is* enabled *in this configuration, the rule is* disabled *otherwise. An atomic execution of a subset of enabled rules transitions the system from one configuration to another. This transition is a* step*. Note that a faulty processor is assumed to always have an enabled rule and its step consists of writing arbitrary values to its variables. A* run *of a distributed system is a maximal sequence of such transitions. By maximality we mean that the sequence is either infinite or ends in a state where none of the rules are enabled.*

Schedulers. *A* scheduler *(also called* daemon*) is a restriction on the runs to be considered. The schedulers differ by execution semantics and by fairness. The scheduler is* synchronous *if in every run each step contains the execution of every enabled rule. The scheduler is asynchronous otherwise. There are several types of asynchronous schedulers. In the runs of* distributed *(also called* powerset*) scheduler, a step may contain the execution of an arbitrary subset of enabled*

rules. This is the lest restrictive scheduler. In the runs of a central *scheduler, every step contains the execution of exactly one enabled rule. In the runs of* locally central *scheduler, the step may contain the execution of multiple enabled rules as long as none of the rules belong to neighbor processors. Central and locally central schedulers are equivalent. That is, they define the same set of runs. In this paper we consider these two types of schedulers.*

With respect to fairness, the schedulers are classified as follows. The most restrictive is a strongly fair scheduler*. In every run of this scheduler, a rule is executed infinitely often if it is enabled in infinitely many configurations of the run. Note that the strongly fair scheduler requires that the rule is executed even if it continuously keeps being enabled and disabled throughout the run. A less restrictive is* weakly fair scheduler*. In every run of this scheduler, a rule is executed infinitely often if it is enabled in all but finitely many configurations of the run. That is, the rule has to be executed only if it is continuously enabled. An* unfair scheduler *places no fairness restrictions on the runs of the distributed system. Faulty processors are not subject to scheduling restrictions of any of the schedulers: a faulty processor may take no steps during a run or it may take an infinitely many steps.*

Predicates and specifications. *A predicate is a boolean function over program configurations. A configuration* conforms *to some predicate R, if R evaluates to* ***true*** *in this configuration. The configuration* violates *the predicate otherwise. Predicate R is* closed *in a certain program $\mathcal{P}$, if every configuration of a run of $\mathcal{P}$ conforms to R provided that the program starts from a configuration conforming to R. Note that if a program configuration conforms to R and, after the execution of any step of $\mathcal{P}$, the resultant configuration also conforms to R, then R is closed in $\mathcal{P}$.*

A processor specification *for a processor p defines a set of configuration sequences. These sequences are formed by variables of some subset of processors in the system. This subset always includes p itself. A* problem specification*, or just* problem*, defines specifications for each processor of the system. A problem specification in the presence of faults defines specifications for correct processors only. Program $\mathcal{P}$* solves *problem $\mathcal{S}$ under a certain scheduler if every run of $\mathcal{P}$ satisfies the specifications defined by $\mathcal{S}$. A closed predicate I is an* invariant *of program $\mathcal{P}$ with respect to problem $\mathcal{S}$ if every run of $\mathcal{P}$ that starts in a state conforming to I satisfies $\mathcal{S}$. An f-fault d-distance invariant I_{fd} is a particular invariant of $\mathcal{P}$ such that if the system has no more than f processors then in every run that starts in a configuration conforming to I_{fd}, each processor in the distance of at least d away from the fault satisfies the problem $\mathcal{S}$. That is, only correct processors at distance d or higher have to satisfy the specification.*

A program $\mathcal{P}$ is self-stabilizing *to specification $\mathcal{S}$ if every run of $\mathcal{P}$ that starts in an arbitrary configuration contains a configuration conforming to an invariant of $\mathcal{P}$. A program $\mathcal{P}$ is* strictly-stabilizing *for f faults and distance d, denoted* (f, d)-strictly-stabilizing*, to problem $\mathcal{S}$ if $\mathcal{P}$ converges to an f-fault d-distance invariant I_{fd}.*

Unison specification. *Consider the system of processors each of which has a natural number variable c called* clock*. The clock* drift *between two processors is the difference between their clock values. Two neighbor processor are* in unison *if their drift is no more than one.*

Asynchronous unison *specifies that, for every processor p, every program run has to comply with the following two properties.*

Safety: *in every configuration, processor p is in unison with its neighbors;*

Liveness: *the clock of processor p is incremented infinitely often.*

A program that solves the asynchronous unison problem is minimal *if the only variable that each processor has it its clock.*

processor p
constants l, r: left and right neighbors of p
dg_p: degree of p
variable c_p : natural number, clock value of p

rules
end processor rules
leftEndUp: $(dg_p = 1) \wedge (c_p \leq c_r) \longrightarrow c_p := c_r + 1$
leftEndDown: $(dg_p = 1) \wedge (c_p > c_r) \longrightarrow c_p := c_r - 1$
rightEndUp and *rightEndDown* are similar

middle processor operation rules
middleLeftUp: $(dg_p = 2) \wedge (c_p = c_l \vee c_p = c_l - 1) \wedge (c_p \leq c_r) \longrightarrow c_p := c_p + 1$
middleLeftDown: $(dg_p = 2) \wedge (c_p = c_l \vee c_p = c_l + 1) \wedge (c_p > c_r) \longrightarrow c_p := c_p - 1$
middleRightUp and *middleRightDown* are similar

middle processor synchronization rules
syncUp: $(dg_p = 2) \wedge (c_p < c_l - 1) \wedge (c_p < c_r - 1) \longrightarrow c_p := min\{c_l, c_r\}$
syncDown: $(dg_p = 2) \wedge (c_p > c_l + 1) \wedge (c_p > c_r + 1) \longrightarrow c_p := max\{c_l, c_r\}$

Figure 1: $\mathcal{SSU}$: $(1, 0)$-strict-stabilizing asynchronous unison algorithm for chains and rings.

