Theoretical Computer Science 412 (2011) 3418–3439

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Dynamic FTSS in asynchronous systems: The case of unison☆,☆☆

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

[a] UPMC Sorbonne Universités, France
[b] INRIA Rocquencourt, Project-Team REGAL, France
[c] Institut Universitaire de France, France

ARTICLE INFO

Article history:
Received 10 December 2009
Received in revised form 31 January 2011
Accepted 6 February 2011
Communicated by D. Peleg

Keywords:
Distributed algorithms
Self-stabilization
Fault-tolerance
Unison
Clock synchronization

ABSTRACT

Distributed fault-tolerance can mask the effect of a limited number of permanent faults, while self-stabilization provides forward recovery after an arbitrary number of transient faults hit the system. FTSS (Fault-Tolerant Self-Stabilizing) protocols combine the best of both worlds since they tolerate simultaneously transient and (permanent) crash faults. To date, deterministic FTSS solutions either consider static (*i.e.* fixed point) tasks, or assume synchronous scheduling of the system components.

In this paper, we present the first study of deterministic FTSS solutions for dynamic tasks in asynchronous systems, considering the unison problem as a benchmark. Unison can be seen as a local clock synchronization problem as neighbors must maintain digital clocks at most one time unit away from each other, and increment their own clock value infinitely often. We present several impossibility results for this difficult problem and propose an FTSS solution (when the problem is solvable) for the state model that exhibits optimal fault-containment.

1. Introduction

The advent of ubiquitous large-scale distributed systems advocates that tolerance to various kinds of faults and hazards must be included from the very early design of such systems. *Self-stabilization* [8,10] is a versatile technique that permits forward recovery from any kind of *transient* fault, while *Fault-tolerance* [17] is traditionally used to mask the effect of a limited number of *permanent* faults. Making distributed systems tolerant to both transient and permanent faults is appealing yet proved difficult [1,16,18] as impossibility results are expected in many cases.

The seminal works of [1,18] define FTSS protocols as protocols that are both fault-tolerant and self-stabilizing, *i.e.* able to tolerate a few crash faults as well as arbitrary initial memory corruption. In [1], impossibility results for size computation and election in asynchronous systems are presented, while unique naming is proved possible. In [18], a general transformer is presented for synchronous systems, as well as positive results with failure detectors. The transformer of [18] was later proved impossible to transpose to asynchronous systems due to the impossibility of tight synchronization in the FTSS context. For *local* tasks (*i.e.* tasks whose correctness can be checked locally, such as vertex coloring), the notion of *strict* stabilization was proposed [28,26]. Strict stabilization guarantees that there exists a *containment radius* outside which the effect of permanent faults is masked, provided that the problem specification makes it possible to break the causality chain that is caused by the faults. Strong stabilization [25,14,15] weakens this requirement and ensures processes outside the containment radius are only impacted a finite number of times by the Byzantine nodes.

☆ This work was funded in part by ANR project SHAMAN, ALADDIN, and SPADES.

☆☆ A preliminary version of this work was published as a 2-pages brief announcement in DISC'09 (Dubois et al. 2009) [16].

* Corresponding address: LIP6, Case 26-00/225, 4 place Jussieu, 75005 Paris, France. Tel.: +33 1 44 27 73 46; fax: +33 1 44 27 74 95.
E-mail addresses: swan.dubois@lip6.fr (S. Dubois), maria.gradinariu@lip6.fr (M. Potop-Butucaru), sebastien.tixeuil@lip6.fr (S. Tixeuil).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.02.012

Table 1
Summary of results.

	Unfair	Weakly fair		Strongly fair		
		Minimal	Priority	$\Delta \geq 3$		$\Delta \leq 2$
				Minimal	Priority	
$f = 1$	Impossible (Proposition 2)	Impossible (Proposition 3)	Impossible (Proposition 4)	Impossible (Proposition 5)	Impossible (Proposition 6)	Possible (Proposition 11)
$f \geq 2$	Impossible (Proposition 1)					

It turns out that FTSS possibility results in fully *asynchronous* systems known to date are restricted to *static* tasks, *i.e.* tasks that require eventual convergence to some global fixed point (tasks such as naming or vertex coloring fall in this category). In this paper, we consider the more challenging problem of *dynamic* tasks, *i.e.* tasks that require both eventual safety and liveness properties (examples of such tasks are clock synchronization and token passing). Due to the aforementioned impossibility of tight clock synchronization, we consider the *unison* problem, which can be seen as a *local* clock synchronization problem. In the unison problem [27], each node is expected to keep its digital clock value within one time unit of every of its neighbors' clock values (weak synchronization), and increment its clock value infinitely often (liveness). Note that in synchronous systems where the underlying topology is a fully connected graph in which clocks have discrete time unit values, unison induces tight clock synchronization. Several self-stabilizing solutions exist for this problem [3–5,20], both in synchronous and asynchronous systems, yet none of those can tolerate crash faults.

As a matter of fact, there exists a number of FTSS results for *dynamic* tasks in *synchronous* systems. [12,29] provide self-stabilizing clock synchronization that is also *wait free*, *i.e* that tolerate napping faults, in complete networks. Also [11] presents a FTSS clock synchronization for general networks. Still in synchronous systems, it was proved that even *malicious* (*i.e.* Byzantine) faults can be tolerated, to some extent. In [2,13], probabilistic FTSS protocols were proposed for up to one third of Byzantine processors, while in [9,22] deterministic solution tolerate up to one fourth and one third of Byzantine processors, respectively. Note that all solutions presented in this paragraph are for fully *synchronous* systems. [21] is a notable exception since it proposes a probabilistic solution to a clock synchronization problem in an asynchronous system.

In this paper, we tackle the open issue of FTSS *deterministic* solutions to *dynamic* tasks in *asynchronous* systems, using the unison problem as a case study. Our first negative results show that whenever two or more crash faults may occur, FTSS unison is impossible in any asynchronous setting. The remaining case of one crash fault drives the most interesting results (see Section 3). The first main contribution of the paper is the characterization of two key properties satisfied by all previous self-stabilizing asynchronous unison protocols: *minimality* and *priority*. Minimality means that nodes maintain no extra variables but the digital clock value. Priority means that if incrementing the clock value does not break the local safety predicate between neighbors, then the clock value is actually incremented in a finite number of activations, even if no neighbor modifies its clock value. Then, depending on the fairness properties of the scheduling of nodes, we provide various results with respect to the possibility or impossibility of unison. When the scheduling is *unfair* (only global progress is guaranteed), *universal* FTSS unison (*i.e.* unison that can operate on every graph of a particular class) is impossible. When the scheduling is *weakly fair* (a processor that is continuously enabled is eventually activated), then it is impossible to solve universal FTSS unison by a protocol that satisfies either minimality or priority. The case of *strongly fair* scheduling (a processor that is enabled infinitely often is eventually activated) is similar whenever the maximum degree of the graph is at least three. Our negative results still apply when the clock variable is unbounded, the local synchronization constraint is relaxed, and the scheduling is central (*i.e.* a single processor is activated at any time).

On the positive side (Section 4), we present a universal FTSS protocol for connected networks of maximum degree at most two (*i.e.* rings and chains), which satisfies both minimality and priority properties. This protocol makes minimal system hypothesis with respect to the aforementioned impossibility results (maximum degree, fairness of the scheduling, etc.) and is optimal with respect to the containment radius that is achieved (*no* correct processor is *ever* prevented from incrementing its clock). This protocol assumes that the scheduling is central. Table 1 provides a summary of the main results of the paper. Remaining open questions are discussed in Section 5.

2. Model, problem and specifications

We model the network as an undirected connected graph $G = (V, E)$ where V is a set of processors and E is a binary relation that denotes the ability for two processors to communicate ($(p, q) \in E$ if and only if p and q are *neighbors*). We consider only *anonymous* systems (*i.e.* there exists no unique identifiers for each processor) but we assume that every processor p can distinguish its neighbors and locally label them. Each processor p maintains N_p, the set of its neighbors' local labels. In the following, n denotes the number of processors, and Δ the maximal degree. If p and q are two processors of the network, we denote by $d(p, q)$ the length of the shortest path between p and q (*i.e* the *distance* from p to q). In this paper, we assume that the network can be hit by *crash faults*, *i.e.* some processors can stop executing their actions permanently and without any warning to their neighborhood. Since the system is assumed to be fully asynchronous, no processor can detect if one of its neighbors is crashed or slow.

We consider the classical local shared memory model of computation (see [10]) where communications between neighbors are modeled by direct reading of variables instead of exchange of messages. In this model, the program of

every processor consists of a set of shared variables (henceforth, referred to as *variables*) and a finite set of *rules*. A processor can write to its own variables only, and read its own variables and those of its neighbors. Each rule consists of: $<label>::<guard> \longrightarrow <statement>$. The label of a rule is simply a name to refer the action in the text. The guard of a rule in the program of p is a Boolean predicate involving variables of p and its neighbors. The statement of a rule of p updates one or more variables of p. A statement can be executed only if the corresponding guard is satisfied (*i.e.* it evaluates to true). The processor rule is then *enabled*, and processor p is *enabled* in $\gamma \in \Gamma$ if and only if at least one rule is enabled for p in γ. The state of a processor is defined by the current value of its variables. The state of a system (*a.k.a.* the *configuration*) is the product of the states of all processors. We also refer to the state of a processor and its neighborhood as a *local configuration*. We note Γ the set of all configurations of the system.

A *step* $\gamma \rightarrow \gamma'$ is defined as an atomic execution of a non-empty subset of enabled rules in γ that transitions the system from γ to γ'. An execution of a protocol $\mathcal{P}$ is a maximal sequence of configurations $\epsilon = \gamma_0\gamma_1 \ldots \gamma_i\gamma_{i+1} \ldots$ such that, $\forall i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$ is a step if γ_{i+1} exists (else γ_i is a *terminal* configuration). *Maximality* means that the sequence is either finite (and no action of $\mathcal{P}$ is enabled in the terminal configuration) or infinite. $\mathcal{E}$ is the set of all possible executions of $\mathcal{P}$. A processor p is *neutralized* in step $\gamma_i \rightarrow \gamma_{i+1}$ if p is enabled in γ_i and is *not* enabled in γ_{i+1}, yet did not execute any rule in step $\gamma_i \rightarrow \gamma_{i+1}$.

A *scheduler* (also called *daemon*) is a predicate over the executions. Recall that, in any execution, each step $\gamma \longrightarrow \gamma'$ results from a *non-empty* subset of enabled processors *atomically executing* a rule. This subset is chosen by the scheduler. A scheduler is *central* if it chooses *exactly one* enabled processor in any particular step, it is *distributed* if it chooses *at least one* enabled processor, and *locally central* if it chooses *at least one* enabled processor yet ensures that no two neighboring processors are chosen concurrently. A scheduler is *synchronous* if it chooses *every* enabled processor in every step. A scheduler is *asynchronous* if it is either central, distributed or locally central. A scheduler may also have some *fairness* properties. A scheduler is *strongly fair* (the strongest fairness assumption for asynchronous schedulers) if every processor that is enabled *infinitely often* is eventually chosen to execute a rule. A scheduler is *weakly fair* if every *continuously* enabled processor is eventually chosen to execute a rule. Finally, the *unfair* scheduler has the weakest fairness assumption: it only guarantees that at least one enabled processor is eventually chosen to execute a rule. As the strongly fair scheduler is the strongest fairness assumption, any problem that cannot be solved under this assumption cannot be solved for all weaker fairness assumptions. In contrast, any algorithm performing under the unfair scheduler also works for all stronger fairness assumptions.

Fault-containment and stabilization. In a particular execution ϵ, we distinguish the set of processors V^* that never crash in ϵ (*i.e.* the set of *correct* processors). By extension, for any part $C \subset V$, the set of correct processors in C is denoted by C^*. As crashed processors cannot be distinguished from slow ones by their neighbors, we assume that variables of crashed processors are always readable.

Let $\mathcal{P}$ be a problem to solve. A *specification* of $\mathcal{P}$ is a predicate that is satisfied by every algorithm solving the problem. We recall definitions about stabilization and fault-tolerance.

Definition 1 (*Self-Stabilization [8]*)**.** Let $\mathcal{P}$ be a problem, and $\mathcal{S}_{\mathcal{P}}$ a specification of $\mathcal{P}$. An algorithm $\mathcal{A}$ is self-stabilizing for $\mathcal{S}_{\mathcal{P}}$ if and only if for every configuration $\gamma_0 \in \Gamma$, for every execution $\epsilon = \gamma_0\gamma_1 \ldots$, there exists a finite prefix $\gamma_0\gamma_1 \ldots \gamma_l$ of ϵ such that all executions starting from γ_l satisfy $\mathcal{S}_{\mathcal{P}}$.

Definition 2 ((f, r)-*Containment [28]*)**.** Let $\mathcal{P}$ be a problem, and $\mathcal{S}_{\mathcal{P}}$ a specification of $\mathcal{P}$. A configuration $\gamma \in \Gamma$ is (f, r)-contained for specification $\mathcal{S}_{\mathcal{P}}$ if and only if, given at most f crashed processors, every execution starting from γ, always satisfies $\mathcal{S}_{\mathcal{P}}$ on the sub-graph induced by processors that are at distance r or more from any crashed processor.

Definition 3 (*Fault-Tolerant Self-Stabilization (FTSS)* [1,18])**.** Let $\mathcal{P}$ be a problem, and $\mathcal{S}_{\mathcal{P}}$ a specification of $\mathcal{P}$. An algorithm $\mathcal{A}$ is fault-tolerant and self-stabilizing with radius r for f crashed processors (and denoted by (f, r)-*FTSS*) for specification $\mathcal{S}_{\mathcal{P}}$ if and only if, given at most f crashed processors, for every configuration $\gamma_0 \in \Gamma$, for every execution $\epsilon = \gamma_0\gamma_1 \ldots$, there exists a finite prefix $\gamma_0\gamma_1 \ldots \gamma_l$ of ϵ such that γ_l is (f, r)-contained for specification $\mathcal{S}_{\mathcal{P}}$.

Unison. In the following, c_p is the variable of processor p that represents its clock value. Values are taken in the set of natural integers (that is, the number of states is unbounded, and a total order can be defined on clock values). Note that we do not consider the case of bounded clocks in this paper. We now define two notions related to local clock synchronization: the first one restricts the safety property to correct processors, while the second one considers all processors. We call *drift* between two processors p and q the absolute value of the difference between their clock values. In this paper, we deal with unison that is a weak clock synchronization: we must ensure that clocks are eventually "close" from each other. More precisely, two processors p and q are *in unison* if the drift between them is no more than 1. We say that a configuration of the system is *weakly synchronized* if any correct processor is in unison with its correct neighbors. More formally,

Definition 4 (*Weakly Synchronized Configuration*)**.** Let $\gamma \in \Gamma$. We say that γ is weakly synchronized, denoted by $\gamma \in \Gamma_1^*$, if and only if : $\forall p \in V^* \, \forall q \in N_p^* \; |c_p - c_q| \leq 1$.

