Dynamic FTSS in Asynchronous Systems: the Case of Unison

Swan Dubois[*] Maria Gradinariu Potop-Butucaru[†] Sébastien Tixeuil[‡]

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 fault hit the system. FTSS protocols combine the best of both worlds since they are simultaneously fault-tolerant and self-stabilizing. To date, 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 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 many impossibility results for this difficult problem and propose a FTSS solution when the problem is solvable that exhibits optimal fault containment.

[*] Université Pierre & Marie Curie - Paris 6, LIP6-CNRS & INRIA Regal, France

[†] Université Pierre & Marie Curie - Paris 6, LIP6-CNRS & INRIA Regal, France

[‡] Université Pierre & Marie Curie - Paris 6, LIP6-CNRS & INRIA Grand Large, France

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* [14] 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 [15, 1, 2] as impossibility results are expected in many cases.

The seminal works of [1, 15] 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 [15], a general transformer is presented for synchronous systems, as well as positive results with failure detectors. The transformer of [15] was proved impossible to transpose to asynchronous systems in [2] 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 [21, 19]. 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.

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, that can bee seen as a *local* clock synchronization problem. In the unison problem [20], 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. Note that in synchronous completely connected systems where clocks have discrete time unit values, unison induces tight clock synchronization. Several self-stabilizing solutions exist for this problem [17, 6, 4, 5], 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. In [12, 22] 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 [13, 3], probabilistic FTSS protocols were proposed for up to one third of Byzantine processors, while in [18, 9] 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.

In this paper, we tackle the open issue of FTSS 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). We first extract 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 whenever incrementing the clock value does not

	Unfair	Weakly fair			Strongly fair		
		Minimal	Priority	Neither	Minimal	Priority	Neither
$f = 1$, $\Delta \geq 3$	Imp. (Prop.2)	Imp. (Prop.3)	Imp. (Prop.4)	??	Imp. (Prop.5)	Imp. (Prop.6)	??
$f = 1$, $\Delta \leq 2$					Pos. (Prop.11)		
$f \geq 2$	Imp. (Prop.1)						

Table 1: Summary of results

break the local safety predicate between neighbors, the clock value is actually incremented in a finite number of activations, even when 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), FTSS unison is impossible. When the scheduling is *weakly fair* (a processor that is continuously enabled is eventually activated), then it is impossible to solve FTSS unison by a protocol that satisfies either minimality or priority. The case of *strongly fair* scheduling (a processor that is activated 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 and the scheduling is central (*i.e.* a single processor is activated at any time).

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

2 Model, definitions and notations

We consider a 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*). Every processor p can distinguish its neighbors and locally label them, and we assume that 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 in 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 (the processor rule is then *enabled*). The state of a processor is defined by the 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.

Processor p is *enabled* in $\gamma \in \Gamma$ if and only if at least one rule is enabled for p in γ. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\rightarrow$, on 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, $\forall i \geq 0, \gamma_i \rightarrow \gamma_{i+1}$ ($(\gamma_i, \gamma_{i+1}) \in \rightarrow$ is called 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. 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 neighbors 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, C^* denotes the set of correct processors in $C \subset V$. As crashed processors cannot be distinguished from slow ones by their neighbors, we assume that variables of crashed processors are always readable. We now recall definitions about self-stabilization and fault-tolerant self-stabilization.

Definition 1 (self-stabilization [8]) *Let $\mathcal{T}$ be a task, and $\mathcal{S}_\mathcal{T}$ a specification of $\mathcal{T}$. A protocol $\mathcal{P}$ is self-stabilizing for $\mathcal{S}_\mathcal{T}$ 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 satisfies $\mathcal{S}_\mathcal{T}$.*

Definition 2 ($(f, r)-$containment [21]) *Let $\mathcal{T}$ be a task, and $\mathcal{S}_\mathcal{T}$ a specification of $\mathcal{T}$. A configuration $\gamma \in \Gamma$ is $(f, r)-$contained for specification $\mathcal{S}_\mathcal{T}$ if and only if, given at most f crashed processors, every execution starting from γ, always satisfies $\mathcal{S}_\mathcal{T}$ on the sub-graph induced by processors which are at distance r or more from any crashed processor.*