3 Impossibility Results and Model Justification

Dubois et al [14] established a number of impossibility results for asynchronous unison and crash faults. These results are immediately applicable to Byzantine faults as a Byzantine process may emulate the crash fault by never executing a step. We summarize their results in the below theorem.

Theorem 1 ([14]) *There does not exist a minimal (f, d)-strictly-stabilizing solution to the asynchronous unison problem in shared memory for any distance $d \geq 0$ if the communication graph of the distributed system contains processors of degree greater than two or if the number of faults is greater than one or if the scheduler is either unfair or weakly fair.*

The intuition behind the impossibility results is as follows. If the system contains a processor p with at least three neighbors, the neighbors can cycle through their states such that all three are always in unison with p yet p cannot update its clock without breaking unison with at least one neighbor. If the system allows two faults, then the faulty processors may contain such clock values so far apart that if the correct processors stay in unison with the faulty ones then they are not able to synchronize with each other. If the execution scheduler is either unfair or weakly fair then, one correct processors may cycle through its unison states such that its neighbor is never given an opportunity to update its clock.

The results of Theorem 1 leave the following execution model that is still open for solutions: system topology with maximum degree at most two (i.e. a chain or a ring), at most one fault, and a strongly fair scheduler. We pursue solutions for this particular model in the remainder of the paper.

4 $\mathcal{SSU}$: A Strict-Stabilizing Unison for Chains and Rings

In this section we present the $(1, 0)$*-strictly-stabilizing minimal priority algorithm unison algorithm, prove its correctness and evaluate its stabilization performance.*

4.1 Algorithm Description

The algorithm can operate on either chain or ring system topologies. For the description of the algorithm, let us introduce some topological terminology. A middle *processor has two neighbors. An* end *processor has only one. In a ring every processor is a middle processor. A chain has two end processors. We consider the system of processors to be laid out horizontally left to right. We, therefore, speak of left and right neighbor for a processor and left and right ends of a chain.*

Recall that drift *between two processors* p *and* q *is the difference between their clock values. Two processors* p *and* q *are* in unison *if the drift between them is no more than* 1*. An* island *is a segment of correct processors such that for each processor* p*, if its neighbor* q *is also in this island, then* p *and* q *are in unison. A processor with no in-unison neighbors is assumed to be a single-processor island. Note that a faulty processor never belongs to an island. The* width *of an island is the number of processors in this island.*

The main idea of the algorithm is as follows. Processors form islands of processors with synchronized clocks. The algorithm is designed such that the clocks of the processors with adjacent islands drift closer to each other and the islands eventually merge. If a faulty processor restricts the drift of one such island, for example by never changing its clock, the other islands still drift and synchronize with the affected island.

Operation description. *A detailed description of* $\mathcal{SSU}$ *is shown in Figure 1. Specifically,* $\mathcal{SSU}$ *operates as follows. Each processor* p *maintains a single variable* c_p *where it stores its current clock value. That is, our algorithm is minimal.*

We grouped the processor rules into end processor rules and middle processor rules. Middle processor rules are further grouped into: operation — executed when the processor is in unison with at least one of its neighbors, and synchronization — executed otherwise.

At least one rule is always enabled at an end processor. Depending on the clock value of its neighbor, the left end processor either increments or decrements its own clock using rules leftEndUp *and* leftEndDown*. The operation of the right end processor is similar.*

Let us describe the rules of a middle processor. If processor p *is in unison with its left neighbor,* p *can adjust* c_p *to match its right neighbor using rules* middleLeftUp *or* middleLeftDown*. The execution of neither rule breaks the unison of* p *and its left neighbor. Similar adjustment is done for the left neighbor using* middleRightUp *and* middleRightDown*. Note that if* p *is in unison with both of its neighbors and* c_l *and* c_r *differ by 2, none of these rules of* p *are enabled as any changes of* c_p *break the unison with a neighbor of* p*.*

If p *is in unison with neither of its neighbors, and the clocks of the two neighbors are either both greater or both less than the clock of* p*, the processor synchronizes its clock with one of the neighbors using rule* syncDown *or* syncUp*.*

Example operation. *The operation of our algorithm is best understood with an example. Figure 2 shows the operation of* $\mathcal{SSU}$ *on a chain without a permanent fault. Figure 3 illustrates the operation of* $\mathcal{SSU}$ *on a chain with a faulty processor. Figures 4 and 5 show the operation of* $\mathcal{SSU}$ *on rings respectively without and with a faulty processor.*

4.2 Correctness Proof

Chains. *For chains it is sufficient to consider the operation of the algorithm for the case where the faulty processor is at the end of the chain. Indeed, if the faulty processor is in the middle of the chain, the synchronization of the two segments of correct processors is independent*

