J. Parallel Distrib. Comput. 72 (2012) 917–923

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Research note

Self-stabilizing byzantine asynchronous unison☆

Swan Dubois [a,*], Maria Potop-Butucaru [a], Mikhail Nesterenko [b], Sébastien Tixeuil [c]

[a] UPMC Sorbonne Universités & Inria, France
[b] Kent State University, USA
[c] UPMC Sorbonne Universités & Institut Universitaire de France, France

ARTICLE INFO

Article history:
Received 10 February 2011
Received in revised form
2 April 2012
Accepted 4 April 2012
Available online 19 April 2012

Keywords:
Self-stabilization
Byzantine containment
Strict stabilization
Unison
Clock synchronization

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 a less than strongly fair scheduler or for system topologies with maximum node degree greater than two.

We present then a self-stabilizing Byzantine-tolerant solution to asynchronous unison for chain and ring topologies under the central strongly fair daemon. Our algorithm has minimum possible containment radius and optimal stabilization time.

1. Introduction

Asynchronous unison [21,6] requires processors to maintain synchrony 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 [5] 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 system and leave it in an arbitrary configuration. *Self-stabilization* [8,10] is a versatile technique for transient fault forward recovery. *Byzantine fault* [23] is the most generic permanent fault model: a faulty processor may behave arbitrarily for an indefinite period of time. However, designing distributed systems that handle both transient and permanent faults proved to be rather difficult [7,11,25]. Some of the difficulty is due to the inability of the system to counter Byzantine behavior by relying on the information contained in the global system configuration: a transient fault may place the system in an arbitrary configuration (at the beginning of the execution).

In the context of the above discussion, considering joint Byzantine and systemic transient fault tolerance for asynchronous unison appears futile (*cf.* impossibility results of [17]). 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 some cases. In this paper we present a shared-memory Byzantine-tolerant self-stabilizing asynchronous unison algorithm that operates on chain and ring system topologies. The algorithm operates under a central strongly fair scheduler. We show that the problem is unsolvable for any other topology or under less stringent scheduler. Our algorithm achieves minimal fault-containment radius: each correct processor eventually synchronizes with its correct neighbors. We prove that our algorithm is correct and demonstrate that its stabilization time is asymptotically optimal.

Related work. The impetus of the present research is the result by Dubois et al. [17]. They consider joint tolerance to crash faults and systemic transient faults. The key observation that enables this avenue of research is that the adopted 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 [21,6,4]. 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 [11,3] and deterministic [9,22]. The probabilistic and

☆ A preliminary version of this work appeared in [16].

* Corresponding author.
E-mail address: swan.dubois@lip6.fr (S. Dubois).

0743-7315/$ – see front matter
doi:10.1016/j.jpdc.2012.04.001

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 [25,24,26,20]. Yet some problems are not local and do not admit strict stabilization. However, the tolerance requirements may be weakened to *strong-stabilization* [15] which allows the processors outside the containment radius to be affected by Byzantine processors after the convergence of the system. The faulty processors can affect these correct processors only a finite number of times after the convergence of the system. Strong-stabilization enables solution to several problems, such as tree orientation and tree construction. Further weakening of the Byzantine containment properties is expected as the problem becomes narrower: for maximal metric trees and related problems, both strict and strong stabilization need to be relaxed using topology awareness [12–14].

Outline. This paper is structured as follows. In Section 2, we formally state the model used in the remainder, Section 3 sums up previous impossibility results while Section 4 presents our algorithm. We prove its correctness and its time optimality respectively in Section 5 and in Section 6. Section 7 concludes this paper.

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. However, the left-hand-side of an assignment statement any command of p may not mention the variables of its neighbors. That is, p can read and update its variables. However, p may not mention the variables of its neighbors on the left-hand-side of the assignment statements of its commands. That is, p may read but not update 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 a Byzantine fault may disregard its program and change the values of its variables arbitrarily. Note that in shared memory even for a Byzantine processor, if a variable holds a certain value, then all its neighbors read this value. That is, the neighbors of a faulty processor cannot simultaneously read different values of the variable. When the fault type is not explicitly mentioned, the fault is Byzantine.