Definition 3 (fault-tolerant self-stabilization (FTSS) [1, 15]) *Let $\mathcal{T}$ be a task, and $\mathcal{S}_{\mathcal{T}}$ a specification of $\mathcal{T}$. A protocol $\mathcal{P}$ is fault-tolerant and self-stabilizing with radius r for f crashed processors (and denoted by $(f, r) - ftss$) for specification $\mathcal{S}_{\mathcal{T}}$ if and only if, given at most f crashed processors, for every configuration $\gamma_0 \in \Gamma$, for every execution $\epsilon = \gamma_0\gamma_1 \dots$, there exists a finite prefix $\gamma_0\gamma_1 \dots \gamma_l$ of ϵ such that γ_l is $(f, r)-$contained for specification $\mathcal{S}_{\mathcal{T}}$.*

Problem and specifications In the following, H_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). 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.

Definition 4 (weakly synchronized configurations Γ_1^*) *Let be $\gamma \in \Gamma$. We say that γ is weakly synchronized (denoted by $\gamma \in \Gamma_1^*$) if and only if :*

$$\forall p \in V^*, \forall q \in N_p^*, |H_p - H_q| \leq 1$$

Definition 5 (uniform weakly synchronized configurations Γ_1) *Let be $\gamma \in \Gamma$. We say that γ is uniformly weakly synchronized (denoted by $\gamma \in \Gamma_1$) if and only if :*

$$\forall p \in V, \forall q \in N_p, |H_p - H_q| \leq 1$$

Remark 1 *If no processor is crashed, we have: $\Gamma_1 = \Gamma_1^*$, on the contrary case, we have: $\Gamma_1 \subsetneq \Gamma_1^*$*

For example, if $G = (V, E)$ with $V = \{p_0, p_1, p_2\}$ and $E = \{\{p_0, p_1\}, \{p_1, p_2\}\}$, then configuration γ defined by $H_{p_0} = 0$, $H_{p_1} = H_{p_2} = 2$, and where p_0 is crashed satisfies $\gamma \in \Gamma_1^*$ and $\gamma \notin \Gamma_1$.

We now specify the two variants of our problem (depending whether safety property is extended to crashed processors):

Specification 1 (asynchronous unison – AU)
*Let be $\gamma_0 \in \Gamma$. An execution $\epsilon = \gamma_0\gamma_1 \dots$ starting from γ_0 is a legitimate execution for **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 infinitely often in ϵ.*

Specification 2 (uniform asynchronous unison – UAU)
*Let be $\gamma_0 \in \Gamma$. An execution $\epsilon = \gamma_0\gamma_1 \dots$ starting from γ_0 is a legitimate execution for **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 infinitely often in ϵ.*

Remark 2 *Note that:*

- *An algorithm which complies to the second specification complies to the first (the converse is not true).*
- *These two specifications do not forbid decrementation of clocks.*

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.

Definition 6 (minimality) *A unison is* minimal *if and only if the set of variables of each processor is reduced to its clock.*

Remark 3 *As the execution of a rule by a processor always modifies its state, every execution of rule by a processor by a minimal unison modifies its clock value.*

Definition 7 (priority) *A unison is* priority *if and only if it satisfies the following property: if there exists a processor p such that $\forall q \in N_p, (H_q = H_p$ or $H_q = H_p + 1)$ in a configuration γ_i, then there exists a fragment of execution $\epsilon = \gamma_i \dots \gamma_{i+k}$ such that:*

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

Remark 4 *If a priority unison is also minimal, then $k = 1$ since every execution of a rule by a processor modifies its clock value.*

3 Impossibility results

In this section we present a broad class of impossibility results related to the FTSS unison. For the sake of the generality we assume the most constrained scheduler (the central one). Additionally we assume each processor has an infinite memory.

3.1 Preliminaries

First, we introduce two preliminary results which show that in any execution of a $(f, r)-$ftss algorithm for **AU** (under an asynchronous daemon) a processor can not modify its clock value if it has two neighbors q and q' such that: $H_q = H_p - 1$ and $H_{q'} = H_p + 1$.