We say that a configuration of the system is *uniformly weakly synchronized* if any processor is in unison with all its neighbors (even with crashed ones). More formally,

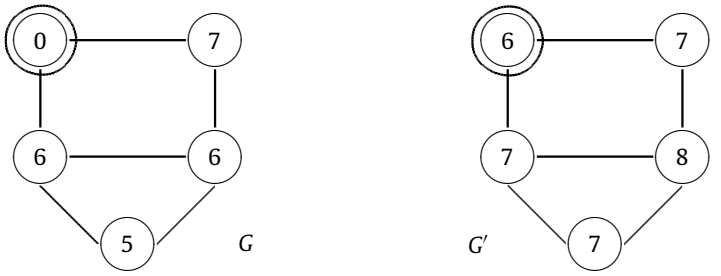

Fig. 1. Some examples of weakly synchronized configurations (the numbers represent clock values, the double circles represent crashed processors). System G is in a weakly synchronized configuration but not in a uniformly weakly synchronized configuration whereas system G' is in a uniformly weakly synchronized configuration (and hence in a weakly synchronized configuration).

Definition 5 (*Uniformly Weakly Synchronized Configuration*)**.** Let $\gamma \in \Gamma$. We say that γ is uniformly weakly synchronized, denoted by $\gamma \in \Gamma_1$, if and only if: $\forall p \in V, \forall q \in N_p,\ |c_p - c_q| \leq 1$.

Fig. 1 gives some examples of weakly synchronized configurations.

We now specify the two variants of our problem (depending whether safety property is extended to crashed processors or not). Intuitively, asynchronous unison (respectively uniform asynchronous unison) ensures that the system is eventually (and remains forever) in a weakly (respectively uniformly weakly) synchronized configuration (safety property) and that clocks of correct processors are infinitely often incremented by 1 (liveness condition). More formally,

Definition 6 (*Asynchronous Unison*)**.** Let $\gamma_0 \in \Gamma$. An execution $\epsilon = \gamma_0\gamma_1\ldots$ is a legitimate execution for asynchronous unison, denoted by **AU**, if and only if:

Safety: $\forall i \in \mathbb{N}, \gamma_i \in \Gamma_1^*$.

Liveness: Each processor $p \in V^*$ increments its clock (by 1) infinitely often in ϵ.

Definition 7 (*Uniform Asynchronous Unison*)**.** Let $\gamma_0 \in \Gamma$. An execution $\epsilon = \gamma_0\gamma_1\ldots$ is a legitimate execution for uniform asynchronous unison, denoted by **UAU**, if and only if:

Safety: $\forall i \in \mathbb{N}, \gamma_i \in \Gamma_1$.

Liveness: Each processor $p \in V^*$ increments its clock (by 1) infinitely often in ϵ.

Note that an algorithm that complies to specification of **UAU** also complies to that of **AU** (the converse is not true) since $\Gamma_1 \subseteq \Gamma_1^*$ (if no processor is crashed, we have: $\Gamma_1 = \Gamma_1^*$, but if at least one processor is crashed, we have: $\Gamma_1 \subsetneq \Gamma_1^*$). Note also that these two specifications do not forbid *decrementing* clocks. Our specification generalizes the classical unison specification [5] as any solution to the former is also a solution of ours. Unison protocols that are useful in a distributed setting are those that do not know the underlying communication graph. We refer to *universal* protocols to denote the fact that a protocol that can perform on every communication graph that matches a particular predicate (*e.g.* every graph of degree less than two). To disprove universality of a protocol, it is thus sufficient to exhibit a particular communication graph in its acceptance predicate such that at least one possible execution does not satisfy the specification.

We now present two key properties satisfied by all known self-stabilizing unison protocols. Those properties are used in the impossibility results presented in Section 3. We called these properties respectively *minimality* and *priority*.

Minimality means that nodes maintain no extra variables but the digital clock value. This implies that the code of a minimal unison can only refer to clocks or to predefined constants. We now state the formal definition of this property.

Definition 8 (*Minimality*)**.** A unison is *minimal* if and only if every processor only maintains a clock variable.

Priority means that if, for a given processor, incrementing the clock value does not break the local safety predicate with its neighbors, then its clock value is actually incremented in a finite number of activations, even if no neighbor modifies its clock value. This property implies that, if a processor can increment its clock without breaking unison with its neighbors, then it does so in finite time whether its neighbors are crashed or not. This property is similar to obstruction freedom in the sense that the protocol only has very weak constraints about progress. We formally state this property in the following definition.

Definition 9 (*Priority*)**.** A unison is *priority* if and only if it satisfies the following property: if there exists a processor p such that $\forall q \in N_p$, ($c_q = c_p$ or $c_q = c_p + 1$) in a configuration γ_i, then there exists a fragment of execution $\epsilon = \gamma_i \ldots \gamma_{i+k}$ such that:

- only p is chosen by the scheduler during ϵ.
- c_p is not modified during $\gamma_{i+j} \longrightarrow \gamma_{i+j+1}$, for $j \in \{0, \ldots, k-2\}$.
- c_p is incremented during $\gamma_{i+k-1} \longrightarrow \gamma_{i+k}$.

For example, protocols proposed by [3–5,20] fall in the category of minimal and priority unison using these definitions. Another example is the protocol of [29] that is priority but not minimal. To our knowledge, any existing unison protocol satisfies either minimality or priority.

3. Impossibility results

In this section we present a broad class of impossibility results related to the FTSS unison. First, we show a preliminary result that states that a processor cannot modify its clock value if it has two neighbors q and q' with $c_q = c_p - 1$ and $c_{q'} = c_p + 1$ (Lemma 1). This property is further used in the sequel of this section. Proposition 1 proves that there exists no (f, r)-FTSS algorithm for any r value if $f \geq 2$. Furthermore, in Proposition 2, we prove that there exists no $(1, r)$-FTSS algorithm for **AU** under an unfair daemon for any r value. Then we study the minimal and priority asynchronous unison and prove there exists no $(1, r)$-FTSS algorithm for minimal or priority **AU** under a weakly fair daemon for any r value (Lemma 2, Propositions 3 and 4). Finally, we prove there exists no $(1, r)$-FTSS algorithm for minimal or priority **AU** under a strongly fair daemon for any r value if the network has a maximal degree of at least 3 (Lemma 3, Propositions 5 and 6). In the following we assume, for the sake of generality, the most constrained scheduler (the central one).

3.1. Preliminaries

First, we introduce a preliminary result that shows that in any execution of a universal (f, r)-ftss algorithm for **AU** (under an asynchronous daemon) a processor cannot modify its clock value if it has two neighbors q and q' such that: $c_q = c_p - 1$ and $c_{q'} = c_p + 1$.

Lemma 1. *Let $\mathcal{A}$ be a universal (f, r)-ftss algorithm for **AU** (under an asynchronous daemon). Let γ be a configuration where a processor p (such that $c_p \geq 1$) has two neighbors q and q' such that: $c_q = c_p - 1$ and $c_{q'} = c_p + 1$. If p executes an action of $\mathcal{A}$ during the step $\gamma \longrightarrow \gamma'$, then this action does not modify the value of c_p. If $\mathcal{A}$ is also minimal, then the processor p is not enabled for $\mathcal{A}$ in γ.*

Proof. Let $\mathcal{A}$ be a universal (f, r)-ftss algorithm for **AU** (under an asynchronous daemon). Let G be a network and γ be a configuration of G such that no processor is crashed, $\gamma \in \Gamma_1$ and there exists a processor p (such that $c_p \geq 1$) that has two neighbors q and q' such that: $c_q = c_p - 1$ and $c_{q'} = c_p + 1$.

Assume p executes an action of $\mathcal{A}$ during the step $\gamma \longrightarrow \gamma'$ (and only p) such that this action modifies the value of c_p. Note that c_q and $c_{q'}$ are identical in γ and γ'. Let α be the value of c_p in γ and α' be the value of c_p in γ'. Values of α and α' satisfy one of the two following relations:

Case 1: $\alpha < \alpha'$.
This implies that $|\alpha' - c_q| = |\alpha' - \alpha| + |\alpha - c_q| > 1$ (since $|\alpha' - \alpha| \geq 1$ by hypothesis and $|\alpha - c_q| = 1$).

Case 2: $\alpha' < \alpha$.
This implies that $|\alpha' - c_{q'}| = |\alpha' - \alpha| + |\alpha - c_{q'}| > 1$ (since $|\alpha' - \alpha| \geq 1$ by hypothesis and $|\alpha - c_{q'}| = 1$).

In the two above cases, $\gamma' \notin \Gamma_1$, hence the safety property of $\mathcal{A}$ is not satisfied.
If $\mathcal{A}$ is also minimal, then the previous result implies that p is not enabled for $\mathcal{A}$ in γ. □

3.2. Impossibility result due to the number of crashed processors

Proposition 1. *For any natural number r, there exists no universal (f, r)-ftss algorithm for **AU** under an asynchronous daemon if $f \geq 2$.*

Proof. Let r be a natural number. Let $\mathcal{A}$ be a universal $(2, r)$-ftss algorithm for **AU** (under an asynchronous daemon). Consider a network represented by the following graph: $G = (V, E)$ with $V = \{p_0, \ldots, p_{2(r+1)}\}$ and $E = \{\{p_i, p_{i+1}\} | i \in \{0, \ldots, 2r+1\}\}$. Let γ be the following configuration of the network: p_0 and $p_{2(r+1)}$ are crashed and $\forall i \in \{0, \ldots, 2(r+1)\}$, $c_{p_i} = i$ (all the other variables may have any value).

By Lemma 1, no processor between p_2 and p_{2r+1} can change its clock value in every execution starting from γ. This contradicts the definition of $\mathcal{A}$. Indeed, p_{r+1} must eventually satisfy the specification of **AU** since the closest crashed processor is at r hops away. In particular, any execution starting from γ must contain a suffix where the clock of p_{r+1} is infinitely often incremented. This contradiction shows us the result. □

3.3. Impossibility result due to unfair daemon

Proposition 2. *For any natural number r, there exists no universal $(1, r)$-ftss algorithm for **AU** under an unfair daemon.*

Proof. Let r be a natural number. Assume that there exists a universal $(1, r)$-ftss algorithm $\mathcal{A}$ for **AU** under an unfair daemon. Consider a network G, of diameter greater than $2r + 2$ (note that in this case, at least one processor must eventually satisfy the specification of the **AU** problem). Let p be a processor of G. Since the daemon is unfair, it can choose to never activate p in an execution ϵ unless this processor becomes the only enabled processor of G in a configuration of ϵ by definition.

For the sake of contradiction, assume that there exists a configuration γ such that no processor is crashed and where p is the only enabled processor of the network. Denote by γ' the same configuration when p is crashed. Note that the set of enabled processors is identical in γ and γ' by construction. As we assumed that only p is enabled in γ, this implies that no

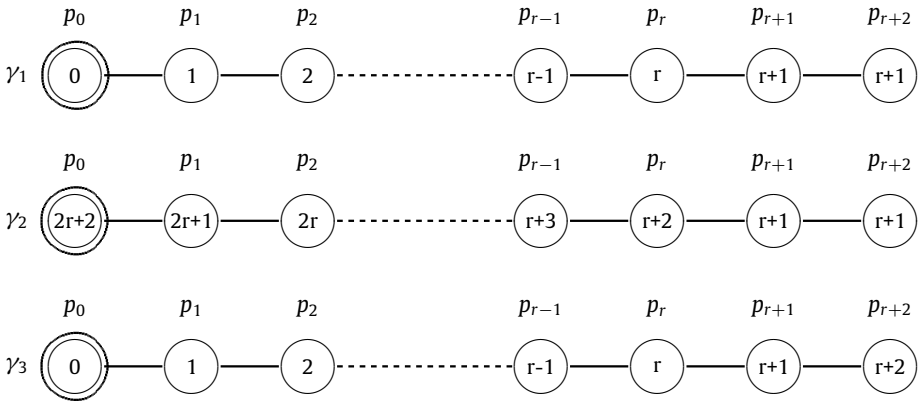

Fig. 2. The three configurations used in the proof of Lemma 2 (the numbers represent clock values and the double circles represent crashed processors).

correct processor is enabled in γ'. Hence, the system is deadlocked in γ' and the specification of **AU** is not satisfied since no clock of correct processor can be updated. This contradiction implies that, for any configuration where no processor is crashed, at least two processors are enabled.

Since there exists no configuration where p is the unique enabled processor (in every execution starting from an arbitrary configuration), the unfair daemon can starve p infinitely (if no crash occurs). This contradicts the liveness property of $\mathcal{A}$ since p cannot update its clock in this execution. □

3.4. Impossibility results due to weakly fair daemon

In this section we prove there exists no universal $(1, r)$-ftss algorithm for minimal or priority **AU** under a weakly fair daemon for any r value.

The first impossibility result uses the following property: if there exists a universal algorithm $\mathcal{A}$ that is $(1, r)$-ftss for minimal **AU** under a weakly fair daemon for a natural number r, then an arbitrary processor p is not enabled for $\mathcal{A}$ if it has only one neighbor p' and if $c_p = c_{p'}$ (proved in Lemma 2 formally stated below). Then, we show that $\mathcal{A}$ starves the network reduced to a two-correct-processor chain where all clock values are identical (see Proposition 3).

Lemma 2. *If there exists a universal algorithm $\mathcal{A}$ that is $(1, r)$-ftss for minimal **AU** under a weakly fair daemon for a natural number r, then an arbitrary processor p is not enabled for $\mathcal{A}$ if it has only one neighbor p' and if $c_p = c_{p'}$.*

Proof. Let r be a natural number. Let $\mathcal{A}$ be a universal $(1, r)$-ftss algorithm for the minimal **AU** under a weakly fair daemon.

Let G be the network reduced to a chain of length $r + 2$. Assume processors in G labeled as follows: $p_0, p_1, \ldots, p_{r+2}$. Consider the following configurations of G (see Fig. 2):

- γ_1 defined by $\forall i \in \{0, \ldots, r + 1\}$, $c_{p_i} = i$ and $c_{p_{r+2}} = r + 1$ and p_0 crashed.
- γ_2 defined by $\forall i \in \{0, \ldots, r + 1\}$, $c_{p_i} = 2r + 2 - i$ and $c_{p_{r+2}} = r + 1$ and p_0 crashed.
- γ_3 defined by $\forall i \in \{0, \ldots, r + 2\}$, $c_{p_i} = i$ and p_0 crashed.

By Lemma 1, processors from p_1 to p_r are not enabled in such configurations (and remain not enabled until one of the processors within $p_0 \ldots p_{r+1}$ executes a rule).