An assignment of values to all variables of the system is a *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 Byzantine 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 sequence of such transitions.

Schedulers. A *scheduler*, also called *daemon*, is a restriction on the runs to be considered. The schedulers differ by execution semantics and by fairness [19]. The scheduler is *synchronous* if in every run each step contains the execution of every enabled rule of any correct processor. The scheduler is asynchronous otherwise. There are several types of asynchronous schedulers. In the runs of *distributed* scheduler, also called *powerset*, a step may contain the execution of an arbitrary subset of enabled rules of correct processors. This is the least restrictive scheduler. In the runs of a *central* scheduler, every step contains the execution of exactly one enabled rule of one correct processor. In the runs of *locally central* scheduler, the step may contain the execution of multiple enabled rules of correct processors 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 of a correct processor 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 the *weakly fair scheduler*. In every run of this scheduler, a rule of a correct processors 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 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 faulty processors then in every run that starts in a configuration conforming to I_{fd}, each processor in the distance of strictly more than d hops away from any fault satisfies the problem $\mathcal{S}$. That is, only correct processors at distance d or higher from a faulty processor have to satisfy the specification.

A program $\mathcal{P}$ is *self-stabilizing* [8] to specification $\mathcal{S}$ if every run of $\mathcal{P}$ that starts in an arbitrary configuration contains a configuration conforming to an invariant of $\mathcal{P}$ with respect to problem $\mathcal{S}$. A program $\mathcal{P}$ is *strictly stabilizing* [25] 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 of $\mathcal{P}$ with respect to problem $\mathcal{S}$.

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 absolute difference between their

clock values. Two neighboring processors are *in unison* if their drift is no more than one.

The classical definition of *asynchronous unison* [21,6], specifies that, for every (correct) 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 p is incremented infinitely often and never decremented.

Proposition 1. *There does not exist a strictly stabilizing solution to Byzantine faults for classic asynchronous unison (even when there is only one Byzantine processor).*

An informal argument follows. Consider the following initial configuration: the Byzantine processor b has a clock value of 0 and any correct processor has a clock value equal to the distance between it and b. Then, this configuration satisfies the safety requirement of the problem. Assume now that the Byzantine processor takes no actions and keeps its clock value to 0. Remember that asynchronism of the system implies that this execution is indistinguishable from the one where b is a correct processor and is very slow. Consequently, no correct processor can increment its clock without violating the safety requirement of the problem from this configuration. Hence, no correct processor can increment its clock infinitely often in any run starting from this configuration.

To make the problem solvable we weaken the specification of asynchronous unison as follows. For every correct processor p, every program run has to satisfy the following properties.

Safety: in every configuration, processor p is in unison with its correct neighbors;
Liveness: the clock of processor p is incremented infinitely often.

This specification is weaker than the classic one since it allows both increments and decrements as long as the processors remain in synchrony.

At this step, one may think about a very simple solution to the problem. The idea of this solution is to allow clocks of correct processors to cycle between the values 0 and 1 whatever is the clock of the Byzantine processor. This solution is composed of two rules. The first one sets the clock value of the processor to 1 when its value is 0. The second one sets the clock value of the processor to 0 when its value is not 0. This simple solution ensures our liveness property but is not strictly stabilizing since the closure of the safety property is not guaranteed. Consider the following counter-example: in the initial configuration of the system, any clock has the same value (strictly greater than 2), say 15 for example and there is no Byzantine processor (remember that clock values are unbounded integers by specification). Note that this configuration satisfies the safety condition of our problem (the drift clock between any two correct processors is at most one). Then, any correct processor is enabled by the algorithm. Assume now that the scheduler chooses only one correct processor, the next configuration does not satisfy the safety condition (since the chosen processor takes the clock value 0 whereas its correct neighbors keep the clock value 15).

A program that solves the asynchronous unison problem is *minimal* if the only variable that each processor has is its clock. However, note that it may have some constants.