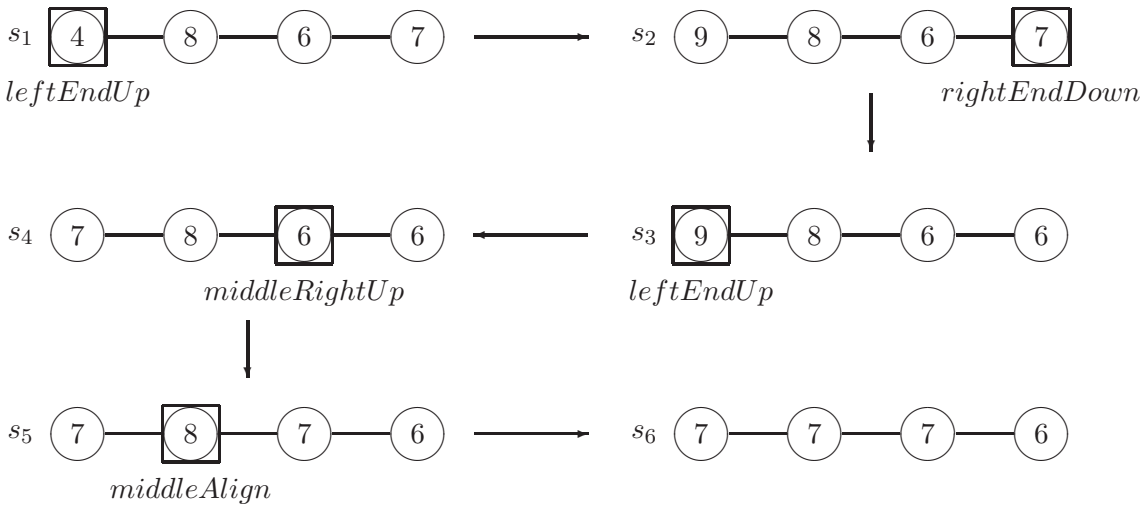

Figure 2: An example operation sequence of $\mathcal{SSU}$ on a chain with no faults. Numbers represent clock values. Squared processor has an enabled rule to be executed.

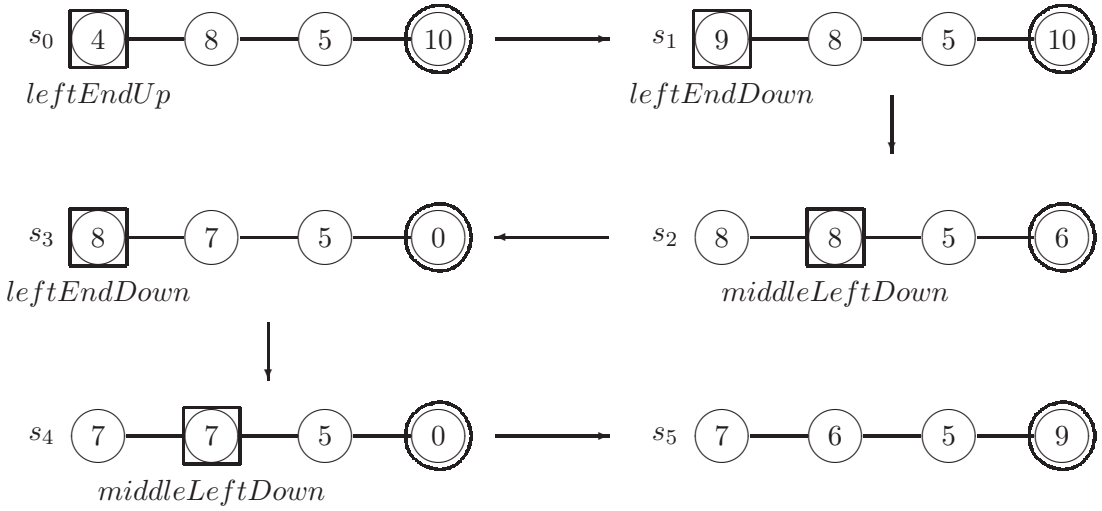

Figure 3: An example operation sequence of $\mathcal{SSU}$ on a chain with a faulty processor. Numbers are processor clock values. The faulty processor is in double circle. Squared processor has an enabled rule to be executed.

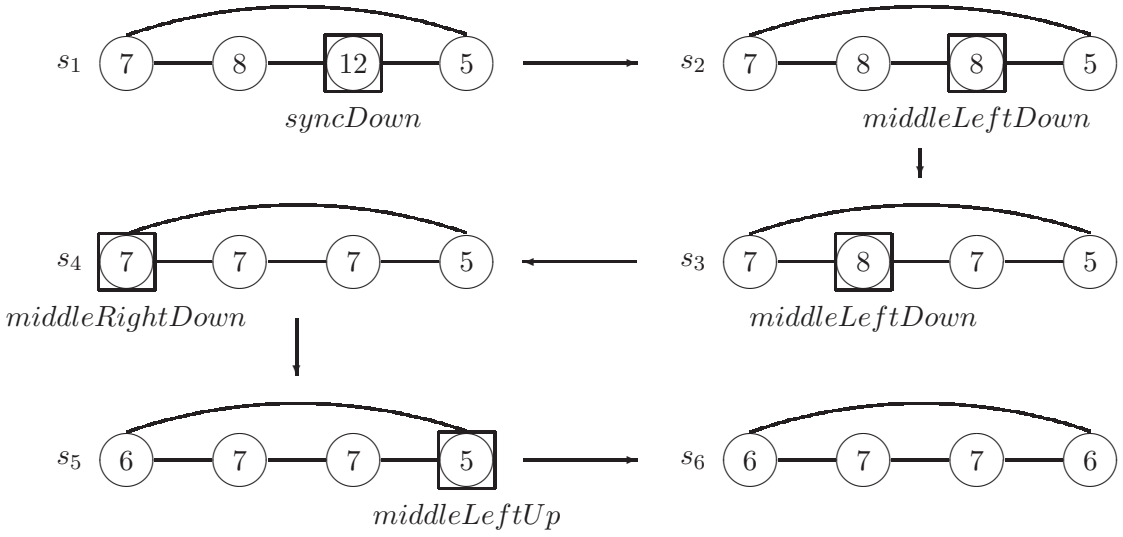

Figure 4: An example operation sequence of $\mathcal{SSU}$ on a ring with no faults. Numbers represent clock values. Squared processor has an enabled rule to be executed.