Note that for the processor p_{r+2}, the configurations γ_1 and γ_2 are indistinguishable (otherwise the unison would not be minimal). We are going to prove the result by contradiction. Assume p_{r+2} is enabled in γ_1 and γ_2. The safety property of $\mathcal{A}$ implies that the enabled rule for p_{r+2} modifies its clock either to $r + 2$ or to r. In the following we discuss these cases separately:

Case 1: The enabled rule for p_{r+2} modifies its clock into $r + 2$.
Assume without loss of generality that p_{r+2} is the only activated processor. Hence its clock takes the value $r + 2$. The following cases are possible in the obtained configuration:

Case 1.1: p_{r+2} is not enabled.
If an execution started from γ_1, then no processor is enabled, which contradicts the liveness property of **AU**.

Case 1.2: p_{r+2} is enabled and the enabled rule modifies its clock into $r + 1$.
Let ϵ be an execution starting from γ_1 where only p_{r+2} is activated. Consequently, the clock of the processor p_{r+2} takes infinitely the following sequence of values: $r + 1, r + 2$. In this execution, p_{r+2} executes infinitely often while processors from p_0 to p_r are never enabled. Note that p_{r+1} is not enabled when $c_{p_{r+2}} = r + 2$, hence this processor is never infinitely enabled. In conclusion, this execution is allowed by the weakly fair scheduler. Note that this execution starves p_{r+1}, which contradicts the liveness property of $\mathcal{A}$.

Case 1.3: p_{r+2} is enabled and the enabled rule modifies its clock into r.
The execution of this rule leads to Case 2.

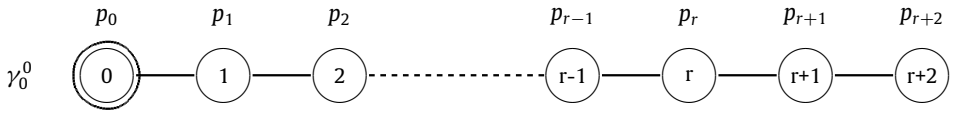

Fig. 3. Initial configuration used in the proof of Proposition 4 (the numbers represent clock values and the double circles represent crashed processors).

Case 2: The enabled rule for p_{r+2} modifies its clock into r.
Assume without loss of generality that p_{r+2} is the only activated processor and after its execution the new configuration satisfies one of the following cases:

Case 2.1: p_{r+2} is not enabled.
If an execution started from γ_2, then no processor is enabled, which contradicts the liveness property (the network is starved).

Case 2.2: p_{r+2} is enabled and the enabled rule modifies its clock into $r+1$.
Let ϵ be an execution starting from γ_2 that contains only actions of p_{r+2} (its clock takes infinitely the following value sequence : $r+1, r$). In this execution, p_{r+2} executes a rule infinitely often (by construction) and processors from p_0 to p_r are never enabled. Note that p_{r+1} is not enabled when $c_{p_{r+2}} = r$, so this processor is never infinitely enabled. In conclusion, this execution satisfies the weakly fair scheduling.
Note that this execution starves p_{r+1}, which contradicts the liveness property of $\mathcal{A}$.

Case 2.3: p_{r+2} is enabled and the enabled rule modifies its clock into $r+2$.
The execution of these rule leads to Case 1.

Overall, the only two possible cases (Cases 1.3 and 2.3) are the following:

1. p_{r+2} is enabled for modifying its clock value into r when $c_{p_{r+2}} = r+2$ and $c_{p_{r+1}} = r+1$.
2. p_{r+2} is enabled for modifying its clock value into $r+2$ when $c_{p_{r+2}} = r$ and $c_{p_{r+1}} = r+1$.

Let ϵ be an execution starting from γ_3 that contains only actions of p_{r+2} (its clock takes infinitely the following sequence of values: $r+2, r$). In this execution, p_{r+2} executes a rule infinitely often (by construction) and processors in $p_0 \dots p_r$ are never enabled. Note that p_{r+1} is not enabled when $c_{p_{r+2}} = r+2$, so this processor is never infinitely enabled. In conclusion, this execution satisfies the weakly fair scheduling.
This execution starves p_{r+1}, which contradicts the liveness property of $\mathcal{A}$ and proves the result. □

Proposition 3. *For any natural number r, there exists no universal $(1, r)$-ftss algorithm for* minimal ***AU*** *under a weakly fair daemon.*

Proof. Let r be a natural integer. Assume there exists a universal $(1, r)$-ftss algorithm $\mathcal{A}$ for the minimal **AU** under a weakly fair daemon. By Lemma 2, an arbitrary processor p is not enabled for $\mathcal{A}$ if it has only one neighbor p' and if $c_p = c_{p'}$.
Let G be a network reduced to a chain of 2 processors p and p'. Let γ be a configuration of G where $c_p = c_{p'}$ with no crashed processor. Notice that no processor is enabled in γ that contradicts the liveness property of $\mathcal{A}$ and proves the result. □

The second main result of this section is that there exists no universal $(1, r)$-ftss algorithm for priority **AU** under a weakly fair daemon for any natural number r (see Proposition 4).
We prove this result by contradiction. We construct an execution starting from the configuration γ_0^0 shown in Fig. 3 allowed by a weakly fair scheduler. We prove that this execution starves p_{r+1} that contradicts the liveness property of the algorithm.

Proposition 4. *For any natural number r, there exists no universal $(1, r)$-ftss algorithm for* priority ***AU*** *under a weakly fair daemon.*

Proof. Let r be a natural number. Assume that there exists a universal $(1, r)$-ftss algorithm $\mathcal{A}$ for priority **AU** under a weakly fair daemon. Let G be the network reduced to a chain of length $r+2$. Assume that processors in G are labeled as follows: $p_0, p_1, \dots, p_{r+2}$. Let γ_0^0 be a configuration such that p_0 is crashed and $\forall i \in \{0, \dots, r+2\}$, $c_{p_i} = i$ (see Fig. 3). Note that all the other variables may have any value.
We construct a fragment of execution $\epsilon_0' = \gamma_0^0 \gamma_1^0 \gamma_2^0 \dots \gamma_{r+1}^0$ starting from γ_0^0 such that $\forall i \in \{0, 1, \dots, r\}$, the step $\gamma_i^0 \to \gamma_{i+1}^0$ contains only an action of p_{i+1} if p_{i+1} is enabled. By Lemma 1, this fragment does not modify the clock value of any processor in $\{p_0 \dots p_{r+1}\}$.
We also construct a fragment of execution, ϵ_0'', starting from γ_{r+1}^0 using the following cases:

Case 1: p_{r+2} is not enabled in γ_{r+1}^0.
Let ϵ_0'' be ϵ (empty word).

Case 2: p_{r+2} is enabled in γ_{r+1}^0.
We distinguish now the following sub-cases:

Case 2.1: There exists a rule of p_{r+2} enabled in γ_{r+1}^0 that does not modify the clock value of p_{r+2}.
Let ϵ_0'' be $\gamma_{r+1}^0 \gamma_{r+2}^0$ where step $\gamma_{r+1}^0 \to \gamma_{r+2}^0$ contains only the execution of this rule by p_{r+2}.

Case 2.2: Any enabled rule of p_{r+2} in γ^0_{r+1} modifies its clock value.

Note that the safety property of $\mathcal{A}$ implies that the clock of p_{r+2} takes the value r or $r+1$. Let us study the following cases.

Case 2.2.1: There exists a rule of p_{r+2} enabled in γ^0_{r+1} that modifies its clock value into $r+1$.

Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon''_0 = \gamma^0_{r+1}\gamma^0_{r+2}\ldots\gamma^0_{r+k}$ that contains only actions of p_{r+2} such that (i) p_{r+2} executes one of the rules that modifies its clock value into $r+1$ in the step $\gamma^0_{r+1} \to \gamma^0_{r+2}$ (ii) in the steps from γ^0_{r+2} to γ^0_{r+k-1} the clock value of p_{r+2} is not modified while (iii) in the step $\gamma^0_{r+k-1} \to \gamma^0_{r+k}$ the clock value of p_{r+2} is incremented.

Case 2.2.2: Any enabled rule of p_{r+2} in γ^0_{r+1} modifies its clock value into r.

Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon_a = \gamma^0_{r+1}\gamma^0_{r+2}\ldots\gamma^0_{r+k}$ that contains only actions of p_{r+2} such that (i) p_{r+2} executes one of the rules that modifies its clock value into r in the step $\gamma^0_{r+1} \to \gamma^0_{r+2}$ (ii) in the steps from γ^0_{r+2} to γ^0_{r+k-1} the clock value of p_{r+2} is not modified and (iii) in the step $\gamma^0_{r+k-1} \to \gamma^0_{r+k}$ the clock of p_{r+2} takes the value $r+1$.

Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon_b = \gamma^0_{r+k}\gamma^0_{r+k+1}\ldots\gamma^0_{r+j}$ that contains only actions of p_{r+2} such that (i) in the steps from γ^0_{r+k} to γ^0_{r+j-1} the clock value of p_{r+2} is not modified and (ii) in the step $\gamma^0_{r+j-1} \to \gamma^0_{r+j}$ the clock value of p_{r+2} is incremented.

Let ϵ''_0 be $\epsilon_a\epsilon_b$.

In all cases, we construct a fragment of execution $\epsilon_0 = \epsilon'_0\epsilon''_0$ such that its last configuration (let us denote it by γ^1_0) satisfies: the value of any clock is identical to the one in γ^0_0 (the others variables may have changed). Then, we can reiterate the reasoning and obtain a fragment of execution $\epsilon_1, \epsilon_2 \ldots$ (respectively starting from $\gamma^1_0, \gamma^2_0, \ldots$) that satisfies the same property.

We finally obtain an execution $\epsilon = \epsilon_0\epsilon_1\ldots$ that satisfies:

- No processor is infinitely enabled without executing a rule (since all enabled processors in γ^i_0 execute a rule or are neutralized during ϵ_i). Consequently ϵ is an execution that satisfies the weakly fair scheduling.
- The clock of processor p_{r+1} never changes (whereas $d(p_0, p_{r+1}) = r+1$).

This execution contradicts the liveness property of $\mathcal{A}$ that is a $(1, r)$-ftss algorithm for priority **AU** under a weakly fair daemon by hypothesis. □

3.5. Impossibility results due to strongly fair daemon

In this section we prove that there exists no universal $(1, r)$-ftss algorithm for minimal or priority **AU** under a strongly fair daemon if the degree of the network is at least 3.

In order to prove the first impossibility result, we use the following property: if a processor p has only one neighbor q such that $c_q = r+1$ and if $|c_p - c_q| \leq 1$, then p is enabled in any universal $(1, r)$-ftss algorithm for minimal **AU** (see Lemma 3). Then we construct a strongly fair infinite execution that starves a processor such that the closest crashed processor is at more than r hops away. This execution contradicts the liveness property of the **AU** problem (see Proposition 5).

Lemma 3. *Let $\mathcal{A}$ be a universal $(1, r)$-ftss algorithm for minimal **AU**. If a processor p has only one neighbor q such that $c_q = r+1$ and if $|c_p - c_q| \leq 1$, then p is enabled in $\mathcal{A}$.*

Proof. Assume that there exists a universal algorithm $\mathcal{A}$ that is $(1, r)$-ftss for minimal **AU**. Let G be a network that executes $\mathcal{A}$ and that contains at least one processor p that has only one neighbor q. Assume that $c_q = r+1$ and $|c_p - c_q| \leq 1$. Then, we have:

1. If $c_p = r$, then p is enabled for at least one rule of $\mathcal{A}$. Otherwise, all processors are starved in the network reduced to the chain $p_0, \ldots, p_r, q, p$ in the configuration γ_1 defined by $\forall i \in \{0, \ldots, r\}$, $c_{p_i} = 2r+2-i$, $c_q = r+1$, $c_p = r$ where p_0 is crashed (see Fig. 4) since no correct processor is enabled (by Lemma 1).
2. If $c_p = r+1$, then p is enabled for at least one rule of $\mathcal{A}$. Otherwise, all processors are starved in the network reduced to the chain q, p in the configuration γ_2 defined by $c_q = c_p = r+1$ and where no processor is crashed (see Fig. 4). Indeed, the symmetry of the configuration implies that q is enabled if and only if p is enabled.
3. If $c_p = r+2$, then p is enabled for at least one rule of $\mathcal{A}$. Otherwise, all processors are starved in the network reduced to the chain $p_0, \ldots, p_r, q, p$ in the configuration γ_3 defined by $\forall i \in \{0, \ldots, r\}$, $c_{p_i} i$, $c_q = r+1$, $c_p = r+2$ and p_0 crashed (see Fig. 4) since no correct processor is enabled (by Lemma 1). □

Proposition 5. *For any natural number r, there exists no universal $(1, r)$-ftss algorithm for* minimal ***AU*** *under a strongly fair daemon if the system has a maximal degree of at least* 3.

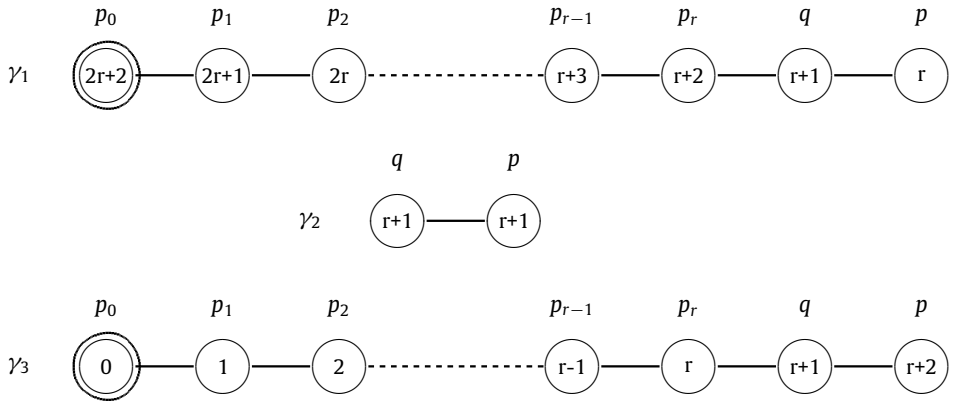

Fig. 4. The three configurations used in the proof of Lemma 3 (the numbers represent clock values and the double circles represent crashed processors).