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

Proof. Let $\mathcal{A}$ be a $(f, r)-$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 $H_p \geq 1$) which has two neighbors q and q' such that: $H_q = H_p - 1$ and $H_{q'} = H_p + 1$.

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 H_p. Note that H_q and $H_{q'}$ are identical in γ and γ'. Let α be the value of H_p in γ and α' be the value of H_p in γ'. $alpha$ and $alpha'$ verify one of the two following relations:

Case 1: $\alpha < \alpha'$.

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

Case 2: $\alpha' < \alpha$.

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

In the two above cases, $\gamma' \notin \Gamma_1$, hence the safety property of $\mathcal{A}$ is not verified. □

Lemma 2 *Let $\mathcal{A}$ be a $(f,r)-$ftss algorithm for minimal $\boldsymbol{AU}$ (under an asynchronous daemon). Let γ be a configuration in which a processor p (such that $H_p \geq 1$) has two neighbors q and q' such that: $H_q = H_p - 1$ and $H_{q'} = H_p + 1$. Processor p is not enabled for $\mathcal{A}$ in γ.*

Proof. This is a direct consequence of Lemma 1. □

3.2 With respect to the number of crashed processors

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

Proof. Let r be a natural number. Let $\mathcal{A}$ be a $(2,r)-$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 $\forall i \in \{0, \ldots, 2(r+1)\}, H_{p_i} = i$ (all the other variables can 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. However, p_{r+1} must verify the specification of the problem since the nearest crashed processor is at r hops away. This contradicts the liveness property of $\mathcal{A}$. □

3.3 With respect to unfair daemon

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

Proof. Let r be a natural number. Assume that there exists an $(1,r)-$ftss algorithm $\mathcal{A}$ for **AU** under an unfair daemon. Consider a network, G, of diameter greater than $2r+2$ [1]. Let p be a processor of G. Since the daemon is unfair, it can choose to never activate p unless this processor becomes the only enabled processor of G.

Assume that there exists a configuration γ such that no processor is crashed and in which p is the only enabled processor of the network. The asynchronism assumption makes this configuration indistinguishable from γ', the same configuration in which p is crashed. We assumed that in γ no other processor but p is enabled. Consequently, the network is starved in γ'. This contradicts the liveness property of $\mathcal{A}$, hence no such configuration γ exists.

Since there exists no configuration in which 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 $\mathcal{A}$. □

[1] At least one processor verifies the specification of the **AU** problem

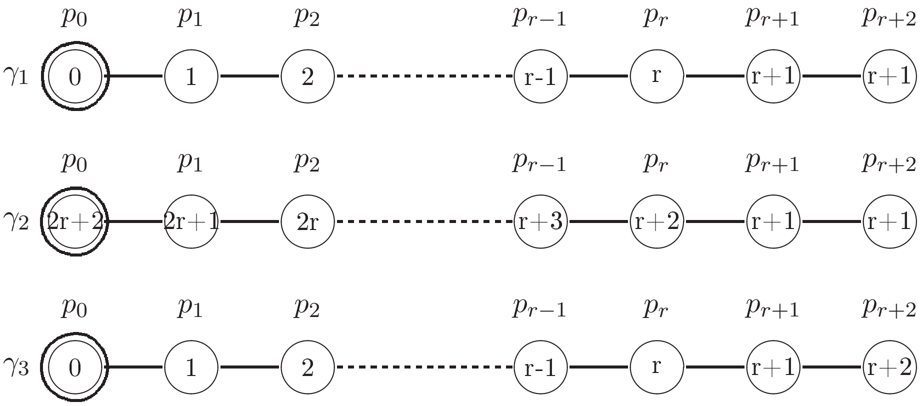


Figure 1: The three configurations used in the proof of Lemma 3 (the numbers represent clock values and the double circles represent crashed processors).

3.4 With respect to weakly fair daemon

In this section we prove there exists no $(1, r)-$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 an algorithm $\mathcal{A}$ which 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 $H_p = H_{p'}$ (proved in Lemma 3 formally stated below). Then, we show that $\mathcal{A}$ starves the network reduced to a two-correct-processor chain in which all clock values are identical (see Proposition 3).