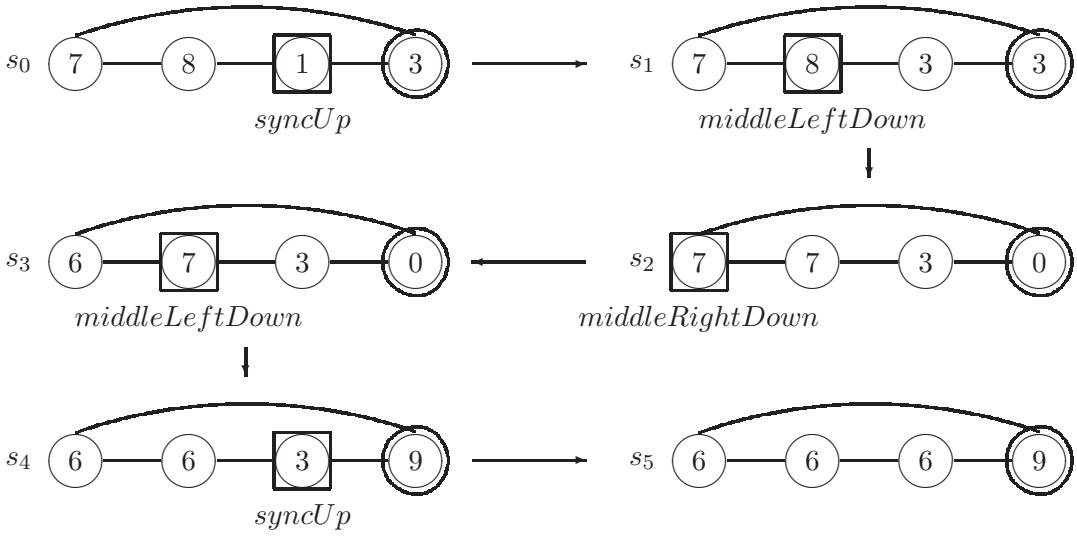

Figure 5: An example operation sequence of $\mathcal{SSU}$ on a chain with a faulty processor. Numbers are processor clock values. The faulty processor is in double circle. Squared processor has an enabled rule to be executed.

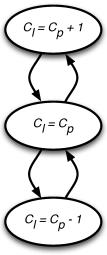

Figure 6: The transitions of in-unison neighbor processors l and p. An illustration for the proof of Lemma 2.

of each other. Thus, without loss of generality, we assume that if there exists a faulty processor in the system, it is always the right end processor.

Lemma 1 *If a run of $\mathcal{SSU}$ on a chain starts from a configuration where two processors p and q belong to the same island, then the two processors belong to the same island in every configuration of this run.*

In other words, Lemma 1 states that an island is never broken. The validity of the lemma can be easily ascertained by the examination of the algorithm's rules as a processor never desynchronizes from its in-unison neighbor.

Lemma 2 *In every run of $\mathcal{SSU}$ on a chain, each processor in the leftmost island takes an infinite number of steps.*

Proof. *The proof is by induction on the width of the island. In every configuration, the left end processor has either* leftEndUp *or* leftEndDown *enabled. Due to the strongly fair scheduler, this processor takes an infinite number of steps in every run.*

Assume that the left neighbor l of processor p that belongs to the leftmost island takes an infinite number of steps in the run. According to Lemma 1, l and p are in unison in every configuration of this run. That is, l and p transition between the three sets of states: $c_l = c_p + 1$, $c_l = c_p$ and $c_l = c_p - 1$. See Figure 6 for illustration. Observe that, regardless of the clock value of the right neighbor of p, if $c_l = c_p$ then p has either middleLeftUp *or* middleLeftDown *rule enabled. If p executes this rule, the system goes either in the state where $c_l = c_p + 1$ or $c_l = c_p - 1$. Since l executes infinitely many steps in the run then a configuration where $c_l = c_p$ repeats infinitely often. That is, one of p's rules are enabled infinitely often in this run. Since the scheduler is strongly fair, p executes infinitely many steps.* □

Lemma 3 *If a run of $\mathcal{SSU}$ on a chain starts from a configuration where processor p belongs to the leftmost island while its right correct neighbor r does not, then this run contains a configuration where both p and r belong to the same island.*

In other words, Lemma 3 claims that every two adjacent islands eventually merge.

Proof. *We prove the lemma by demonstrating that the drift between p and r decreases to zero in every run of $\mathcal{SSU}$. Let us consider the rules of r. The execution of any rule by r can only decrease the drift between the two processors. The execution of the rules by p always decreases the drift as well. According to Lemma 2, p takes infinitely many steps in this run. This means that this run contains a configuration where the drift between p and r is zero.* □

Define the following predicate:

$$INV \equiv \textit{each correct processor is in unison with its correct neighbors}$$

Theorem 2 *Algorithm $\mathcal{SSU}$ on chains stabilizes to INV.*

Proof. *(*sketch*) If every correct processor is in unison with its neighbors, all correct processors belong to a single island. The closure of* INV *follows from Lemma 1. Note that Lemma 3 guarantees that the two leftmost islands eventually merge. The convergence if $\mathcal{SSU}$ to* INV *can be proven by induction on the number of islands in the initial configuration.* □