3. Impossibility results and model justification

Dubois et al. [17] established a number of impossibility results for asynchronous unison and crash faults. These results are immediately applicable to Byzantine faults as a Byzantine processor may emulate the crash fault by never executing a step. We summarize their results in the theorem below.

Theorem 1 ([17])**.** *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 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 processor may cycle through its unison states such that its neighbor is never given an opportunity to update its clock.

Theorem 1 implies that the execution model that may still admit a solution to the asynchronous unison problem is as follows. The system has a maximum degree of two, that is the system is a ring or a chain. There is at most one fault and the scheduling is strongly fair. We pursue solutions for this particular model in the remainder of the paper.

4. $\mathcal{SSU}$: a strictly stabilizing unison for chains and rings

In this section we present the $(1, 0)$-strictly stabilizing minimal unison algorithm, $\mathcal{SSU}$. We prove its correctness and evaluate its performance in the following sections.

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 neighbors for a processor and left and right ends of a chain. This global orientation of the chain is only assumed for the purposes of exposition, we do not require that the local orientation of processors is globally consistent (that is, the labeling of right and left neighbor is arbitrary).

Recall that *drift* between two processors p and q is the absolute value of 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 by definition). 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 description of $\mathcal{SSU}$ is shown in Fig. 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.

processor p
constants l, r: left and right neighbors of p (this labelling is arbitrary and we do not require that it is consistent with the one of neighborhood processors)
dg_p: degree of p
variable c_p : natural number, clock value of p
rules
end processor rules
leftEndUp: $(dg_p = 1) \wedge ((c_p = c_r) \vee (c_p = c_r - 1)) \longrightarrow c_p := c_p + 1$
leftEndDown: $(dg_p = 1) \wedge ((c_p \geq c_r + 1) \vee (c_p < c_r - 1)) \longrightarrow c_p := c_r - 1$
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: $(dg_p = 2) \wedge (c_p > c_l + 1) \wedge (c_p > c_r + 1) \longrightarrow c_p := max\{c_l, c_r\}$

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

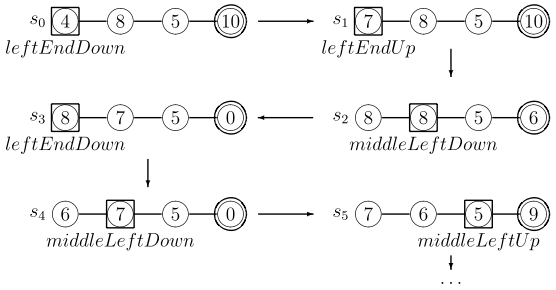

Fig. 2. An example operation sequence of $\mathcal{SSU}$ on a chain with a faulty processor. Numbers are processor clock values. The faulty processor is in double circles. The squared processor has an enabled rule to be executed.

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. Fig. 2 illustrates the operation of $\mathcal{SSU}$ on a chain with a faulty processor. Figs. 3 and 4 show the operation of $\mathcal{SSU}$ on rings respectively without and with a faulty processor.

5. Correctness proof of $\mathcal{SSU}$

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

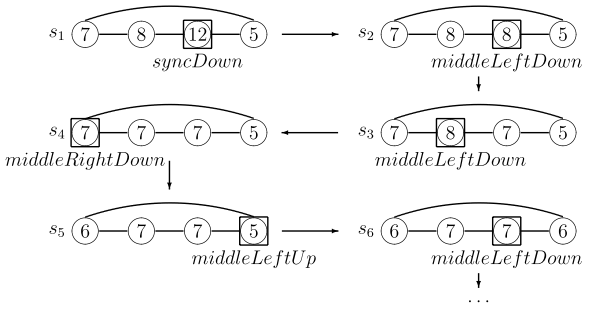

Fig. 3. An example operation sequence of $\mathcal{SSU}$ on a ring with no faults. Numbers represent clock values. The squared processor has an enabled rule to be executed.