Lemma 3 *If there exists an algorithm $\mathcal{A}$ which 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 $H_p = H_{p'}$.*

Proof. Let r be a natural number. Let $\mathcal{A}$ be a $(1, r)-$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 Figure 1):

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

By Lemma 2, processors from p_1 to p_r are not enabled in such configurations (and remain not enabled until one of the processors within $p_0 \ldots p_{r+1}$ execute 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 absurd. 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 to $r+2$.
Assume w.r.g. p_{r+2} is the only activated processor hence its clock takes the value $r+2$. The following cases are possible in the new configuration:

Case 1.1: p_{r+2} is not enabled.
If the 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 after execution its clock modifies to $r+1$.
Let ϵ be an execution starting from γ_1 in which 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 $H_{p_{r+2}} = r+2$, hence this processor is never infinitely enabled. Overall, this execution is allowed by the weakly fair scheduler, however it starves p_{r+1}, which contradicts the liveness property of $\mathcal{A}$.

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

Case 2 : The enabled rule for p_{r+2} modifies its clock into r.
Assume w.r.g. p_{r+2} is the only activated processor and after its execution the new configuration verifies one of the the following cases:

Case 2.1 : p_{r+2} is not enabled.
If the 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 its clock modifies to $r+1$.
Let ϵ be an execution starting from γ_2 which 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 $H_{p_{r+2}} = r$, so this processor is never infinitely enabled. In conclusion, this execution verifies 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 execution of its enabled rule modifies its clock to $r+2$.
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 to r when $H_{p_{r+2}} = r+2$ and $H_{p_{r+1}} = r+1$.
2. p_{r+2} is enabled for modifying its clock value to $r+2$ when $H_{p_{r+2}} = r$ and $H_{p_{r+1}} = r+1$.

Let ϵ be an execution starting from γ_3 which 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 $p_0 \ldots p_r$ are never enabled. Note that p_{r+1} is not enabled when $H_{p_{r+2}} = r+2$, so this processor is never infinitely enabled. In conclusion, this execution verifies 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 $(1,r)-$ftss algorithm for* minimal ***AU*** *under a weakly fair daemon.*

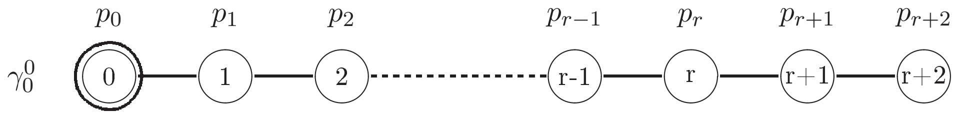


Figure 2: Initial configurations used in the proof of Proposition 4 (the numbers represent clock values and the double circles represent crashed processors).

Proof. Let r be a natural integer. Assume there exists a $(1, r)-$ftss algorithm $\mathcal{A}$ for the minimal **AU** under a weakly fair daemon. By Lemma 3, an arbitrary processor p is not enabled for $\mathcal{A}$ if it has only one neighbor p' and if $H_p = H_{p'}$.

Let G be a network reduced to a chain of 2 processors p and p'. Let γ be a configuration of G in which $H_p = H_{p'}$ and no crashed processor. Notice that no processor is enabled in γ which contradicts the liveness property of $\mathcal{A}$ and proves the result. □

The second main result of this section is that there exists no $(1, r)-$ftss algorithm for priority **AU** under a weakly fair daemon for any natural number r (see Proposition 4).

To prove this result by contradiction we construct an execution (allowed by a weakly fair scheduler) starting from the configuration γ_0^0 shown in Figure 2. We prove that this execution starves p_{r+1} which contradicts the liveness property of the algorithm.

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

Proof. Let r be a natural number. Assume that there exists a $(1, r)-$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: $p_0, p_1, \ldots, p_{r+2}$. Let γ_0^0 be a configuration and p_0 crashed and $\forall i \in \{0, \ldots, r + 2\}, H_{p_i} = i$ (See Figure 2). Note that all the other variables can have any value.