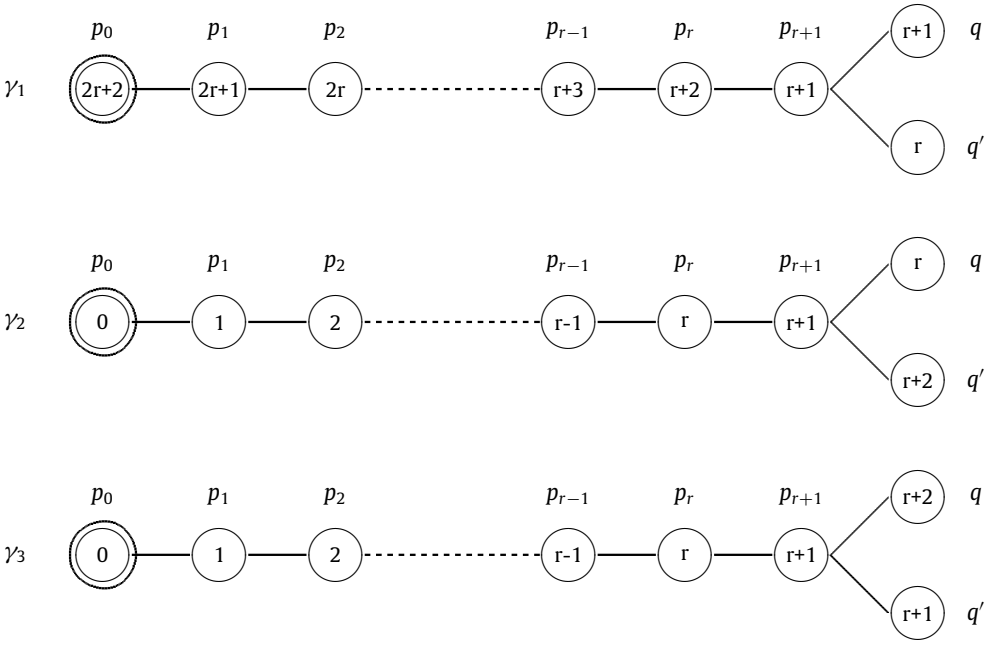

Fig. 5. The three configurations used in the proof of Proposition 5 (the numbers represent clock values and the double circles represent crashed processors).

Proof. Let r be a natural number. Assume that there exists a universal $(1, r)$-ftss algorithm $\mathcal{A}$ for the minimal **AU** under a strongly fair daemon in a network with a degree of at least 3. Let G be the network defined by: $V = \{p_0, \ldots, p_{r+1}, q, q'\}$ and $E = \{\{p_i, p_{i+1}\}, i \in \{0, \ldots, r\}\} \cup \{\{p_{r+1}, q\}, \{p_{r+1}, q'\}\}$.

As $\mathcal{A}$ is deterministic and the system anonymous, q and q' must behave identically if they have the same clock value (in this case, their local configurations are identical). If $c_{p_{r+1}} = r + 1$ and $|c_{p_{r+1}} - c_q| \leq 1$, there exists three local configurations for q: (1) $c_q = r$, (2) $c_q = r + 1$ or (3) $c_q = r + 2$ (the same property holds for q').

By Lemma 3, processor q (respectively q') is enabled in any configuration where $c_{p_{r+1}} = r + 1$ and $|c_{p_{r+1}} - c_q| \leq 1$ (respectively $|c_{p_{r+1}} - c_{q'}| \leq 1$). Moreover, in this case, the enabled rule for q (respectively q') modifies its clock into a value in $\{r, r + 1, r + 2\} - \{c_q\}$ (respectively $\{r, r + 1, r + 2\} - \{c_{q'}\}$) by the safety property of $\mathcal{A}$.

For each of the three possible local configurations for q or q' (studied in the proof of Lemma 3), $\mathcal{A}$ can only allow 2 moves. Hence, there exists 8 possible moves for $\mathcal{A}$. Let us denote each of these possibilities by a triplet (a, b, c) where a, b and c are the clock value of q after the allowed move when $c_q = r$, $c_q = r + 1$, and $c_q = r + 2$ respectively. Note that, due to the determinism of $\mathcal{A}$, moves allowed for q' and q are identical. There exists the following cases:

Case 1: $(r + 1, r, r)$

Let γ_1 be the configuration of G defined by: $\forall i \in \{0, \ldots, r + 1\}$, $c_{p_i} = 2r + 2 - i$, $c_q = r + 1$ and $c_{q'} = r$ and p_0 crashed (see Fig. 5). Note that only q and q' are enabled (by Lemma 1). Assume q executes. Hence, its clock takes the value r. By Lemma 1, only q and q' are enabled. Assume now that q' executes. Its clock takes the value $r + 1$.

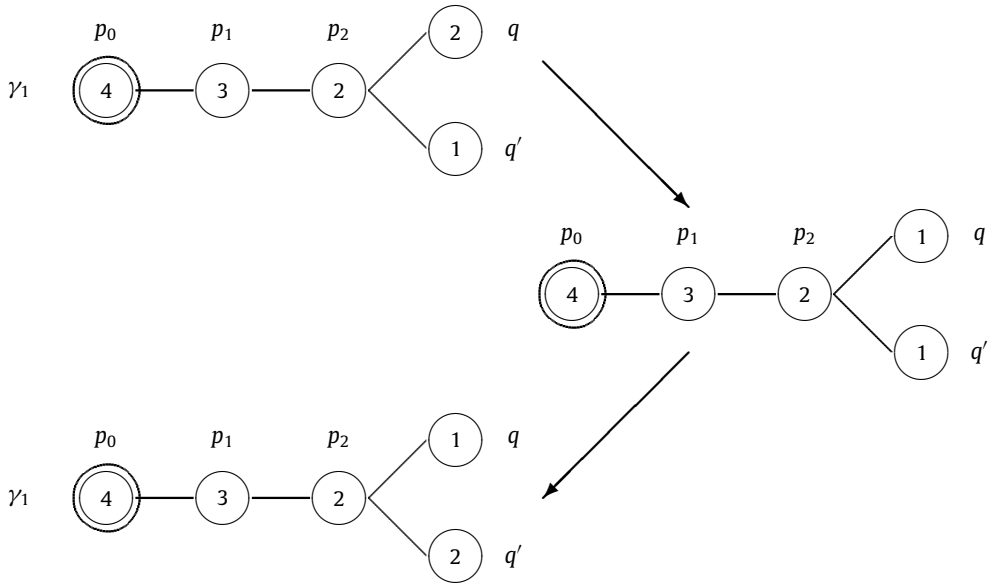

Fig. 6. Example of the execution constructed in Case 1 of Proposition 5 when $r = 1$ (the numbers represent clock values and the double circles represent crashed processors).

This configuration is identical to γ_1 (since processors are anonymous), we can repeat the above reasoning in order to obtain an infinite execution where processors $p_1, \ldots, p_{r+1}$ are never enabled (see Fig. 6 for an illustration when $r = 1$).

Case 2: $(r + 1, r + 2, r)$
Let γ_2 be the configuration of G defined by: $\forall i \in \{0, \ldots, r+1\}$, $c_{p_i} i, c_q = r$ and $c_{q'} = r+2$ and p_0 crashed (see Fig. 5). Note that only q and q' are enabled (by Lemma 1). Assume q executes. Its clock takes the value $r + 1$. By Lemma 1, only q and q' are enabled. Assume q executes its rule again. Its clock takes the value $r + 2$. By Lemma 1, only q and q' are enabled. Assume now that q' executes its rule. Its clock takes the value r. This configuration is identical to γ_2 (since processors are anonymous). We can repeat the reasoning in order to obtain an infinite execution where processors in $p_1, \ldots, p_{r+1}$ are never enabled.

Case 3: $(r + 1, r, r + 1)$
Similar to the reasoning of Case 1.

Case 4: $(r + 1, r + 2, r + 1)$
Let γ_3 be the configuration of G defined by: $\forall i \in \{0, \ldots, r + 1\}$, $c_{p_i} = i, c_q = r + 2$ and $c_{q'} = r + 1$ and where p_0 is crashed (see Fig. 5). Note that only q and q' are enabled (by Lemma 1). Assume q' executes its rule. Its clock takes the value $r + 2$. By Lemma 1, only q and q' are enabled. Assume now that q executes its rule. Its clock takes the value $r + 1$. This configuration is identical to γ_3 (since processors are anonymous). We can repeat the reasoning in order to obtain an infinite execution where processors in $p_1, \ldots, p_{r+1}$ are never enabled.

Case 5: $(r + 2, r, r)$
Let γ_2 be the configuration of G as defined in the Case 2 above. Note that only q and q' are enabled (by Lemma 1). Assume q executes its rule. Its clock takes the value $r+2$. By Lemma 1, only q and q' are enabled. Assume now that q' executes its rule. Its clock takes the value r. This configuration is identical to γ_2 (since processors are anonymous). We can repeat the reasoning in order to obtain an infinite execution where processors $p_1, \ldots, p_{r+1}$ are never enabled.

Case 6: $(r + 2, r + 2, r)$
The reasoning is similar to the Case 5.

Case 7: $(r + 2, r, r + 1)$
Let γ_2 be the configuration of G as defined in the Case 2 above. Note that only q and q' are enabled (by Lemma 1). Assume q executes its rule. Its clock takes the value $r+2$. By Lemma 1, only q and q' are enabled. Assume q' executes its rule. Its clock takes the value $r + 1$. By Lemma 1, only q and q' are enabled. Assume q' executes again its rule. Its clock takes the value r. This configuration is identical to γ_2 (since processors are anonymous). We can repeat the above scenario in order to obtain an infinite execution where processors $p_1, \ldots, p_{r+1}$ are never enabled.

Case 8: $(r + 2, r + 2, r + 1)$
The proof is similar to the Case 4.

Overall, we can construct an infinite execution where processor p_0 is crashed, processors from p_1 to p_{r+1} are never enabled and processors q and q' execute a rule infinitely often. This execution satisfies the strongly fair scheduling. Notice that in this execution p_{r+1} is never enabled, hence it is starved. This contradicts the liveness property of $\mathcal{A}$ and proves the result. □

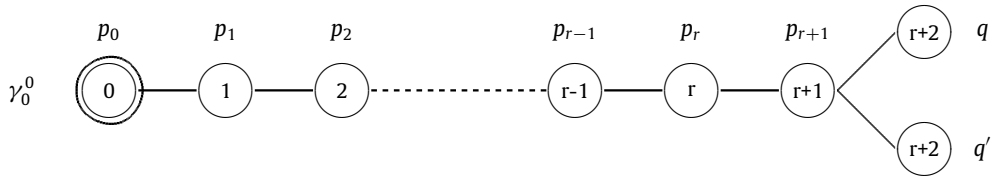

Fig. 7. The initial configuration for the proof of Proposition 6 (the numbers represent clock values and the double circles represent crashed processors).

The second main result of this section is that there exists no universal $(1, r)$-ftss algorithm for priority **AU** under a strongly fair daemon for any natural number r if the degree of the graph modeling the network is at least 3. (see Proposition 6).

We prove this result by contradiction. We construct an execution starting from the configuration γ_0^0 of Fig. 7 satisfying the strongly fair scheduling that starves p_{r+1}, which contradicts the liveness of the algorithm.

Proposition 6. *For any natural number r, there exists no universal* $(1, r)$*-ftss algorithm for* priority ***AU*** *under a strongly fair daemon if the system has a maximal degree of at least* 3.

Proof. Let r be a natural number. Assume that there exists a universal $(1, r)$-ftss algorithm $\mathcal{A}$ for priority **AU** under a strongly fair daemon even if the graph modeling the network has a degree of at least 3. Let G be the network defined by: $V = \{p_0, \ldots, p_{r+1}, q, q'\}$ and $E = \{\{p_i, p_{i+1}\}, i \in \{0, \ldots, r\}\} \cup \{\{p_{r+1}, q\}, \{p_{r+1}, q'\}\}$. Note that G has a degree equal to 3.

Let γ_0^0 be the following configuration: $\forall i \in \{0, \ldots, r+1\}, c_{p_i} = i, c_q = c_{q'} = r+2$ and p_0 crashed (see Fig. 7). Note that, for any execution ϵ starting from γ_0^0, one of the processors q and q' must be enabled to modify its clock in a finite time (otherwise the network would be starved following Lemma 1). This implies the existence of a fragment of execution $\epsilon_a^0 = \gamma_0^0\gamma_1^0 \ldots \gamma_k^0$ with the following properties:

1. $k \geq 1$ if there exists $i \in \{0, \ldots, r+1\}$ such that p_i is enabled in γ_0^0, $k = 0$ otherwise;
2. ϵ_a^0 contains no modification of clock values;
3. γ_k^0 is the first configuration where q or q' is enabled to modify its clock value.

Assume now that the scheduling of ϵ_a^0 satisfies the following property: at each step, the daemon chooses the processor that was last activated among enabled processors. Note that this scenario is compatible with a strongly fair scheduling.

Let us study the following cases:

Case 1: q is enabled in γ_k^0 for a modification of its clock value. The safety property of $\mathcal{A}$ implies that the value of c_q should be modified either to r or to $r+1$.

Case 1.1: The value of c_q is modified to r.

Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon_{b1}^0 = \gamma_k^0\gamma_{k+1}^0 \cdots \gamma_{k+r}^0$ that contains only actions of q such that (i) in the steps from γ_k^0 to γ_{k+r-1}^0 the clock value of q is not modified and (ii) in the step $\gamma_{k+r-1}^0 \to \gamma_{k+r}^0$ the clock value of q is incremented.

Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon_{b2}^0 = \gamma_{k+r}^0\gamma_{k+r+1}^0 \cdots \gamma_{k+j}^0$ that contains only executions of a rule by q such that (i) in the steps from γ_{k+r}^0 to γ_{k+j-1}^0 the clock value of q is not modified and (ii) in the step $\gamma_{k+j-1}^0 \to \gamma_{k+j}^0$ the clock value of q is incremented.

Let ϵ_b^0 be $\epsilon_{b1}^0\epsilon_{b2}^0$.

Case 1.2: The value of c_q is modified to $r+1$.

Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon_b^0 = \gamma_k^0\gamma_{k+1}^0 \ldots \gamma_{k+r}^0$ that contains only actions of q such that (i) in the steps from γ_k^0 to γ_{k+r-1}^0 the clock value of q is not modified and (ii) in the step $\gamma_{k+r-1}^0 \to \gamma_{k+r}^0$ the clock value of q increments.

If q' is enabled in the last configuration of ϵ_b^0,[1] we can construct ϵ_c^0 similarly to ϵ_b^0 using processor q'. Otherwise, let ϵ_c^0 be ϵ (the empty word).

Case 2: q' is enabled in γ_k^0 for a modification of its clock value.

We can construct ϵ_b^0 and ϵ_c^0 similar to the Case 1 by reversing the roles of q and q'.

Let us define $\epsilon^0 = \epsilon_a^0\epsilon_b^0\epsilon_c^0$. Notice that the clock values are identical in the first and the last configuration of ϵ^0. This implies that we can infinitely repeat the previous reasoning in order to obtain an infinite execution $\epsilon = \epsilon^0\epsilon^1 \ldots$ that satisfies:

- No correct processor is infinitely often enabled without executing a rule (since q and q' execute a rule infinitely often and others processors are chosen in function of their last execution of a rule, which implies that an infinitely often enabled processor executes a rule in a finite time). This execution satisfies a strongly fair scheduling.
- The clock value of p_{r+1} is never modified (whereas $d(p_0, p_{r+1}) = r+1$).

This execution contradicts the liveness property of $\mathcal{A}$, which implies the result. □

[1] In this case, q' was already enabled in the last configuration of ϵ_a^0.

Algorithm 1 ($\mathcal{UFTSS}$): universal (1, 0)-FTSS **AU** for chains and rings.