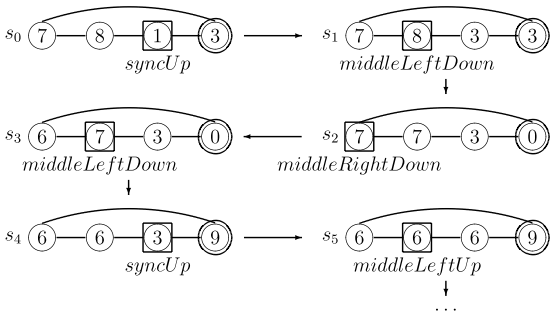

Fig. 4. An example operation sequence of $\mathcal{SSU}$ on a chain with a faulty processor. Numbers are processor clock values. The faulty processor is in double circles. The squared processor has an enabled rule to be executed.

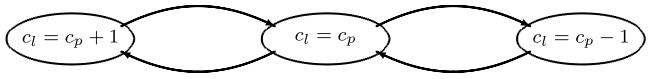

Fig. 5. The transitions of in-unison neighbor processors l and p. An illustration for the proof of Lemma 2.

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.*

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 de-synchronizes 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 Fig. 5 for an 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 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.*

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 one 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* ≡ *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*-fault* 0*-distance 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.

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 of 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 are always a largest and a smallest clock value. Hence, there is at least one correct processor whose clock holds the largest or the smallest value in the system. Indeed, in the worst case, the Byzantine processor holds only one of them. This correct processor has a rule enabled. Consequently, in every configuration of $\mathcal{SSU}$ on a ring, there exists at least one enabled correct processor and then, there are infinitely many steps executed by correct processors in every run of $\mathcal{SSU}$ since we consider a strongly fair scheduler. 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 there exists two neighbor processors p and q that belong to two distinct islands in this configuration such that this run contains a configuration where both p and q belong to the same island.*

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 two theorems below are similar to their equivalents for chains.

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

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

6. Stabilization time of $\mathcal{SSU}$

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 a strongly fair scheduler, as the actions of processors may become disabled and then enabled an arbitrary number of 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 processor executes a step.

Lemma 7. *The stabilization time of $\mathcal{SSU}$ is in $O(L)$ asynchronous rounds where L is the largest clock drift between correct processors in the initial configuration of the system.*

Proof (*Sketch*)**.** For this discussion we assume that the system topology is a chain. The argument for a ring is similar. We first argue that the largest drift between correct processors does not increase. The only actions where a correct processor changes its clock value by more than one are *syncUp* and *syncDown*. However, in *syncUp*, the processor selects the minimum of the clock values of its neighbors. Since the number of faults is no more than one, out of the two neighbors one has to be correct. Therefore, the execution of either rule does not increase the largest drift. A correct processor p whose neighbor q does not share an island with it can only change its clock to decrease the drift between p and q. That is, in every round, this drift decreases by one. In the worst case it takes L rounds to synchronize p and q. Hence the theorem. □

Then, we prove the optimality of $\mathcal{SSU}$ by providing the lower bound of the stabilization time of the asynchronous unison. Complete proofs are available in [18].

Lemma 8. *If a run of any* $(1, 0)$*-strictly stabilizing solution to the minimal asynchronous unison starts from a configuration where two neighbor processors are in unison, these two processors are in unison in every configuration of this run.*

To rephrase the lemma, in a solution to the asynchronous unison a processor cannot break unison.

Proof. Assume the opposite. That is, there is an algorithm $\mathcal{A}$ that solves the asynchronous unison problem yet there is a run of $\mathcal{A}$ such that it starts from a configuration where two neighbor processors u and v are in unison yet this run contains a configuration where u and v are not in unison. That is, this run contains two sequential configurations cn_1 and cn_2 such that u and v are in unison in cn_1, then one of the two processors executes a rule and moves the system into cn_2 where the two processors are not in unison. Let this processor be u. If u has a neighbor besides v, let w be the other neighbor of u. Consider a configuration cn_3 where the clock values of u, v and w are as in cn_1 and all processors of the system, except for possibly w, are in unison while w is faulty. Since the states of u and its neighbors are the same in cn_1 and cn_3, the rule of u that breaks the unison with v is enabled in cn_3. However, all correct processors are in unison in cn_3. Thus, the execution of a rule that breaks the unison between two correct processors. This violates the safety property of the asynchronous unison problem. That means $\mathcal{A}$ is not a solution to the problem and our initial assumption is incorrect. □