We construct a fragment of execution $\epsilon_0' = \gamma_0^0\gamma_1^0\gamma_2^0 \ldots \gamma_{r+1}^0$ starting from γ_0^0 such that $\forall i \in \{0, 1, \ldots, r\}$, the step $\gamma_i^0 \to \gamma_{i+1}^0$ contains only the action of p_{i+1} if p_{i+1} is enabled. By Lemma 1, this fragment does not modify the clock value of processors in $p_0 \ldots p_{r+1}$.

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.
In the sequel we distinguish following cases:

Case 2.1: The execution of a rule by p_{r+2} in γ_{r+1}^0 doesn't modify its clock value.
Let ϵ_0'' be $\gamma_{r+1}^0\gamma_{r+2}^0$ in which the step $\gamma_{r+1}^0 \to \gamma_{r+2}^0$ contains only the execution of a rule by p_{r+2}.

Case 2.2: The execution of a rule by p_{r+2} in γ_{r+1}^0 modifies its clock value.
The safety property of $\mathcal{A}$ implies that the clock of p_{r+2} takes the value r or $r + 1$.

Case 2.2.1: The execution of a rule by p_{r+2} in γ_{r+1}^0 modifies its clock value into $r + 1$.
Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon_0'' = \gamma_{r+1}^0\gamma_{r+2}^0 \ldots \gamma_{r+k}^0$ which contains only actions of p_{r+2} such that (i) in the steps from γ_{r+2}^0 to γ_{r+k-1}^0 the clock value of p_{r+2} is not modified while (ii) in the step $\gamma_{r+k-1}^0 \to \gamma_{r+k}^0$ the clock value of p_{r+2} is incremented.

Case 2.2.2: The execution of a rule by 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 $\epsilon_a = \gamma^0_{r+1}\gamma^0_{r+2}\dots\gamma^0_{r+k}$ which contains only actions of p_{r+2} such that (i) 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 (ii) in the step $\gamma^0_{r+k-1} \rightarrow \gamma^0_{r+k}$ 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 $\epsilon_b = \gamma^0_{r+k}\gamma^0_{r+k+1}\dots\gamma^0_{r+j}$ which contains only actions of p_{r+2} such that (i) in the steps from γ^0_{r+k+1} to γ^0_{r+j-1} the clock value of p_{r+2} is not modified and (ii) in the step $\gamma^0_{r+j-1} \rightarrow \gamma^0_{r+j}$ 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) verifies: the values of the network clocks are identical to those in γ^0_0 (the others variables may have changed). Then, we can reiterate the reasoning and obtain a fragment of execution $\epsilon_1, \epsilon_2 \dots$ (respectively starting from $\gamma^1_0, \gamma^2_0, \dots$) that verifies the same property.

We finally obtain an execution $\epsilon = \epsilon_0\epsilon_1\dots$ which verifies:

- 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 verifies the weakly fair scheduling.
- The clock of the processor p_{r+1} never changes (whereas $d(p_0, p_{r+1}) = r+1$).

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

3.5 With respect to strongly fair daemon

In this section we prove that there exists no $(1,r)-$ftss algorithm for minimal or priority **AU** under a strongly fair daemon if the degree of the network is greater or equal to 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 $H_q = r+1$ and if $|H_p - H_q| \leq 1$, then p is enabled in any $(1,r)-$ftss algorithm for minimal **AU** (see Lemma 4). Then we construct a strongly fair infinite execution which starves a processor more than r hops away from a crashed processor. This execution contradicts the liveness property of the **AU** problem (see Proposition 5).

Lemma 4 *Let $\mathcal{A}$ a $(1,r)-$ftss algorithm for minimal **AU**. If a processor p has only one neighbor q such that $H_q = r+1$ and if $|H_p - H_q| \leq 1$, then p is enabled in $\mathcal{A}$.*

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

1. If $H_p = r$, then p is enabled for at least one rule of $\mathcal{A}$. Otherwise, the network reduced to the chain $p_0, \dots, p_r, q, p$ in the configuration γ_1 defined by $\forall i \in \{0, \dots, r\}, H_{p_i} = 2r+2-i$, $H_q = r+1$, $H_p = r$ where p_0 is crashed (see Figure 3) is starved since no correct processor is enabled (by Lemma 2).
2. If $H_p = r+1$, then p is enabled for at least one rule of $\mathcal{A}$. Otherwise, the network reduced to the chain q, p in the configuration γ_2 defined by $H_q = H_p = r+1$ and in which no processor is crashed (see Figure 3) is starved since no correct processor is enabled.