Data:
- N_p: set of neighbors of p.

Variable:
- c_p: natural integer representing the clock of the processor.

Macros:
- For $A \subseteq \mathbb{N}$ and $a \in \mathbb{N}$, $next(A, a) = \begin{cases} a+1 \text{ if } a+1 \in A \\ min\{A\} \text{ otherwise.} \end{cases}$
- For $q \in N_p$, $poss(q) = \begin{cases} \{c_q - 1, c_q, c_q + 1\} \text{ if } c_q \neq 0 \\ \{c_q, c_q + 1\} \text{ otherwise.} \end{cases}$
- $Inter(N_p) = \bigcap_{q \in N_p} poss(q)$.

Rules:
/* Normal rule */
$(\boldsymbol{N})$ :: $|Inter(N_p)| \geq 2 \longrightarrow c_p := next\left(Inter(N_p), c_p\right)$
/* Correction rules */
$(\boldsymbol{C_1})$ :: $\left(|Inter(N_p)| = 0\right) \wedge \left(c_p \neq \left\lceil \frac{\sum_{q \in N_p} c_q}{|N_p|} \right\rceil\right) \wedge \left(c_p \neq \left\lfloor \frac{\sum_{q \in N_p} c_q}{|N_p|} \right\rfloor\right) \longrightarrow c_p := \left\lfloor \frac{\sum_{q \in N_p} c_q}{|N_p|} \right\rfloor$
$(\boldsymbol{C_2})$ :: $(Inter(N_p) = \{h\}) \wedge (c_p \neq h) \longrightarrow c_p := h$

4. A universal protocol for chains and rings

In the following we consider the only remaining possibility results (see Table 1) that are related to asynchronous unison on chains and rings (*i.e.* networks with a degree inferior to 3). In this section, we propose an (1, 0)-FTSS algorithm for **AU** under a locally central strongly fair daemon. The proposed algorithm is both minimal and priority.

The main difference between our protocol and the many self-stabilizing unison algorithms existing in the literature [9,11,12,29] is that our correction rules use averaging rather than maximizing or minimizing, in order to not favor the clock value of a particular neighbor. Indeed, using a maximum or a minimum strategy could make the chosen neighbor prevent stabilization if it is crashed. The averaging idea was previously studied in [24] in a non-stabilizing fault-free setting. [23] uses also average to perform clock synchronization in a non-stabilizing Byzantine-tolerant system. The main difference with our approach is that authors of [23] reject values that are too far from others (in order to avoid values proposed by Byzantine neighbors). In our case, we cannot reject any value due to the arbitrary initial clock values and the small number of available values (as our protocol operates on chains or rings, each processor has at most two neighbors).

4.1. Our algorithm

The main idea of our algorithm follows. Each processor checks if it is "locally synchronized", *i.e.* if the drift between its clock value and the clock values of its neighbors does not exceed 1. If a processor p is "locally synchronized", it modifies its clock value in a finite time in order to preserve this property. Otherwise, p corrects its clock value in finite time.

More precisely, each processor p has only one variable: its clock denoted by c_p. At each step, every processor p computes a set of *possible clock values*, *i.e.* the set of clock values that have a drift of at most 1 with respect to all neighbors of p (note that computing this set relies only on the clock values of p's neighbors, but not on the one of p). This set is denoted by $Inter(N_p)$.

Then, the following cases may appear:

- $|Inter(N_p)| = 0$, then p has two neighbors and the drift between their clock values is strictly greater than 2. In this case, p is enabled to take the average value between these two clock values if its clock does not have yet this value.
- $|Inter(N_p)| = 1$, then p has two neighbors and the drift between their clock values is exactly 2. In this case, p is enabled to take the average value between these two clock values if its clock does not have yet this value.
- $|Inter(N_p)| \geq 2$, then p has one neighbor or the drift between the clock values of its two neighbors is strictly less than 2. In this case, p is enabled to modify its clock value as follows: if $c_p + 1 \in Inter(N_p)$, then c_p is modified to $c_p + 1$, otherwise c_p is modified to $min\{Inter(N_p)\}$.

The reader can find some examples of execution of our algorithm in Figs. 8–11.

The detailed description of our solution is proposed in Algorithm 1.

4.2. Correction proof road map

In this section, we present the key ideas in order to prove the correctness of our algorithm.

First, we introduce some useful notations:

Notation 1. *Let p be a processor. If q denotes one of its neighbors, we denote the other neighbor by $\bar{q}$ (if this neighbor exists).*

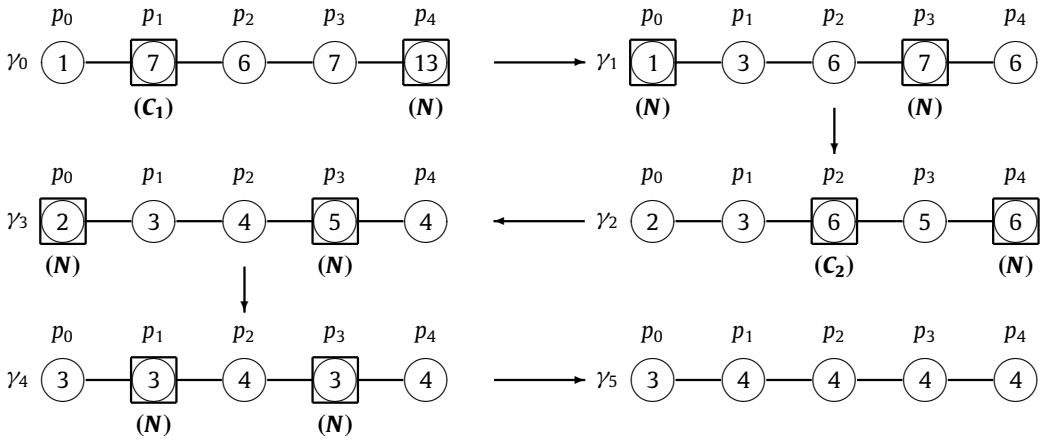

Fig. 8. An example of execution of $\mathcal{UFTSS}$ on a chain with no crash (the numbers represent clock values and squared processors in γ_i executed the indicated rule during the step $\gamma_i \longrightarrow \gamma_{i+1}$).

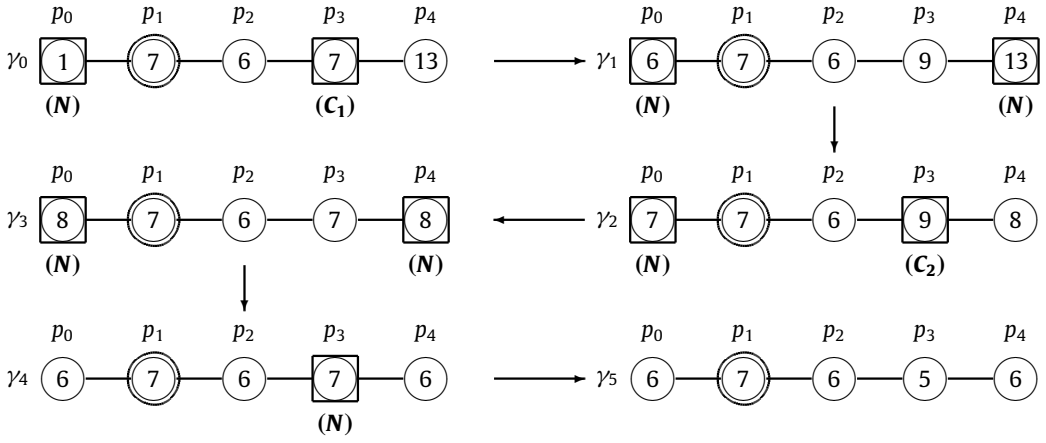

Fig. 9. An example of execution of $\mathcal{UFTSS}$ on a chain with a crash (the numbers represent clock values, the double circles represent crashed processors and squared processors in γ_i executed the indicated rule during the step $\gamma_i \longrightarrow \gamma_{i+1}$).

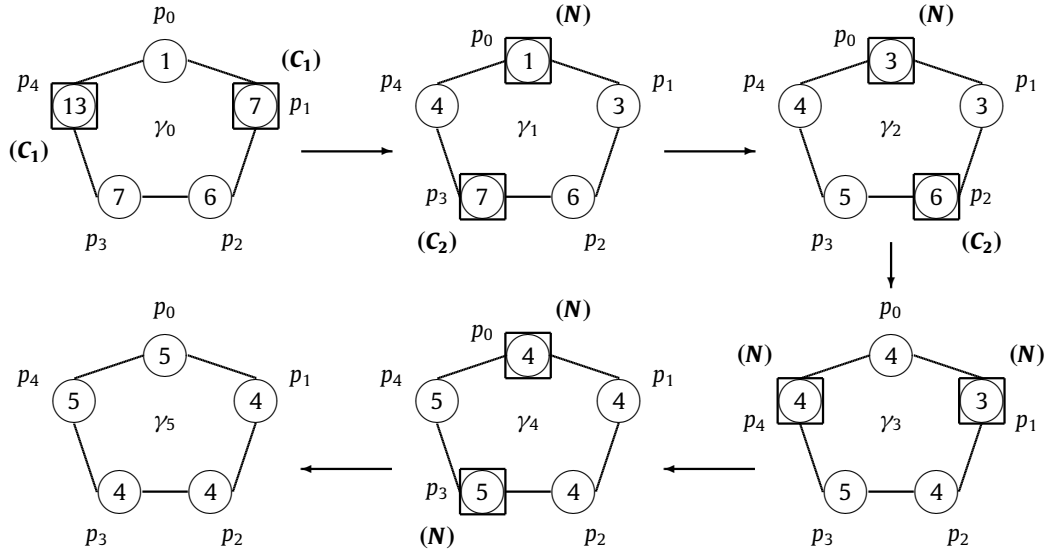

Fig. 10. An example of execution of $\mathcal{UFTSS}$ on a ring with no crash (the numbers represent clock values and squared processors in γ_i executed the indicated rule during the step $\gamma_i \longrightarrow \gamma_{i+1}$).

Notation 2. *We denote the value of c_p for a processor p in a configuration γ_i by $(c_p)^{\gamma_i}$.*

We denote the value of $Inter(N_p)$ for a processor p in a configuration γ_i by $\left(Inter(N_p)\right)^{\gamma_i}$.

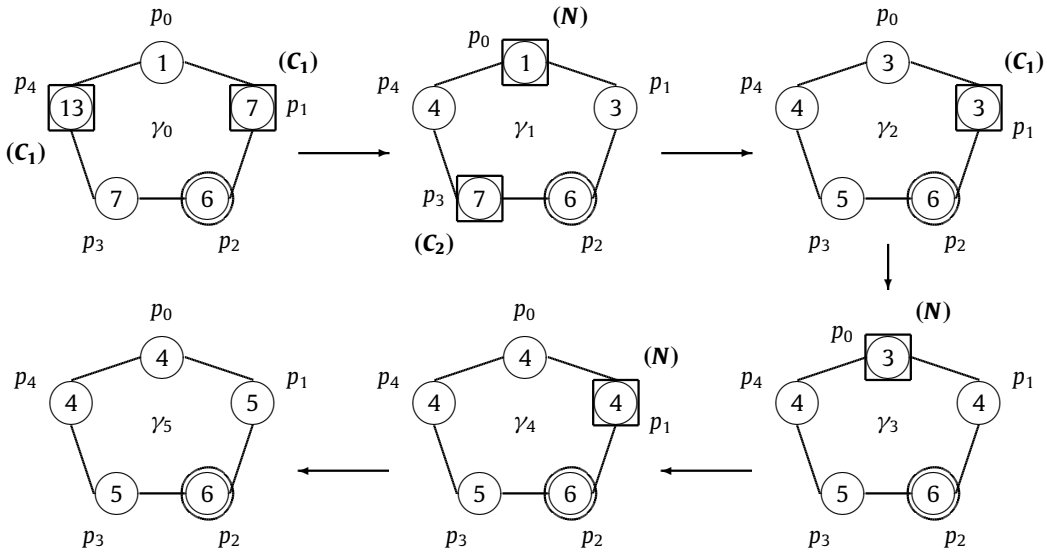

Fig. 11. An example of execution of $\mathcal{UFTSS}$ on a ring with a crash (the numbers represent clock values, the double circles represent crashed processors and squared processors in γ_i executed the indicated rule during the step $\gamma_i \longrightarrow \gamma_{i+1}$).

In order to prove that $\mathcal{UFTSS}$ is a (1, 0)-ftss algorithm for **AU** under a locally central strongly fair daemon on a chain and on a ring (see Proposition 11), we prove in what follows the following properties:

1. $\mathcal{UFTSS}$ is a self-stabilizing algorithm for **AU** under a locally central strongly fair daemon on a chain (Proposition 7).
2. $\mathcal{UFTSS}$ is a self-stabilizing algorithm for **AU** under a locally central strongly fair daemon on a chain even if one processor is crashed in the initial configuration (Proposition 8).
3. $\mathcal{UFTSS}$ is a self-stabilizing algorithm for **AU** under a locally central strongly fair daemon on a ring (Proposition 9).
4. $\mathcal{UFTSS}$ is a self-stabilizing algorithm for **AU** under a locally central strongly fair daemon on a ring even if one processor is crashed in the initial configuration (Proposition 10).

The proof of each of these 4 propositions is deduced from 3 lemmas as follows:

1. Firstly, we prove that $\mathcal{UFTSS}$ satisfies the closure of the safety of **UAU** under the considered hypothesis (*i.e.* if there exists a configuration γ such that $\gamma \in \Gamma_1$, then every configuration γ' reachable from γ satisfies: $\gamma' \in \Gamma_1$, see respectively Lemmas 4, 10, 13 and 19).
   The idea of the proof is as follows: we first prove that only the normal rule is enabled in such a configuration and then, we show that this rule ensures the closure of the safety property.
2. Secondly, we prove that $\mathcal{UFTSS}$ satisfies liveness of **UAU** under the considered hypothesis in every execution starting from a legitimate configuration (*i.e.* every (correct) processor increments infinitely often its clock, see respectively Lemmas 6, 11, 15 and 20).
   This proof is done in the following way: we first show that every (correct) processor executes infinitely often the normal rule in every execution starting from a configuration $\gamma \in \Gamma_1$ and then, we show that if a processor executes infinitely often the normal rule, it increments its clock in a finite time.
3. Finally, we prove that $\mathcal{UFTSS}$ converges to a legitimate configuration of **UAU** under the considered hypothesis in every execution (*i.e.* there exists a configuration $\gamma \in \Gamma_1$ in every execution, see respectively Lemmas 9, 12, 18 and 21).
   In order to complete this proof we study a potential function.

4.3. Proof on a chain

In this section, we assume that our algorithm is executed on a chain under a strongly fair locally central daemon. In the following we prove that $\mathcal{UFTSS}$ is a FTSS **UAU** (that implies that it is a FTSS **AU**) under these assumptions. The proof contains two major steps:

- First, we prove that our algorithm is self-stabilizing.
- Second, we prove that our algorithm is self-stabilizing even if the initial configuration contains a crashed processor.