Theorem 3 *Predicate INV is an $(1,0)$-invariant of $\mathcal{SSU}$ on chains with respect to the asynchronous unison problem.*

In other words, Theorem 3 states that every run of $\mathcal{SSU}$ starting from a configuration conforming to INV *satisfies the specification of asynchronous unison.*

Proof. *The safety property of the asynchronous unison follows immediately from the closure of* INV*. Let us consider the liveness property. Once in unison the only operation that a processor can execute on its clock is increment or decrement. According to Lemma 2, every correct processor of the system takes an infinite number of steps. Since the clock values are natural numbers, each processor is bound to execute an infinite number clock increments. Hence the liveness.* □

Rings. *Since there are no end processors on a ring, we only have to consider the middle processor rules.*

Lemma 4 *If a run of $\mathcal{SSU}$ on a ring starts from a configuration where two processors p and q belong to the same island, then the two processors belong to the same island in every configuration of this run.*

The above lemma is proven similarly to Lemma 1.

Lemma 5 *In every run of $\mathcal{SSU}$ on a ring, there is an island where every processor takes an infinite number of steps.*

Proof. *(*sketch*) Observe that in every configuration of $\mathcal{SSU}$ on a ring, there is at least one correct processor whose clock holds the largest or the smallest value in the system. This processor has a rule enabled. Since we consider a strongly fair scheduler, there are infinitely many steps executed by correct processors in every run of $\mathcal{SSU}$. Since there are finitely many correct processors, at least one correct processor takes infinitely many steps. Let us consider the island to which this processor belongs. The rest of the lemma is proven by induction on the width of this island similar to Lemma 2.* □

Lemma 6 *If a run of $\mathcal{SSU}$ starts from a configuration where there is more than one island, then this run contains a configuration where some two islands merge.*

Proof. *(*sketch*) Let us consider the initial configuration of $\mathcal{SSU}$ on a ring with more than one island. According to Lemma 5, there is at least one island in this configuration where every processor takes an infinite number of steps. Assume, without loss of generality, that this island has an adjacent island to the right. An argument similar to the one employed in the proof of Lemma 3 demonstrates that these islands eventually merge.* □

The below two theorems are proven similarly to their equivalents for the chain topology.

Theorem 4 *Algorithm $\mathcal{SSU}$ on rings stabilizes to INV.*

Theorem 5 *Predicate INV is an $(1,0)$-invariant of $\mathcal{SSU}$ on rings with respect to the asynchronous unison problem.*

4.3 Stabilization Time

In this section, we compute the stabilization time of $\mathcal{SSU}$. We estimate the stabilization time in the number of asynchronous rounds. In general, this notion is somewhat tricky to define for strongly fair scheduler, at the actions of processors may become disabled and then enabled an arbitrary many times before execution. However, this definition simplifies for the case of $\mathcal{SSU}$ as every correct processor takes an infinite number of steps. We define an asynchronous round *to be the smallest segment of a run of the algorithm where every correct process executes a step.*

Upper bound of $\mathcal{SSU}$. *First, we show that $\mathcal{SSU}$ needs at most L rounds to stabilize where L is the largest clock drift between correct processors in the system.*

Theorem 6 *The stabilization time of $\mathcal{SSU}$ is in $O(L)$ rounds both on chains and rings where L is the maximum clock drift between two correct neighbors in the initial configuration.*

Proof. *Assume that there exists an execution ω such that there exists at least two distinct islands I_1 and I_2 at the end of the round L_ω (where L_ω is the maximum clock drift between two correct neighbors in the initial configuration of ω). Note that $L_\omega \geq 2$. Otherwise, any processor is in unison with its neighbor in the initial configuration and Lemma 1 or 4 implies I_1 and I_2 are never distinct.*

Let p and q be two neighbor processors such that $p \in I_1$ and $q \in I_2$. Without loss of generality, we can assume that $c_q < c_p$ in the initial configuration of ω. By construction, we have $c_p - c_q \leq L_\omega$.

While I_1 and I_2 are distinct, according to the proof of Lemma 3 or 6, the following property holds: $c_q < c_p$.

In the case where the system is a chain, note that p and q are not end processors. Otherwise, p and q are in unison at the end of the first round since the end processor synchronizes its clock with the one of its neighbor at its first activation and this contradicts the construction of ω and the fact that $L_\omega \geq 2$.

Now, we can observe that any activation of p by a middle processor operation or synchronization rule can only decrease the clock value of p by at least one.

Following the definition of asynchronous round, there is at least one activation of p during each round of ω. Then, we can conclude that, at the end of the round i ($1 \leq i \leq L_\omega$), we have: $c_p - c_q \leq L_\omega - i$.

We can deduce that p and q are necessarily in unison at the end of the round $L_\omega - 1$ which contradicts the construction of ω. Then, the stabilization time of $\mathcal{SSU}$ is in $O(L)$ rounds both on chains and rings. Hence the result. □

Lower bound on chains. *Then, we show that any $(1,0)$-strictly-stabilizing deterministic minimal asynchronous unison on a chain needs at least L rounds to stabilize where L is the largest clock drift between correct processors in the system.*

In the following lemmas, $\mathcal{A}$ denotes any $(1,0)$-strictly-stabilizing deterministic minimal asynchronous unison on a chain under a central strongly fair scheduler.