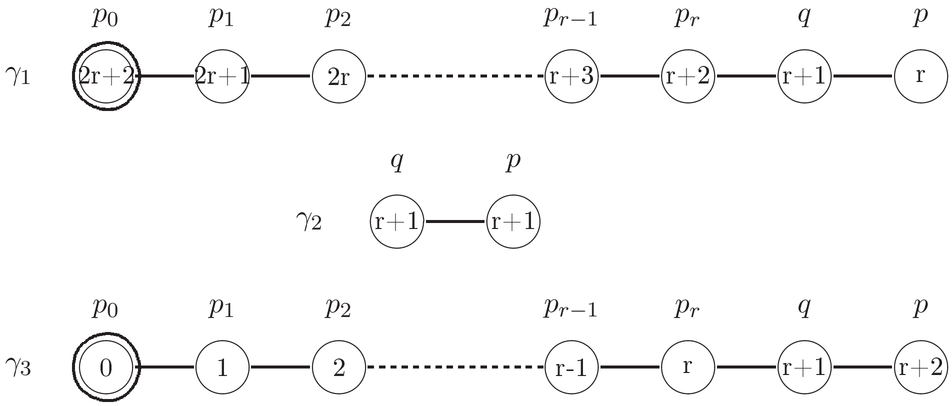


Figure 3: The three configurations used in the proof of Lemma 4 (the numbers represent clock values and the double circles represent crashed processors).

3. If $H_p = r+2$, then p is enabled for at least one rule of $\mathcal{A}$. Otherwise, the network reduced to the chain $p_0, \ldots, p_r, q, p$ in the configuration γ_3 defined by $\forall i \in \{0, \ldots, r\}, H_{p_i} i$, $H_q = r+1$, $H_p = r+2$ and p_0 crashed (see Figure 3) is starved since no correct processor is enabled (by Lemma 2).

□

Proposition 5 *For any natural number r, there exists no $(1, r)-$ftss algorithm for* minimal ***AU*** *under a strongly fair daemon if the graph modeling the network has a degree greater or equal to 3.*

Proof. Let r be a natural number. Assume that there exists a $(1, r)-$ftss algorithm $\mathcal{A}$ for the minimal **AU** under a strongly fair daemon in a network with a degree greater or equal to 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, q and q' must behave identically if they have the same clock value (in this case, their local configurations are identical). If $H_{p_{r+1}} = r+1$ and $|H_{p_{r+1}} - H_q| \leq 1$, there exists three local configurations for q: (1) $H_q = r$, (2) $H_q = r+1$ or (3) $H_q = r+2$ (the same property holds for q').

By Lemma 4, Processor q (respectively q') is enabled in any configuration in which $H_{p_{r+1}} = r+1$ and $|H_{p_{r+1}} - H_q| \leq 1$ (respectively $|H_{p_{r+1}} - H_{q'}| \leq 1$). Moreover, in this case, the enabled rule for q (respectively q') modifies its clock into a value in $\{r, r+1, r+2\} \setminus H_q$ (respectively $\{r, r+1, r+2\} \setminus H_{q'}$) 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 4), $\mathcal{A}$ can only allow 2 moves. Hence, there exists 8 possible moves for $\mathcal{A}$. Let 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 $H_q = r$, $H_q = r+1$, and $H_q = r+2$ 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: $\forall i \in \{0, \ldots, r+1\}, H_{p_i} = 2r+2-i$, $H_q = r+1$