4.3.1. Proof of self-stabilization

In this section, $\epsilon = \gamma_0, \gamma_1 \ldots$ denotes an execution of $\mathcal{UFTSS}$ where there is no crash.
Firstly, we are going to prove the closure of our algorithm.

Lemma 4. *If there exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$, then $\gamma_{i+1} \in \Gamma_1$.*

Proof. Assume that there exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$. This implies that $\forall p \in V$, $\left(Inter(N_p)\right)^{\gamma_i} \neq \emptyset$ and then the rule **(C_1)** is not enabled in γ_i. Assume rule **(C_2)** is enabled in γ_i. This implies that $\left(Inter(N_p)\right)^{\gamma_i} = \{h\}$ and that $\left(c_p\right)^{\gamma_i} \neq h$. Then, we have $\gamma_i \notin \Gamma_1$ (since if $\left(c_p\right)^{\gamma_i} \neq h$, then the following holds: $\exists q \in N_p, |\left(c_p\right)^{\gamma_i} - \left(c_q\right)^{\gamma_i}| \geq 2$). This contradiction allows us to conclude that the enabled processors in γ_i are only enabled for rule **(N)**.

Let p be a processor that executes a rule during the step $\gamma_i \rightarrow \gamma_{i+1}$. Since the daemon is locally central, neighbors of p do not execute a rule during this step (their clock values remain identical). Assume the following holds: $\exists q \in N_p, \left|\left(c_p\right)^{\gamma_{i+1}} - \left(c_q\right)^{\gamma_{i+1}}\right| \geq 2$. By construction of rule **(N)**, $\left(c_p\right)^{\gamma_{i+1}} \in \left(Inter(N_p)\right)^{\gamma_i}$. By construction, $\left(Inter(N_p)\right)^{\gamma_i} \subseteq \left\{\left(c_q\right)^{\gamma_i} - 1, \left(c_q\right)^{\gamma_i}, \left(c_q\right)^{\gamma_i} + 1\right\}$. It follows that $\forall q \in N_p, \left|\left(c_p\right)^{\gamma_{i+1}} - \left(c_q\right)^{\gamma_{i+1}}\right| < 2$ for each processor p that executes a rule (since $\forall q \in N_p, \left(c_q\right)^{\gamma_i} = \left(c_q\right)^{\gamma_{i+1}}$). Overall, $\gamma_{i+1} \in \Gamma_1$. □

Secondly, we prove the liveness of our algorithm.

Lemma 5. *$\forall \gamma_0 \in \Gamma_1, \forall p \in V$, p executes the rule **(N)** in a finite time in any execution starting from γ_0.*

Proof. Let $\gamma \in \Gamma_1$. Following Lemma 4, the only enabled rule is **(N)**. We prove this property by induction. To this end, we define the following property (where p denotes a processor):
(P_d): If d is the distance between p and the closest end of the chain, then p executes the rule **(N)** in a finite time in any execution starting from γ_0.

Initialization ($d = 0$): For all γ', configurations contained in an execution starting from γ_0, p is enabled for rule **(N)** since $\left(Inter(N_p)\right)^{\gamma'} \supseteq \left\{\left(c_q\right)^{\gamma'}, \left(c_q\right)^{\gamma'} + 1\right\}$ where q denotes the only neighbor of p. Since the daemon is strongly fair, p executes a rule in a finite time.

Induction ($d > 0$): Assume **(P_{d-1})** is true. Denote by q the neighbor of p that is on the half-chain starting with p of length d. Assume for the sake of contradiction that p is never enabled for rule **(N)** in an execution ϵ starting from $\gamma_0 \in \Gamma_1$. This implies that, for each configuration γ' that is contained in ϵ, we have $\left|\left(Inter(N_p)\right)^{\gamma'}\right| = 1$ (since if $\left|\left(Inter(N_p)\right)^{\gamma'}\right| = 0$, then $\gamma' \notin \Gamma_1$). Let us study the following cases (remind that, if q denotes a neighbor of p, $\bar{q}$ denotes the second neighbor of p as stated in Notation 1):

Case 1: $\bar{q}$ never executes a rule in ϵ (this implies that $c_{\bar{q}}$ is a constant in ϵ).

It follows that: $\forall \gamma' \in \epsilon$, $\left(c_q\right)^{\gamma'} = \left(c_{\bar{q}}\right)^{\gamma'} + 2$ or $\left(c_q\right)^{\gamma'} = \left(c_{\bar{q}}\right)^{\gamma'} - 2$.

As q executes infinitely often rule **(N)**, its clock moves at each activation from a value to the other. Hence, we have $\left(c_q\right)^{\gamma'} = \left(c_{\bar{q}}\right)^{\gamma'} - 2$ in a finite time. Then, the next activation of q moves its clock value to $\left(c_{\bar{q}}\right)^{\gamma'} + 2$, which is contradictory with the construction of macro *next* (it can only increment the clock value by 1 or decrement it).

Case 2: $\bar{q}$ executes a rule in a finite time in ϵ.

Let $\gamma \rightarrow \gamma'$ be the first step when $\bar{q}$ executes the rule **(N)**. It is known that, for any $\gamma \in \Gamma_1$:

$$\left|\left(Inter(N_p)\right)^{\gamma}\right| = 1 \Rightarrow \begin{cases} \left(c_{\bar{q}}\right)^{\gamma} = \left(\left(c_p\right)^{\gamma} - 1\right) \wedge \left(c_q\right)^{\gamma} = \left(\left(c_p\right)^{\gamma} + 1\right) \ \textbf{(A)} \\ \textit{or} \\ \left(c_{\bar{q}}\right)^{\gamma} = \left(\left(c_p\right)^{\gamma} + 1\right) \wedge \left(c_q\right)^{\gamma} = \left(\left(c_p\right)^{\gamma} - 1\right) \ \textbf{(B)} \end{cases}$$

Let us study the following cases:

Case 2.1: **(A)** is true in γ and **(B)** is true in γ'. The clock move of $\bar{q}$ is in contradiction with the construction of macro *next*.

Case 2.2: **(B)** is true in γ and **(A)** is true in γ'. The clock move of q is in contradiction with the construction of macro *next*.

This proves that Case 2 is contradictory.

Since the two cases are contradictory, we can conclude that p is enabled for rule **(N)** in a finite time in every execution starting from a configuration $\gamma \in \Gamma_1$. Since the daemon is strongly fair, we can say that p executes rule **(N)** in a finite time in every execution starting from γ_0. Consequently **(P_d)** is true. □

The above property implies that $\forall \gamma_0 \in \Gamma_1, \forall p \in V$, p executes the rule **(N)** infinitely often in every execution starting from γ_0.

Lemma 6. *If $\gamma \in \Gamma_1$, then any processor increments its clock in a finite time in any execution starting from γ.*

Proof. Assume for the sake of contradiction that there exists a processor p and an execution ϵ starting from $\gamma_0 \in \Gamma_1$ such that p never increments its clock in ϵ.

Let $\alpha = \left(c_p\right)^{\gamma_0}$. By Lemma 5, p executes infinitely often **(N)**. But, it never increments its clock, which implies that $next\left(\left(Inter(N_p)\right)^{\gamma}, \left(c_p\right)^{\gamma}\right) = min\left\{\left(Inter(N_p)^{\gamma}\right)\right\}$ at each execution of a rule by p (in a configuration γ). Since $\forall \gamma \in \Gamma_1, \forall q \in N_p, \left|\left(c_p\right)^{\gamma} - \left(c_q\right)^{\gamma}\right| < 2$ and $\forall q \in N_p, \left(Inter(N_p)\right)^{\gamma} \subseteq \left\{\left(c_q\right)^{\gamma} - 1, \left(c_q\right)^{\gamma}, \left(c_q\right)^{\gamma} + 1\right\}$, we have: $min\left\{\left(Inter(N_p)\right)^{\gamma}\right\} \leq \left(c_p\right)^{\gamma}$.

Assume that there exists $\gamma \in \Gamma_1$ such that $min\{\left(Inter(N_p)\right)^\gamma\} = \left(c_p\right)^\gamma$. This implies that there exists $q \in N_p$ such that $\left(c_q\right)^\gamma = \left(c_p\right)^\gamma + 1$.

Remind that, if q denotes a neighbor of p, $\bar{q}$ denotes the second neighbor of p as stated in Notation 1. If $\bar{q}$ does not exist or if $\left(c_{\bar{q}}\right)^\gamma \in \left\{\left(c_p\right)^\gamma, \left(c_p\right)^\gamma + 1\right\}$, then $\left(c_p\right)^\gamma + 1 \in \left(Inter(N_p)\right)^\gamma$. This contradicts $next(\left(Inter(N_p)\right)^\gamma, \left(c_p\right)^\gamma) = min\{\left(Inter(N_p)\right)^\gamma)\}$. We deduce that $\bar{q}$ exists and that $\left(c_{\bar{q}}\right)^\gamma = \left(c_p\right)^\gamma - 1$. This implies that $(\boldsymbol{N})$ is not enabled for p.

We can deduce that, if rule $(\boldsymbol{N})$ is executed by a processor p in a configuration γ, then $min\left\{\left(Inter(N_p)\right)^\gamma\right\} < \left(c_p\right)^\gamma$. We can now state that, in at most α executions of p, $c_p = 0$. The next execution of p increments its clock value, which contradicts the assumption on p and the construction of ϵ. Then, we obtain the result. □

In the following we prove the convergence of our algorithm.

Let $\gamma \in \Gamma$, we define the following notations:

$$\begin{aligned} \forall e = \{p, q\} \in E, \omega(e, \gamma) &= \left|\left(c_p\right)^\gamma - \left(c_q\right)^\gamma\right| \\ \forall p \in V, \varpi(p, \gamma) &= \max_{e \in E / p \in e} \{\omega(e, \gamma)\} \\ \forall i \in \mathbb{N}, p(i, \gamma) &= |\{e \in E / \omega(e, \gamma) = i\}|. \end{aligned}$$

Consider the following potential function:

$$P : \begin{cases} \Gamma \longrightarrow \mathbb{N}^\infty \\ \gamma \longmapsto (\ldots, 0, 0, p(k, \gamma), p(k-1, \gamma), \ldots, p(2, \gamma)) \text{ with } k = \max_{e \in E}\{\omega(e, \gamma)\}. \end{cases}$$

To compare values of P, we define the following total order. If γ and γ' are two configurations such that $P(\gamma) = (\ldots, 0, p_i, p_{i-1}, \ldots, p_2)$ and $P(\gamma') = (\ldots, 0, q_j, q_{j-1}, \ldots, q_2)$, then

$$P(\gamma) > P(\gamma') \Leftrightarrow \begin{cases} i > j \\ \text{or} \\ (i = j) \wedge (\exists t \in \{2, \ldots, i\}, (\forall k \in \{t+1, \ldots, i\}, p_k = q_k) \wedge (p_k > q_k)). \end{cases}$$

The following properties are satisfied:

$$\begin{aligned} \forall \gamma \in \Gamma, P(\gamma) &\geq (\ldots 0, 0) \\ \forall \gamma \in \Gamma, \gamma \in \Gamma_1 &\Leftrightarrow P(\gamma) = (\ldots, 0, 0) \\ \forall \gamma \in \Gamma, \gamma \in \Gamma \setminus \Gamma_1 &\Leftrightarrow P(\gamma) > (\ldots, 0, 0). \end{aligned}$$

Lemma 7. *If $\gamma \in \Gamma \setminus \Gamma_1$, then every step $\gamma \to \gamma'$, which contains the execution of a rule by a processor p such that $\varpi(p) \geq 2$ satisfies $P(\gamma') < P(\gamma)$.*

Proof. Let $\gamma \in \Gamma \setminus \Gamma_1$. Let $\gamma \to \gamma'$ be a step that contains the execution of a rule by a processor p such that $\varpi(p) \geq 2$ and $\gamma \in \Gamma \setminus \Gamma_1$. Since the daemon is locally central, neighbors of p do not modify their clocks during this step. Consider the following cases:

Case 1: p's degree equals 1.
Let q be its only neighbor and $j = \omega(\{p, q\}, \gamma) = \left|\left(c_p\right)^\gamma - \left(c_q\right)^\gamma\right|$. $\left(Inter(N_p)\right)^\gamma = \left\{\left(c_q\right)^\gamma - 1, \left(c_q\right)^\gamma, \left(c_q\right)^\gamma + 1\right\}$.
It follows that p executed rule $(\boldsymbol{N})$. So, we have $\left|\left(c_p\right)^{\gamma'} - \left(c_q\right)^{\gamma'}\right| \leq 1$. Then: $\varpi(\{p, q\}, \gamma') \leq 1$ and:

$$\begin{aligned} P(\gamma) &= (\ldots, 0, 0, p(k, \gamma), p(k-1, \gamma), \ldots, p(j, \gamma), \ldots, p(2, \gamma)) \\ P(\gamma') &= (\ldots, 0, 0, p(k, \gamma), p(k-1, \gamma), \ldots, p(j, \gamma) - 1, \ldots, p(2, \gamma)). \end{aligned}$$

And then: $P(\gamma') < P(\gamma)$.

Case 2: p's degree equals 2.
Let q be the neighbor of p such that $\omega(\{p, q\}, \gamma) = \varpi(p, \gamma) \geq 2$ and denote $j = \omega(\{p, \bar{q}\}, \gamma) \leq \varpi(p, \gamma)$, $e = \{p, q\}$ and $\bar{e} = \{p, \bar{q}\}$. Consider the following cases:

Case 2.1: p executed the rule $(\boldsymbol{N})$ during the step $\gamma \to \gamma'$.
By construction of $\left(Inter(N_p)\right)^\gamma$, we have $\omega(e, \gamma') \leq 1$ and $\omega(\bar{e}, \gamma') \leq 1$. Then:

$$\begin{aligned} P(\gamma) &= (\ldots, 0, 0, p(k, \gamma), p(k-1, \gamma), \ldots, p(\varpi(p, \gamma), \gamma), \ldots, p(j, \gamma), \ldots, p(2, \gamma)) \\ P(\gamma') &= (\ldots, 0, p(k, \gamma), \ldots, p(\varpi(p, \gamma), \gamma) - 1, \ldots, p(j, \gamma) - 1, \ldots, p(2, \gamma)). \end{aligned}$$

And then: $P(\gamma') < P(\gamma)$.

Case 2.2: p executed the rule $(\boldsymbol{C_2})$ during the step $\gamma \to \gamma'$.
This case is similar to the Case 2.1.

Case 2.3: p executed the rule $(\boldsymbol{C_1})$ during the step $\gamma \to \gamma'$.

Let us study the following cases:

Case 2.3.1: We have: $(c_q)^\gamma < (c_{\bar{q}})^\gamma$.