Lemma 9. *Any* $(1, 0)$*-strictly stabilizing solution to the minimal asynchronous unison requires* $\Omega(L)$ *stabilization rounds where* L *is the largest initial drift between correct processors in the initial configuration of the system.*

Proof. According to Lemma 8, no solution can break a unison between correct neighbors during stabilization. This means in-unison processors can increase or decrease their clock value by at most one. Consider an initial configuration of the system where the processors are grouped into two islands whose drift is L such that there is at least a pair of processors in each group. This system then can achieve unison in no less than L rounds. □

Let us review our conclusions so far. Lemma 7 proves that the stabilization complexity of $\mathcal{SSU}$ is in $O(L)$ while Lemma 9 shows that any $(1, 0)$-strictly stabilizing algorithm requires at least that many rounds to stabilize. The following theorem summarizes these results.

Theorem 6. *The stabilization complexity of* $\mathcal{SSU}$ *is optimal. It stabilizes in* $\Theta(L)$ *asynchronous rounds where* L *is the largest drift between correct processors in the initial configuration of the system.*

7. Conclusion

In this paper we explored joint tolerance to Byzantine and systemic transient faults for the asynchronous unison problem in shared memory. We presented a $(1, 0)$-strictly stabilizing solution to this problem on chain and ring topologies under central strongly fair scheduler. On the basis of previously published results, we demonstrate that all the assumptions of the execution model of the algorithm are necessary except for possible weakening of the scheduler. Solutions under distributed scheduler, that allows multiple concurrent steps, remain to be explored. Another way to complete these results is to consider bounded clocks.

The existence of a solution for shared memory execution model opens another avenue of research. It is interesting to consider the existence of a solution in lower atomicity models such as shared register or message-passing. We conjecture that a solution in such model is more difficult to obtain as the lower atomicity tends to empower faulty processors. Indeed, in the shared-register model a Byzantine processor may report differing clock values to its right and left neighbor. Such behavior makes a single fault ring topology essentially equivalent to two fault chain topology. The latter is proven unsolvable. Hence, we posit that in the lower atomicity models, the only topology that allows a solution to asynchronous unison to the chain.