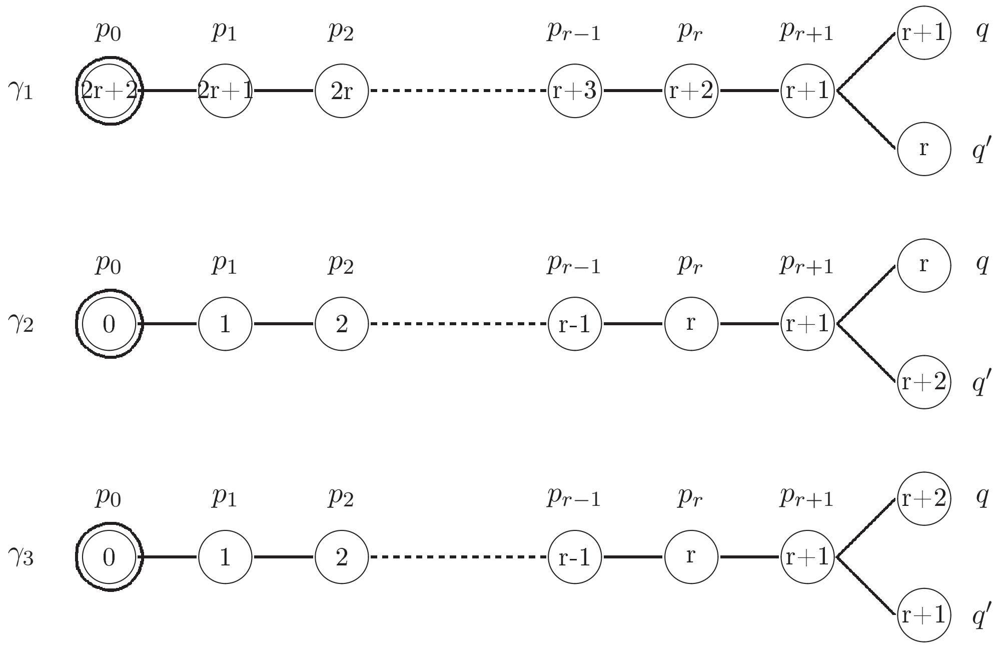


Figure 4: The three configurations used in the proof of Proposition 5 (the numbers represent clock values and the double circles represent crashed processors).

and $H_{q'} = r$ and p_0 crashed (see Figure 4). Note that only q and q' are enabled (by Lemma 2). Assume q executes. Hence, its clock takes the value r. By Lemma 2, only q and q' are enabled. Assume now that q' executes. Its clock takes the value $r+1$. This configuration is identical to γ_1 (since processors are anonymous), we can repeat the above reasoning in order to obtain an infinite execution in which processors $p_1, \ldots, p_{r+1}$ are never enabled (see Figure 5 for an illustration when $r = 1$).

Case 2: $(r+1, r+2, r)$

Let γ_2 be the configuration of G defined by: $\forall i \in \{0, \ldots, r+1\}, H_{p_i} i$, $H_q = r$ and $H_{q'} = r+2$ and p_0 crashed (see Figure 4). Note that only q and q' are enabled (by Lemma 2). Assume q executes. Its clock takes the value $r+1$. By Lemma 2, only q and q' are enabled. Assume q executes its rule again. Its clock takes the value $r+2$. By Lemma 2, 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 in which 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: $\forall i \in \{0, \ldots, r+1\}, H_{p_i} = i$, $H_q = r+2$ and $H_{q'} = r+1$ and in which p_0 is crashed (see Figure 4). Note that only q and q' are enabled (by Lemma 2). Assume q' executes its rule. Its clock takes the value $r+2$. By Lemma 2, only q and q' are enabled. Assume now that q executes its rule. Its clock takes the