By hypothesis, we know that $\omega(e, \gamma) \geq \omega(\bar{e}, \gamma)$ and then:

$$(c_p)^\gamma \geq \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2}.$$

(1) Assume that $(c_p)^\gamma > (c_{\bar{q}})^\gamma + \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2}$.

We can say that:

$$\omega(e, \gamma) > (c_{\bar{q}})^\gamma - (c_q)^\gamma + \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2}$$

$$\omega(e, \gamma') = \left\lfloor \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2} \right\rfloor.$$

Then: $\omega(e, \gamma') < \omega(e, \gamma)$.

On the other hand,

$$\omega(\bar{e}, \gamma) > \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2}$$

$$\omega(\bar{e}, \gamma') = (c_{\bar{q}})^\gamma - \left\lfloor \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2} \right\rfloor.$$

Then: $\omega(\bar{e}, \gamma') \leq \omega(\bar{e}, \gamma)$.

In conclusion, we have: $P(\gamma') < P(\gamma)$.

(2) Assume that $(c_p)^\gamma \leq (c_{\bar{q}})^\gamma + \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2}$.

We have then:

$$\omega(e, \gamma) > \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2}$$

$$\omega(e, \gamma') = \left\lfloor \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2} \right\rfloor.$$

Then: $\omega(e, \gamma') < \omega(e, \gamma)$.

In contrast, we have that: $\omega(\bar{e}, \gamma') \geq \omega(\bar{e}, \gamma)$. But we can say that $\omega(\bar{e}, \gamma') < \omega(e, \gamma)$ (obvious if $(c_p)^\gamma > (c_{\bar{q}})^\gamma$, due to the fact that $(c_p)^\gamma > \left\lceil \frac{(c_q)^\gamma + (c_{\bar{q}})^\gamma}{2} \right\rceil$ in the contrary case).

In conclusion, we have: $P(\gamma') < P(\gamma)$.

Case 2.3.2: We have $(c_q)^\gamma > (c_{\bar{q}})^\gamma$.

This case is similar to the Case 2.3.1 when we permute q and $\bar{q}$.

That proves the result. □

Lemma 8. *If $\gamma_0 \in \Gamma \setminus \Gamma_1$, then every execution starting from γ_0 contains the execution of a rule by a processor p such that $\varpi(p, \gamma_0) \geq 2$.*

Proof. Let $\gamma_0 \in \Gamma \setminus \Gamma_1$. We prove the result by contradiction. Assume that there exists an execution $\epsilon = \gamma_0\gamma_1 \ldots$ starting from γ_0, which contains no execution of a rule by processors p satisfying $\varpi(p, \gamma_0) \geq 2$.

In a first time, assume that one end of the chain (denote it by p) satisfies: $\varpi(p, \gamma_0) \geq 2$. Denote q the only neighbor of p. If q is activated during ϵ, we obtain a contradiction (since $\varpi(q, \gamma_0) \geq \varpi(p, \gamma_0) \geq 2$). If q is not activated during ϵ, we obtain that $\forall i \in \mathbb{N}$, $(Inter(N_p))^{\gamma_i} = \{(c_q)^{\gamma_0} - 1, (c_q)^{\gamma_0}, (c_q)^{\gamma_0} + 1\}$, p is so always enabled for rule (***N***). Since the daemon is strongly fair, p executes a rule in a finite time, which is contradictory. We can deduce that the two ends of the chain satisfy: $\varpi(p, \gamma_0) < 2$.

Under a strongly fair daemon, the only way for a processor to never execute a rule is to be never enabled from a given configuration. Here, we assume that all processors p satisfying $\varpi(p, \gamma_0) \geq 2$ never execute a rule, which implies that the network satisfies:

$$\exists k \in \mathbb{N}, \forall j \geq k, \forall p \in V / \varpi(p, \gamma_0) \geq 2, \begin{cases} (Inter(N_p))^{\gamma_j} = \emptyset \\ \text{and} \\ (c_p)^{\gamma_j} \in \left\{ \left\lceil \frac{(c_q)^{\gamma_j} + (c_{\bar{q}})^{\gamma_j}}{2} \right\rceil, \left\lfloor \frac{(c_q)^{\gamma_j} + (c_{\bar{q}})^{\gamma_j}}{2} \right\rfloor \right\}. \end{cases}$$

Number processors of the chain from p_1 to p_n. Let i be the smallest integer such that $\varpi(p_i, \gamma_k) \geq 2$ (remark that, by hypothesis, p_{i+1} never execute a rule, which implies that its clock value never changes). All these constraints allows us to say:

$$\begin{cases} (c_{p_{i-1}})^{\gamma_k} = (c_{p_i})^{\gamma_k} + 1 \wedge (c_{p_{i+1}})^{\gamma_k} = (c_{p_i})^{\gamma_k} - 2 \text{ (}\boldsymbol{A}\text{)} \\ \text{or} \\ (c_{p_{i-1}})^{\gamma_k} = (c_{p_i})^{\gamma_k} - 1 \wedge (c_{p_{i+1}})^{\gamma_k} = (c_{p_i})^{\gamma_k} + 2 \text{ (}\boldsymbol{B}\text{)} \end{cases}$$

By a reasoning similar to these of the proof of Lemma 6, we can prove that all processors between p_0 and p_{i-1} executes infinitely often the rule (**N**) in every execution starting from γ_k even if p_i never executes a rule (this is the case by hypothesis). By a reasoning similar to the one of the proof of Lemma 6, we can state that $c_{p_{i-1}}$ not remains constant. The construction of $Inter(N_{p_{i-1}})$ implies that $\left(Inter(N_{p_{i-1}})\right)^{\gamma_j} \subseteq \left\{\left(c_{p_i}\right)^{\gamma_k} - 1, \left(c_{p_i}\right)^{\gamma_k}, \left(c_{p_i}\right)^{\gamma_k} + 1\right\}$ for each $j \geq k$ (since c_{p_i} does not change by hypothesis).

If we are in case (**A**), we can deduce that $c_{p_{i-1}}$ takes infinitely often the value $\left(c_{p_i}\right)^{\gamma_k} - 1$ or $\left(c_{p_i}\right)^{\gamma_k}$. We can see that p_i is enabled by (**N**) and (**C1**) respectively. This contradicts the construction of k (recall that p_i is never enabled in ϵ from γ_k).

If we are in case (**B**), we can deduce that $c_{p_{i-1}}$ takes infinitely often the value $\left(c_{p_i}\right)^{\gamma_k} + 1$ or $\left(c_{p_i}\right)^{\gamma_k}$. We can see that p_i is enabled by (**N**) and (**C1**) respectively. This contradicts the construction of k (recall that p_i is never enabled in ϵ from γ_k).

This finishes the proof. □

Lemma 9. *There exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$.*

Proof. The result follows directly from Lemmas 7 and 8. □

Finally, we can conclude:

Proposition 7. *$\mathcal{UFTSS}$ is a self-stabilizing **AU** under a locally central strongly fair daemon.*

Proof. Lemmas 4, 6 and 9 allows us to say that $\mathcal{UFTSS}$ is a self-stabilizing **UAU** under a locally central strongly fair daemon. Then, we can deduce the result. □

4.3.2. Proof of self-stabilization in spite of a crash

In this section, $\epsilon = \gamma_0, \gamma_1 \ldots$ denotes an execution of $\mathcal{UFTSS}$ such that a processor c is crashed in γ_0.

Firstly, we are going to prove the closure of our algorithm under these assumptions.

Lemma 10. *If there exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$, then $\gamma_{i+1} \in \Gamma_1$.*

Proof. We can repeat the reasoning of Lemma 4 since the fact that a processor is crashed or not does not modify the proof. □

Secondly, we are going to prove the liveness of our algorithm under these assumptions.

Lemma 11. *If $\gamma_0 \in \Gamma_1$, then every processor $p \neq c$ increments its clock in a finite time in ϵ.*

Proof. We repeat the reasoning of Lemma 6 taking in account a processor $p \in V^*$.

In order to prove the property of Lemma 5, we take d as the distance between p and the end e of the chain that satisfy: no processor between p and e is crashed. This implies that the processor q is not crashed. The case where $\bar{q}$ is crashed appear in the Case 1 of the induction.

We can repeat the reasoning of the proof of Lemma 6 since the fact that a processor is crashed or not does not modify the proof. □

Now, we are going to prove the convergence of our algorithm under these assumptions.

Lemma 12. *There exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$.*

Proof. We repeat the reasoning of Lemma 9 taking in account a processor $p \in V^*$.

We can repeat the reasoning of the proof of the property of Lemma 7 since the fact that a processor is crashed or not does not modify the proof.

In order to prove the property of Lemma 8, we take a numbering of processors that ensures the following property: no processor between p_0 and p_i (including) is crashed. It is always possible to choose such numbering since there exists at least one edge e such that $\omega(e, \gamma_k) \geq 2$ by hypothesis, which implies that there exists at least two processors p such that $\varpi(p, \gamma_k) \geq 2$, which allows us to choose one that is not crashed. The case when p_{i+1} is crashed does not modify the proof since we assumed that this processor never executes a rule. □

Finally, we can conclude:

Proposition 8. *$\mathcal{UFTSS}$ is a self-stabilizing **AU** under a locally central strongly fair daemon even if a processor is crashed in the initial configuration.*

Proof. Lemmas 10–12 allows us to say that $\mathcal{UFTSS}$ is a self-stabilizing **UAU** under a locally central strongly fair daemon even if a processor is crashed in the initial configuration. Then, we can deduce the result. □

4.4. Proof on a ring

In this section, we assume that our algorithm is executed on a ring under a strongly fair locally central daemon. In fact, we are going to show that $\mathcal{UFTSS}$ is a FTSS **UAU** (that implies that it is a FTSS **AU**) under these assumptions. The proof contains two major steps:

- Firstly, we show that our algorithm is self-stabilizing under these assumptions.
- Secondly, we show that our algorithm is self-stabilizing even if the initial configuration contains a crashed processor under these assumptions.

4.4.1. Proof of self-stabilization

In this section, $\epsilon = \gamma_0, \gamma_1 \ldots$ denotes an execution of $\mathcal{UFTSS}$ where there is no crash.
Firstly, we are going to prove the closure of our algorithm under these assumptions.

Lemma 13. *If there exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$, then $\gamma_{i+1} \in \Gamma_1$.*

Proof. We can repeat the reasoning of the proof of Lemma 4 since the topology of the network has no impact on the proof. □

Secondly, we are going to prove the liveness of our algorithm under these assumptions.

Lemma 14. *$\forall \gamma_0 \in \Gamma_1, \forall p \in V$, p executes rule **(N)** in a finite time in every execution starting from γ_0.*

Proof. Let $\gamma_0 \in \Gamma_1$ (we have seen in the proof of Lemma 4 that implies that only rule **(N)** can be enabled). Assume that there exists a processor p and an execution $\epsilon = \gamma_0, \gamma_1 \ldots$ starting from γ_0 such that p never execute a rule in ϵ. Since the daemon is strongly fair, which implies that $\exists k \in \mathbb{N}, \forall j \geq k$, p is not enabled in γ_j.

Since Processor p is not enabled, it satisfies: $\exists q \in N_p$, $(c_p)^{\gamma_j} = (c_q)^{\gamma_j} + 1$ and $(c_p)^{\gamma_j} = (c_{\bar{q}})^{\gamma_j} - 1$. Let i be the smallest integer greater than k such that the step $\gamma_i \to \gamma_{i+1}$ contains the execution of rule by at least one neighbor of p. Let us study the following cases:

Case 1: q and $\bar{q}$ simultaneously execute a rule during the step $\gamma_i \to \gamma_{i+1}$.
Since p is not enabled in γ_{i+1} (by hypothesis) and that the execution of rule **(N)** always modifies the clock values (*cf.* proof of Lemma 6), we have:

$$\begin{cases} (c_p)^{\gamma_i} = (c_q)^{\gamma_i} + 1 \text{ and } (c_p)^{\gamma_i} = (c_{\bar{q}})^{\gamma_i} - 1 \\ \text{and} \\ (c_p)^{\gamma_{i+1}} = (c_q)^{\gamma_{i+1}} - 1 \text{ and } (c_p)^{\gamma_{i+1}} = (c_{\bar{q}})^{\gamma_{i+1}} + 1. \end{cases}$$

The clock move of $\bar{q}$ contradicts the construction of rule **(N)** and $\left(Inter(N_p)\right)^{\gamma_i}$. Therefore, this case is impossible.

Case 2: Only q executes a rule during the step $\gamma_i \to \gamma_{i+1}$.
By construction of rule **(N)**, $\left(Inter(N_q)\right)^{\gamma_i}$, and the fact that the execution of this rule must change the clock value, we have: $(c_q)^{\gamma_{i+1}} \in \left\{(c_p)^{\gamma_i}, (c_p)^{\gamma_i} - 1\right\}$. Processor p is then enabled for rule **(N)** (since the clocks of p and $\bar{q}$ have not changed by hypothesis). This contradicts the construction of k. Therefore, this case is impossible.

Case 3: Only $\bar{q}$ executes a rule during the step $\gamma_i \to \gamma_{i+1}$.
This case is similar to Case 2.

Case 4: Neither q nor $\bar{q}$ executes a rule during the step $\gamma_i \to \gamma_{i+1}$.
By the three previous contradictions, it is the only possible case.

We can deduce that $\forall j \geq k$, q and $\bar{q}$ do not execute a rule in γ_j, which implies that their clock values remains constant from γ_k. If we repeat the previous reasoning, we obtain that it is possible only if the second neighbor of q has a clock value equal to $(c_p)^{\gamma_k} + 2$ and if the second neighbor of $\bar{q}$ have a clock value equals to $(c_p)^{\gamma_k} - 2$, etc.

Since the ring has a finite length n, we obtain (following the same reasoning) that there exists two neighboring processors p_1 and p_2 such that $(c_{p_1})^{\gamma_k} = (c_p)^{\gamma_k} + \alpha$ and $(c_{p_2})^{\gamma_k} = (c_p)^{\gamma_k} - \beta$ (with α and β integers greater or equal to 1 depending on the parity of n). Therefore, $\left|(c_{p_1})^{\gamma_k} - (c_{p_2})^{\gamma_k}\right| = \alpha + \beta \geq 2$. Then, we obtain that $\gamma_k \notin \Gamma_1$, which contradicts Lemma 13 and proves the lemma. □

Lemma 15. *If $\gamma_0 \in \Gamma_1$, then every processor increments its clock in a finite time in ϵ.*

Proof. The proof is similar to the one of Lemma 6 using Lemma 14 (instead of Lemma 5) since the topology of the network has no impact on the proof. □

Now, we are going to prove the convergence of our algorithm under these assumptions.

In the following, we consider the potential function P previously defined and use similar arguments as for the proof of Lemma 9.