References

[1] Baruch Awerbuch, Complexity of network synchronization, Journal of the ACM 32 (4) (1985) 804–823.
[2] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, George Varghese, A time-optimal self-stabilizing synchronizer using a phase clock, IEEE Transactions on Dependable and Secure Computing 4 (3) (2007) 180–190.
[3] Michael Ben-Or, Danny Dolev, Ezra N. Hoch, Fast self-stabilizing byzantine tolerant digital clock synchronization, in: PODC, 2008, pp. 385–394.
[4] Christian Boulinier, Franck Petit, Vincent Villain, When graph theory helps self-stabilization, in: PODC, 2004, pp. 150–159.
[5] K. Mani Chandy, Leslie Lamport, Distributed snapshots: Determining global states of distributed systems, ACM Transactions on Computer Systems 3 (1) (1985) 63–75.
[6] Jean-Michel Couvreur, Nissim Francez, Mohamed G. Gouda, Asynchronous unison (extended abstract), in: ICDCS, 1992, pp. 486–493.
[7] Ariel Daliot, Danny Dolev, Self-stabilization of byzantine protocols, in: SSS, 2005, pp. 48–67.
[8] Edsger W. Dijkstra, Self-stabilizing systems in spite of distributed control, Communication of the ACM 17 (11) (1974) 643–644.
[9] Danny Dolev, Ezra N. Hoch, On self-stabilizing synchronous actions despite byzantine attacks, in: DISC, 2007, pp. 193–207.
[10] Shlomi Dolev, Self-stabilization, MIT Press, 2000.
[11] Shlomi Dolev, Jennifer L. Welch, Self-stabilizing clock synchronization in the presence of byzantine faults, Journal of the ACM 51 (5) (2004) 780–799.
[12] Swan Dubois, Toshimitsu Masuzawa, Sébastien Tixeuil, The impact of topology on byzantine containment in stabilization, in: DISC, 2010, pp. 495–509.
[13] Swan Dubois, Toshimitsu Masuzawa, Sébastien Tixeuil, On byzantine containment properties of the min+1 protocol, in: SSS, 2010, pp. 96–110.
[14] Swan Dubois, Toshimitsu Masuzawa, Sébastien Tixeuil, Maximum metric spanning tree made byzantine tolerant, in: DISC, 2011, pp. 150–164.
[15] Swan Dubois, Toshimitsu Masuzawa, Sébastien Tixeuil, Bounding the impact of unbounded attacks in stabilization, IEEE Transactions on Parallel and Distributed Systems 23 (3) (2012) 460–466.
[16] Swan Dubois, Maria Potop-Butucaru, Mikhail Nesterenko, Sébastien Tixeuil, Self-stabilizing byzantine asynchronous unison, in: OPODIS, 2010, pp. 83–86.
[17] Swan Dubois, Maria Potop-Butucaru, Sébastien Tixeuil, Dynamic ftss in asynchronous systems: the case of unison, Theoretical Computer Science 412 (29) (2011) 3418–3439.
[18] Swan Dubois, Maria Gradinariu Potop-Butucaru, Mikhail Nesterenko, Sébastien Tixeuil, Self-stabilizing byzantine asynchronous unison, Technical report, 2009.
[19] Swan Dubois, Sébastien Tixeuil, A taxonomy of daemons in self-stabilization, Technical report, 2011.
[20] Swan Dubois, Sébastien Tixeuil, Nini Zhu, The byzantine brides problem, in: FUN, 2012, page to appear.
[21] Mohamed G. Gouda, Ted Herman, Stabilizing unison, Information Processing Letters 35 (4) (1990) 171–175.
[22] Ezra N. Hoch, Danny Dolev, Ariel Daliot, Self-stabilizing byzantine digital clock synchronization, in: SSS, 2006, pp. 350–362.
[23] Leslie Lamport, Robert E. Shostak, Marshall C. Pease, The byzantine generals problem, ACM Transactions on Programming Languages and Systems 4 (3) (1982) 382–401.
[24] Toshimitsu Masuzawa, Sébastien Tixeuil, Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults, International Journal of Principles and Applications of Information Science and Technology 1 (1) (2007) 1–13.
[25] Mikhail Nesterenko, Anish Arora, Tolerance to unbounded byzantine faults, in: SRDS, 2002, pp. 22–29.
[26] Yusuke Sakurai, Fukuhito Ooshita, Toshimitsu Masuzawa, A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks, in: OPODIS, 2005, pp. 283–298.

Swan Dubois is currently a Ph.D. student in computer science at the University Pierre and Marie Curie—Paris 6 and at the INRIA (France). His research interests are fault-tolerance in distributed systems (especially self-stabilization) and graph theory.

Maria Potop-Butucaru is an Associate Professor at the University Pierre and Marie Curie—Paris 6 and a researcher in the Regal team of INRIA Rocquencourt (France). Her main research interests are in fault-tolerant distributed systems and dynamic systems (sensor networks, robot networks, and P2P overlays).

Mikhail Nesterenko got his Ph.D. in 1998 from Kansas State University. Presently he is a full professor at Kent State University. He is interested in wireless networking, distributed algorithms and fault-tolerance.

Sébastien Tixeuil is a full professor at the University Pierre and Marie Curie—Paris 6 (France) and Institut Universitaire de France, where he leads the NPA research group. He received his Ph.D. from the University of Paris Sud-XI in 2000. His research interests include fault and attack tolerance in dynamic networks and systems.