Lemma 7 *When a middle processor is in unison with only one of its neighbors, any enabled rule of $\mathcal{A}$ for this processor maintains this unison.*

Proof. *Assume that there exists a set of clock values $\{a, b, c\}$ (with $|a-b| \leq 1$ and $|b-c| \geq 2$) such that a middle processor p is enabled by a rule R of $\mathcal{A}$ when $c_p = b$ and neighbors clock are respectively a and c and that R modifies c_p into a value b' (with $|a - b'| \geq 2$).*

Then, consider the following initial configuration: $V = \{l, p, r\}$, $E = \{\{l, p\}, \{p, r\}\}$, r is Byzantine and $c_l = a$, $c_p = b$, $c_r = c$ (see Figure 7). We can observe that this configuration satisfies INV. By construction, p is enabled by R in this configuration (recall that $\mathcal{A}$ is minimal and deterministic). If the scheduler chooses p, then we obtain a configuration which does not

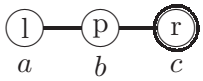

Figure 7: Configuration used in proof of Lemma 7

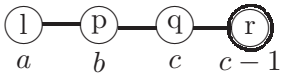

Figure 8: Configuration used in proof of Lemma 8

satisfy INV. Hence, $\mathcal{A}$ does not respect the closure of the safety property of asynchronous unison. This is contradictory with its construction. □

Lemma 8 *When a middle processor p is in unison with only one of its neighbors (denote by q the other neighbor of p), the following property holds: in any execution starting from this configuration in which q remains not synchronized with p, p moves its clock closer to the clock of q in a finite time.*

Proof. *Assume that there exists a set of clock values $\{a, b, c\}$ (with $|a-b| \leq 1$ and $|b-c| \geq 2$) such that there exists an execution ω starting from a configuration (in which $c_p = b$ and neighbors clock are respectively a and c – denote by q the processor such that $c_q = c$) in which q remains not synchronized with p and in which p never moves its clock closer to the clock of q.*

We deal with the case where $b > c$ (the case where $b < c$ is similar). Then, consider the following initial configuration s_0: $V = \{l, p, q, r\}$, $E = \{\{l, p\}, \{p, q\}, \{q, r\}\}$, r is Byzantine and $c_l = a$, $c_p = b$, $c_q = c$, $c_r = c - 1$ (see Figure 8). If r acts as a crashed processor, its clock value remains constant. Then, by Lemma 7, we have $c_q \in \{c, c-1, c-2\}$ in any state of any execution starting from s_0. Hence, p can not distinguish this execution from ω (recall that $\mathcal{A}$ is minimal and deterministic). Consequently, there exists an execution starting from s_0 such that $c_p \geq b$ and $c_q \leq c$ in any state. This contradicts the convergence property of $\mathcal{A}$. □

Lemma 9 *When an end processor is in unison with its neighbor, there exists an enabled rule of $\mathcal{A}$ for this processor.*

Proof. *Assume that there exists a set of clock values $\{a, b\}$ (with $|a - b| \leq 1$) such that an end processor p is not enabled by any rule of $\mathcal{A}$ when $c_p = a$ and its neighbor clock is b.*

Then, consider the following initial configuration: $V = \{p, r\}$, $E = \{\{p, r\}\}$, r is Byzantine and $c_p = a$, $c_r = b$ (see Figure 9). By construction, p is not enabled in this configuration (recall that $\mathcal{A}$ is minimal and deterministic). Assume now that r acts as a crashed processor. Then, we can observe that p is never enabled in this execution, that contradicts the liveness property of $(1, 0)$-strictly-stabilizing asynchronous unison. □

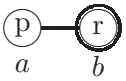

Figure 9: Configuration used in proof of Lemma 9

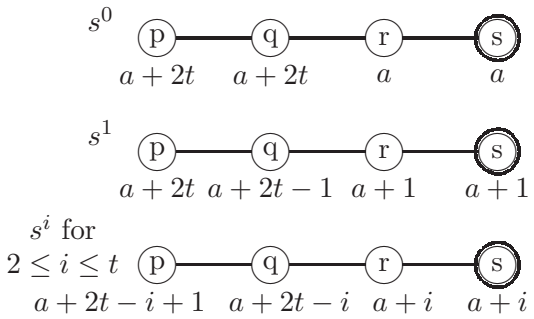

Figure 10: Configurations used in proof of Theorem 7

If we consider the execution described in the proof of Lemma 9, we can observe that p is infinitely often activated (by fairness assumption) and that its clock is always in the set $\{b-1, b, b+1\}$ (by closure of $\mathcal{A}$). Since $\mathcal{A}$ is minimal and deterministic, we can deduce that values of c_p over this execution follow a given cycle. We characterize now $\mathcal{A}$ by this cycle. More formally, we say that:

1. *$\mathcal{A}$ is of type **1** if the cycle is $b, b+1, b, b+1, \ldots$.*
2. *$\mathcal{A}$ is of type **2** if the cycle is $b, b-1, b, b-1, \ldots$.*
3. *$\mathcal{A}$ is of type **3** if the cycle is $b, b+1, b-1, b, b+1, b-1, \ldots$.*

*Notice that the protocol $\mathcal{SSU}$ is of type **1**.*

Theorem 7 *The stabilization time of any $(1,0)$-strictly-stabilizing deterministic minimal asynchronous unison on chains is in $\Omega(L)$ where L is the maximum clock drift between two correct neighbors in the initial configuration.*

Proof. *Assume that $\mathcal{A}$ is a $(1,0)$-strictly-stabilizing deterministic minimal asynchronous unison on chains.*

*We provide the proof of this theorem in the case where $\mathcal{A}$ is of type **1** since other cases are similar.*

Let a, t be natural numbers. Consider the following initial configuration s^0: $V = \{p, q, r, s\}$, $E = \{\{p, q\}, \{q, r\}, \{r, s\}\}$, s is Byzantine and $c_p = a + 2t$, $c_q = a + 2t$, $c_r = a$, $c_s = a$ (see Figure 10). Hence, we have a maximal clock drift of $L = 2t$.