Lemma 16. *If $\gamma \in \Gamma \setminus \Gamma_1$, then every step $\gamma \to \gamma'$ that contains the execution of a rule of a processor p such that $\varpi(p) \geq 2$ satisfies $P(\gamma') < P(\gamma)$.*

Proof. The proof is similar to the proof of Lemma 7 since the topology of the network has no impact on the proof (note that the Case 1 is impossible on a ring). □

Lemma 17. *If $\gamma_0 \in \Gamma \setminus \Gamma_1$, then every execution starting from γ_0 contains the execution of a rule of a processor p such that $\varpi(p, \gamma_0) \geq 2$.*

Proof. Let $\gamma_0 \in \Gamma \setminus \Gamma_1$. Assume, for the sake of contradiction, that there exists an execution $\epsilon = \gamma_0\gamma_1 \ldots$ starting from γ_0 that contains no execution of a rule by any processor p that satisfies $\varpi(p, \gamma_0) \geq 2$. Since the daemon is strongly fair, this implies that $\exists k \in \mathbb{N}, \forall j \geq k$, p is not enabled in γ_j.

Let q be the neighbor of p satisfying $\omega(\{p, q\}, \gamma_k) = \varpi(p, \gamma_k)$. By hypothesis, q never executes a rule. Therefore, its clock value remains constant. Let us study the following cases:

Case 1: $\left|\left(c_q\right)^{\gamma_j} - \left(c_{\bar{q}}\right)^{\gamma_j}\right| \leq 1$
It follows that p is enabled for the rule **(*N*)** since $\left|\left(Inter(N_p)\right)^{\gamma_j}\right| \geq 2$. This contradicts the construction of k.

Case 2: $\left|\left(c_q\right)^{\gamma_j} - \left(c_{\bar{q}}\right)^{\gamma_j}\right| = 2$
It follows that p is enabled for the rule **(C_1)** since $\left(Inter(N_p)\right)^{\gamma_j} = \{h\}$ and $\left(c_p\right)^{\gamma_j} \neq h$ (because $\varpi(p, \gamma_j) = \varpi(p, \gamma_k) \geq 2$). This contradicts the construction of k.

Case 3: $\left|\left(c_q\right)^{\gamma_j} - \left(c_{\bar{q}}\right)^{\gamma_j}\right| \geq 3$
By the two previous contradictions, it is the only possible case. Since p is not enabled (by hypothesis), we obtain that:

$$\forall j \geq k, \begin{cases} \left(Inter(N_p)\right)^{\gamma_j} = \emptyset \\ \text{and} \\ \left(c_p\right)^{\gamma_j} \in \left\{ \left\lceil \dfrac{\left(c_q\right)^{\gamma_j} + \left(c_{\bar{q}}\right)^{\gamma_j}}{2} \right\rceil, \left\lfloor \dfrac{\left(c_q\right)^{\gamma_j} + \left(c_{\bar{q}}\right)^{\gamma_j}}{2} \right\rfloor \right\}. \end{cases}$$

Since the clock values of p and q are constants by hypothesis, we can deduce that the one of $\bar{q}$ remains also constant (because, in the contrary case, p becomes enabled, which contradicts the hypothesis). It follows: $\left(c_q\right)^{\gamma_j} < \left(c_p\right)^{\gamma_j} < \left(c_{\bar{q}}\right)^{\gamma_j}$ or $\left(c_q\right)^{\gamma_j} > \left(c_p\right)^{\gamma_j} > \left(c_{\bar{q}}\right)^{\gamma_j}$.

Since this reasoning holds for every processor on the ring, we can always label the nodes of any ring by $p_0, p_1, \ldots, p_n$ such that the following property is satisfied : $c_{p_0} < c_{p_1} < \cdots < c_{p_n}$.

But, the previous reasoning for processor c_{p_0} implies that we have: $c_{p_n} < c_{p_0} < c_{p_1}$. It is impossible to satisfy simultaneously these two inequalities, which proves the lemma. □

Lemma 18. *There exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$.*

Proof. The result follows directly from Lemmas 16 and 17. □

Finally, we can conclude:

Proposition 9. *$\mathcal{UFTSS}$ is a self-stabilizing **AU** under a locally central strongly fair daemon.*

Proof. Lemmas 13, 15 and 18 lead to the conclusion that $\mathcal{UFTSS}$ is a self-stabilizing **UAU** under a locally central strongly fair daemon. □

4.4.2. Proof of self-stabilization in spite of a crash

In this section, $\epsilon = \gamma_0, \gamma_1 \ldots$ denotes an execution of $\mathcal{UFTSS}$ such that a processor c is crashed in γ_0.

First, we prove the closure of our algorithm, then we prove the convergence property.

Lemma 19. *If there exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$, then $\gamma_{i+1} \in \Gamma_1$.*

Proof. This proof is similar to the proof of Lemma 13 since the fact that a processor is crashed or not does not modify the proof. □

Secondly, we are going to prove the liveness of our algorithm under these assumptions.

Lemma 20. *If $\gamma_0 \in \Gamma_1$, then every processor $p \neq c$ increments its clock in a finite time in ϵ.*

Proof. This proof is similar to the proof of Lemma 15. □

In the following we prove the convergence of our algorithm.

Lemma 21. *There exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$.*

Proof. This proof is similar to the proof of Lemma 18 since the fact that a processor is crashed or not does not modify the proof. □

Finally, we can conclude:

Proposition 10. *$\mathcal{UFTSS}$ is a self-stabilizing **AU** under a locally central strongly fair daemon even if a processor is crashed in the initial configuration.*

Proof. Lemmas 19–21 allows us to say that $\mathcal{UFTSS}$ is a self-stabilizing **UAU** under a locally central strongly fair daemon even if a processor is crashed in the initial configuration. Then, we can deduce the result. □

4.5. Conclusion

We are now in position to state our final result:

Proposition 11. *$\mathcal{UFTSS}$ is a $(0, 1)$-ftss **AU** on a chain or a ring under a locally central strongly fair daemon.*

Proof. This a direct consequence of Propositions 7–10. □

5. Concluding remarks

We presented the first study of FTSS protocols for dynamic tasks in asynchronous systems, and showed the intrinsic problems that are induced by the wide range of faults that we address. The combination of asynchrony and maintenance of liveness properties implies many impossibility results, and the deterministic protocol that we provided for one of the few remaining cases is optimal with respect to all impossibility results and containment measures. Then, we can observe that the results remain even if the weakly synchronized configuration definition is relaxed to allow neighbor clocks to be at most κ away from each other, for some constant κ.

Generalization: κ-**asynchronous unison**. In this paragraph, we briefly explain how to generalize the above results to a weaker problem. Assume that $\kappa \in \mathbb{N}^*$. In the κ-asynchronous unison problem (κ-**AU**), a drift of at most κ units is allowed between clocks of any two neighbors. Hence, the **AU** problem corresponds to the 1-**AU**.

Let us observe that a similar result to Lemma 1 holds in the case of κ-**AU**:

Lemma 22. *Let $\mathcal{A}$ be a universal (f, r)-FTSS algorithm for κ-**AU** (under an asynchronous daemon). Let γ be a configuration where a processor p with $c_p \geq \kappa$ has two neighbors q and q' such that: $c_q = c_p - \kappa$ and $c_{q'} = c_p + \kappa$. If p executes an action of $\mathcal{A}$ during the step $\gamma \longrightarrow \gamma'$, then this action does not modify the value of c_p. If $\mathcal{A}$ is also minimal, then the processor p is not enabled for $\mathcal{A}$ in γ.*

As Lemma 1 is the basis of proofs of Section 3, we can deduce that all impossibility results presented in Section 3 still hold in the case of κ-**AU** .

In order to solve the κ-**AU** problem in the remaining cases, we modify Algorithm $\mathcal{UFTSS}$ (see Section 4) in the definition of macro $poss(q)$ in the following way:

$$\forall q \in N_p, poss(q) = \left\{max\{c_q - \kappa, 0\}, max\{c_q - \kappa, 0\} + 1, \ldots, c_q, \ldots, c_q + \kappa - 1, c_q + \kappa\right\}.$$

This modified algorithm is a universal $(0, 1)$-FTSS κ-**AU** under a locally central strongly fair daemon on a chain or a ring (the proof is a simple generalization of the correctness proof of Section 4).

Open questions. An immediate future work is to generalize the possibility result (that assumes a central scheduler) to cope with a distributed scheduler, or extend the impossibility proof in that case. There also remains the open case of protocols that neither satisfy the minimality or the priority properties (see Table 1). We conjecture that at least one of those properties is necessary for the purpose of *deterministic* self-stabilization, yet none of those could be required for deterministic *weak* stabilization [19] (weak stabilization is a weaker property than self-stabilization since *existence* of execution reaching a legitimate configuration is guaranteed). As recent results [7] hint that weak-stabilizing solutions can be easily turned into *probabilistic* self-stabilizing ones, this raises the open question of the possibility of *probabilistic* FTSS for dynamic tasks in asynchronous systems.

Another possible extension of our work is the feasibility of FTSS solutions for other reactive tasks, such as *dining philosophers* and *mutual exclusion*. In the case of dining philosophers, [28] proposed a solution that can withstand transient (it is self-stabilizing) and Byzantine failures (with a containment radius of 2), so it is also a solution for tolerating transient and crash faults. However, even in the case of crash faults, a containment radius of 2 is also a lower bound [30] when the system is asynchronous. The same paper [28] shows that global tasks such as mutual exclusion cannot admit a constant radius fault-containing solution when both transient and Byzantine fault are considered. It would be interesting to investigate whether limiting the fault model to transient faults and process crashes permits to break this impossibility result.

References

[1] Efthymios Anagnostou, Vassos Hadzilacos, Tolerating transient and permanent failures (extended abstract), in: André Schiper (Ed.), WDAG, in: Lecture Notes in Computer Science, vol. 725, Springer, 1993, pp. 174–188.
[2] Michael Ben-Or, Danny Dolev, Ezra N. Hoch, Fast self-stabilizing byzantine tolerant digital clock synchronization, in: Rida A. Bazzi, Boaz Patt-Shamir (Eds.), PODC, ACM, 2008, pp. 385–394.
[3] Christian Boulinier, Franck Petit, Vincent Villain, When graph theory helps self-stabilization, in: Soma Chaudhuri, Shay Kutten (Eds.), PODC, ACM, 2004, pp. 150–159.
[4] Christian Boulinier, Franck Petit, Vincent Villain, Synchronous vs. asynchronous unison, in: Ted Herman, Sébastien Tixeuil (Eds.), Self-Stabilizing Systems, in: Lecture Notes in Computer Science, vol. 3764, Springer, 2005, pp. 18–32.
[5] Jean-Michel Couvreur, Nissim Francez, Mohamed G. Gouda, Asynchronous unison (extended abstract), in: ICDCS, 1992, pp. 486–493.
[6] Ajoy Kumar Datta, Maria Gradinariu (Eds.), Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006, Dallas, TX, USA, November 17–19, 2006, Proceedings, in: Lecture Notes in Computer Science, vol. 4280, Springer, 2006.
[7] Stéphane Devismes, Sébastien Tixeuil, 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.
[8] Edsger W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[9] Danny Dolev, Ezra N. Hoch, On self-stabilizing synchronous actions despite byzantine attacks, in: Andrzej Pelc (Ed.), DISC, in: Lecture Notes in Computer Science, vol. 4731, Springer, 2007, pp. 193–207.
[10] S. Dolev, Self-Stabilization, MIT Press, 2000.
[11] Shlomi Dolev, Possible and impossible self-stabilizing digital clock synchronization in general graphs, Real-Time Syst. 12 (1) (1997) 95–107.
[12] Shlomi Dolev, Jennifer L. Welch, Wait-free clock synchronization, Algorithmica 18 (4) (1997) 486–511.
[13] Shlomi Dolev, Jennifer L. Welch, Self-stabilizing clock synchronization in the presence of byzantine faults, J. ACM 51 (5) (2004) 780–799.
[14] Swan Dubois, Toshimitsu Masuzawa, Sébastien Tixeuil, The impact of topology on byzantine containment in stabilization, in: Proceedings of DISC 2010, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Boston, MA, USA, 2010.
[15] Swan Dubois, Toshimitsu Masuzawa, Sébastien Tixeuil, On byzantine containment properties of the min + 1 protocol, in: Proceedings of SSS 2010, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, New York, NY, USA, 2010.
[16] Swan Dubois, Maria Potop-Butucaru, Sébastien Tixeuil, Brief announcement: dynamic ftss in asynchronous systems: the case of unison, in: Proceedings of DISC 2009, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Elche, Spain, 2009.
[17] Michael J. Fischer, Nancy A. Lynch, Mike Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382.
[18] Ajei S. Gopal, Kenneth J. Perry, Unifying self-stabilization and fault-tolerance (preliminary version), in: PODC, 1993, pp. 195–206.
[19] Mohamed G. Gouda, The theory of weak stabilization, in: Ajoy Kumar Datta, Ted Herman (Eds.), WSS, in: Lecture Notes in Computer Science, vol. 2194, Springer, 2001, pp. 114–123.
[20] Mohamed G. Gouda, Ted Herman, Stabilizing unison, Inform. Process. Lett. 35 (4) (1990) 171–175.
[21] Ezra N. Hoch, Michael Ben-Or, Danny Dolev, A fault-resistant asynchronous clock function, in: Shlomi Dolev, Jorge Arturo Cobb, Michael J. Fischer, Moti Yung (Eds.), SSS, in: Lecture Notes in Computer Science, vol. 6366, Springer, 2010, pp. 19–34.
[22] Ezra N. Hoch, Danny Dolev, Ariel Daliot, Self-stabilizing byzantine digital clock synchronization, in: Datta and Gradinariu [6], pp. 350–362.
[23] Leslie Lamport, P.M. Melliar-Smith, Synchronizing clocks in the presence of faults, J. ACM 32 (1) (1985) 52–78.
[24] Qun Li, Daniela Rus, Global clock synchronization in sensor networks, in: INFOCOM, 2004.
[25] Toshimitsu Masuzawa, Sébastien Tixeuil, Bounding the impact of unbounded attacks in stabilization, in: Datta and Gradinariu [6], pp. 440–453.
[26] Toshimitsu Masuzawa, Sébastien Tixeuil, Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults, Int. J. Principl. Appl. Inform. Sci. Technol., PAIST 1 (1) (2007) 1–13.
[27] Jayadev Misra, Phase synchronization, Inform. Process. Lett. 38 (2) (1991) 101–105.
[28] Mikhail Nesterenko, Anish Arora, Tolerance to unbounded byzantine faults, in: 21st Symposium on Reliable Distributed Systems, SRDS 2002, IEEE Computer Society, 2002, p. 22.
[29] Marina Papatriantafilou, Philippas Tsigas, On self-stabilizing wait-free clock synchronization, Parallel Process. Lett. 7 (3) (1997) 321–328.
[30] Scott M. Pike, Paolo A.G. Sivilotti, Dining philosophers with crash locality 1, in: ICDCS, IEEE Computer Society, 2004, pp. 22–29.