*Note that p is enabled to take the value $a + 2t + 1$ in s^0 (by Lemma 9 and the fact that $\mathcal{A}$ is minimal and of type **1**). By Lemmas 8, 7, and the fact that $\mathcal{A}$ is minimal, we can deduce that q is enabled to take the value $a + 2t - 1$ only when $c_p = a + 2t$. Similar reasoning holds for r which is enabled to take the value $a + 1$ when $c_s = a$.*

Then, the following execution of $\mathcal{A}$ is possible: p is activated and takes value $a + 2t + 1$, p is activated and takes value $a + 2t$ (p is enabled by Lemma 9 and the new value is determined by the type of $\mathcal{A}$), q is activated and takes value $a + 2t - 1$, r is activated and takes value $a + 1$ and s takes the value $a + 1$ (recall that s is byzantine). We obtain the configuration s^1 depicted in Figure 10.

We can observe that the first round R_1 of our execution ends in s^1 and that we have now a maximal clock drift of $a + 2(t - 1)$.

By the same reasoning, we can construct a sequence of $t-1$ rounds $R_i = s^{i-1} \ldots s^i$ $(2 \leq i \leq t)$ as follows: p is activated and takes value $a + 2t + 1 - i$, q is activated and takes value $a + 2t - i$, r is activated and takes value $a + i$ and s takes the value i. We obtain the configuration s^i at the end of round R_i $(2 \leq i \leq t)$ depicted in Figure 10. At the end of round R_i $(2 \leq i \leq t)$, we have a maximal clock drift of $2(t - i)$.

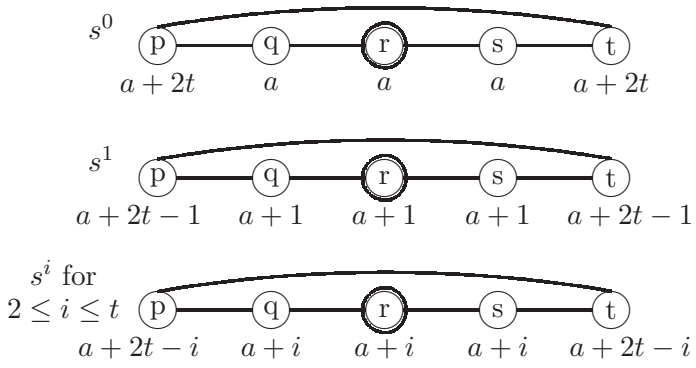

Figure 11: Configurations used in proof of Theorem 8

We can conclude that, at the end of the round R_{t-1}, the maximal clock drift is 2 whereas, at the end of the round R_t, the maximal clock drift is 1 (since we have $c_p - c_q = 1$ and $c_q - c_r = 0$). By construction of t, we can conclude that $\mathcal{A}$ needs $\Omega(L)$ rounds to stabilize. □

Lower bound on rings. *Then, we show that any $(1,0)$-strictly-stabilizing deterministic minimal asynchronous unison on a chain needs at least L rounds to stabilize where L is the largest clock drift between correct processors in the system.*

In the following lemmas, $\mathcal{A}$ denotes any $(1,0)$-strictly-stabilizing deterministic minimal asynchronous unison on a ring under a central strongly fair scheduler.

Lemma 10 *When a processor is in unison with only one of its neighbors, any enabled rule of $\mathcal{A}$ for this processor maintains this unison.*

Proof. *The proof of Lemma 7 directly applies here if we consider the following system: $V = \{p, q, r\}$ and $E = \{\{p, q\}, \{q, r\}, \{r, p\}\}$.* □

Lemma 11 *When a processor p is in unison with only one of its neighbors (denote by q the other neighbor of p), the following property holds: in any execution starting from this configuration in which q remains not synchronized with p, p moves its clock closer to the clock of q in a finite time.*

Proof. *The proof of Lemma 8 directly applies here if we consider the following system: $V = \{p, q, r, s\}$ and $E = \{\{p, q\}, \{q, r\}, \{r, s\}, \{s, p\}\}$.* □

Theorem 8 *The stabilization time of any $(1,0)$-strictly-stabilizing deterministic minimal asynchronous unison on rings is in $\Omega(L)$ where L is the maximum clock drift between two correct neighbors in the initial configuration.*

Proof. *Assume that $\mathcal{A}$ is a $(1,0)$-strictly-stabilizing deterministic minimal asynchronous unison on rings.*

Let a, t be natural numbers. Consider the following initial configuration s^0: $V = \{p, q, r, s, t\}$, $E = \{\{p, q\}, \{q, r\}, \{r, s\}, \{s, t\}, \{t, p\}\}$, r is Byzantine and $c_p = c_t = a + 2t$, $c_q = c_s = c_r = a$ (see Figure 11). Hence, we have a maximal clock drift of $L = 2t$.

Note that p and t are enabled to take the value $a + 2t - 1$ in s^0 (by Lemmas 11 and 10 and the fact that $\mathcal{A}$ is minimal).By similar reasoning, we can deduce that q and s are enabled to take the value $a + 1$.

Then, the following execution of $\mathcal{A}$ is possible: p is activated and takes value $a + 2t - 1$, t is activated and takes value $a + 2t - 1$, q is activated and takes value $a + 1$, s is activated and takes

value $a+1$ and s takes the value $a+1$ (recall that s is byzantine). We obtain the configuration s^1 depicted in Figure 11.

We can observe that the first round R_1 of our execution ends in s^1 and that we have now a maximal clock drift of $a+2(t-1)$.

By the same reasoning, we can construct a sequence of $t-1$ rounds $R_i = s^{i-1} \dots s^i$ $(2 \leq i \leq t)$ as follows: p is activated and takes value $a+2t-i$, t is activated and takes value $a+2t-i$, q is activated and takes value $a+i$, s is activated and takes value $a+i$ and s takes the value $a+i$ (recall that s is byzantine). We obtain the configuration s^i at the end of round R_i $(2 \leq i \leq t)$ depicted in Figure 11. At the end of round R_i $(2 \leq i \leq t)$, we have a maximal clock drift of $2(t-i)$.

We can conclude that, at the end of the round R_{t-1}, the maximal clock drift is 2 whereas, at the end of the round R_t, the maximal clock drift is 0. By construction of t, we can conclude that $\mathcal{A}$ needs $\Omega(L)$ rounds to stabilize. □

Conclusion. *Let us review our conclusions so far. Theorem 6 proves that the stabilization complexity of $\mathcal{SSU}$ is in $O(L)$ rounds while Theorems 7 and 8 show that any $(1,0)$-strict-stabilizing algorithm requires at least that many rounds to stabilize. The following theorem summarizes these results.*

Theorem 9 *The stabilization complexity of $\mathcal{SSU}$ is optimal. It stabilizes in $\Theta(L)$ asynchronous rounds where L is the largest drift between correct processors.*

5 Conclusion

In this paper we explored joint tolerance to Byzantine and systemic transient faults for the asynchronous unison problem in shared memory. The presence of algorithms that tolerate both fault classes poses the question for further study: what are the properties of such algorithms in more concrete execution models of finer atomicity such as shared registers or message-passing. Lower atomicity models tend to empower faulty processors. Indeed, in shared register model, the Byzantine processor on a ring may report differing clock values to its right and left neighbors complicating fault recovery. In our future work we would like to pursue this research question.

References

[1] *Baruch Awerbuch. Complexity of network synchronization.* J. ACM, *32(4):804–823, 1985.*

[2] *Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. A time-optimal self-stabilizing synchronizer using a phase clock.* IEEE Trans. Dependable Sec. Comput., *4(3):180–190, 2007.*

[3] *Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Fast self-stabilizing byzantine tolerant digital clock synchronization. In Rida A. Bazzi and Boaz Patt-Shamir, editors,* PODC, *pages 385–394. ACM, 2008.*

[4] *Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In Soma Chaudhuri and Shay Kutten, editors,* PODC, *pages 150–159. ACM, 2004.*

[5] *Christian Boulinier, Franck Petit, and Vincent Villain. Synchronous vs. asynchronous unison. In Herman and Tixeuil [16], pages 18–32.*

[6] *K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems.* ACM Trans. Comput. Syst., *3(1):63–75, 1985.*

[7] *Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous unison (extended abstract). In* ICDCS, *pages 486–493, 1992.*

[8] *Ariel Daliot and Danny Dolev. Self-stabilization of byzantine protocols. In Herman and Tixeuil [16], pages 48–67.*

[9] *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.*

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

[11] *Danny Dolev and Ezra N. Hoch. On self-stabilizing synchronous actions despite byzantine attacks. In Andrzej Pelc, editor,* DISC*, volume 4731 of* Lecture Notes in Computer Science*, pages 193–207. Springer, 2007.*

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

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

[14] *Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Brief announcement: Dynamic FTSS in Asynchronous Systems: the Case of Unison. In* Proceedings of DISC 2009*, Lecture Notes in Computer Science, Elche, Spain, September 2009. Springer Berlin / Heidelberg.*

[15] *Mohamed G. Gouda and Ted Herman. Stabilizing unison.* Inf. Process. Lett.*, 35(4):171–175, 1990.*

[16] *Ted Herman and Sébastien Tixeuil, editors.* Self-Stabilizing Systems, 7th International Symposium, SSS 2005, Barcelona, Spain, October 26-27, 2005, Proceedings*, volume 3764 of* Lecture Notes in Computer Science*. Springer, 2005.*

[17] *Ezra N. Hoch, Danny Dolev, and Ariel Daliot. Self-stabilizing byzantine digital clock synchronization. In Datta and Gradinariu [9], pages 350–362.*

[18] *Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem.* ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.*

[19] *Toshimitsu Masuzawa and Sébastien Tixeuil. Bounding the impact of unbounded attacks in stabilization. In Datta and Gradinariu [9], pages 440–453.*

[20] *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.*

[21] *Toshimitsu Masuzawa and Sébastien Tixeuil. Strong stabilization: Bounding times affected by byzantine processes in stabilization. In* Asian Association for Algorithms and Computation annual meeting (AAAC 2008)*, Hong Kong, April 2008.*

[22] *Jayadev Misra. Phase synchronization.* Inf. Process. Lett.*, 38(2):101–105, 1991.*

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

[24] *Yusuke Sakurai, Fukuhito Ooshita, and Toshimitsu Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In* Principles of Distributed Systems, 8th International Conference, OPODIS 2004*, volume 3544 of* Lecture Notes in Computer Science*, pages 283–298. Springer, 2005.*

[25] *Sébastien Tixeuil.* Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms. Chapman & Hall/CRC Applied Algorithms and Data Structures. Taylor & Francis, November 2009.*