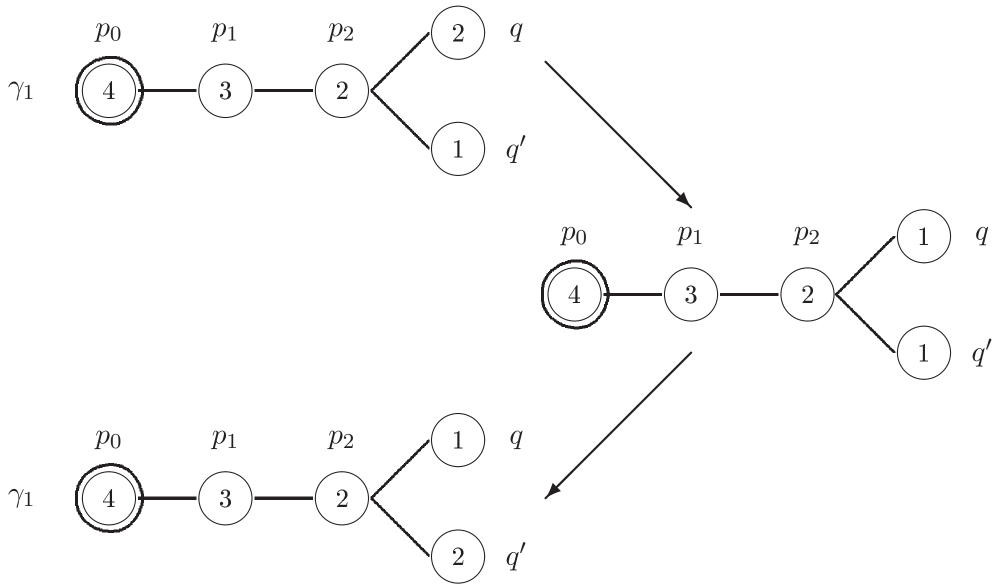


Figure 5: Example of the execution constructed in case 1 of Proposition 5 when $r = 1$ (the numbers represent clock values and the double circles represent crashed processors).

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 in which processors in $p_1, \dots, p_{r+1}$ 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 2). Assume q executes its rule. Its clock takes the value $r + 2$. By Lemma 2, 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 in which processors $p_1, \dots, p_{r+1}$ 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 2). Assume q executes its rule. Its clock takes the value $r + 2$. By Lemma 2, only q and q' are enabled. Assume q' executes its rule. Its clock takes the value $r + 1$. By Lemma 2, 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 in which processors $p_1, \dots, p_{r+1}$ are never enabled.

Case 8: $(r + 2, r + 2, r + 1)$

The proof is similar to the case 4.

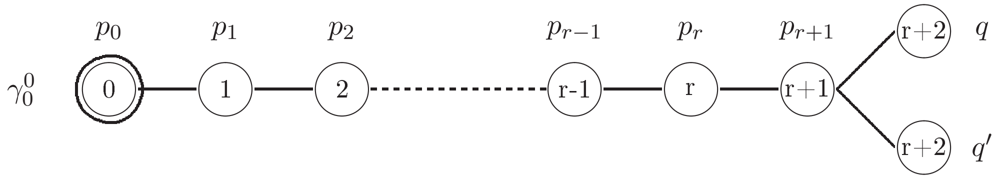


Figure 6: The initial configuration for the proof of Proposition 6 (the numbers represent clock values and the double circles represent crashed processors).

Overall, we can construct an infinite execution in which 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 verifies 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. □

The second main result of this section is that there exists no $(1, r)-$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 greater or equal to 3. (see Proposition 6).

To prove this result we assume the contrary and we construct an execution starting from the configuration γ_0^0 of Figure 6 verifying the strongly fair scheduling which starves p_{r+1}, that contradicts the liveness of the algorithm.

Proposition 6 *For any natural number r, there exists no $(1, r)-$ftss algorithm for* priority ***AU*** *under a strongly fair daemon if the graph modeling the network has a degree greater or equal to 3.*

Proof. Let r be a natural number. Assume that there exists a $(1, r)-$ftss algorithm $\mathcal{A}$ for priority **AU** under a strongly fair daemon even if the graph modeling the network has a degree greater or equal to 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: $\forall i \in \{0, \ldots, r+1\}, H_{p_i} = i$, $H_q = H_{q'} = r+2$ and p_0 crashed (see Figure 6). Note that, for all execution ϵ starting from γ_0^0, the processors q and q' are allowed to modify their clocks in a finite time (otherwise the network would be starved following Lemma 1).

Let $\epsilon_a^0 = \gamma_0^0\gamma_1^0 \ldots \gamma_k^0$ be a fragment of execution 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. it contains no modification of clock values
3. γ_k^0 is the first configuration in which q or q' are enabled for the modification of their clock value.

We consider the following scheduling scenario: in each step in ϵ_a^0 is executed the least recently executed processor in the set of 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 H_q should be modified to either r or $r+1$.

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

Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon_{b1}^0 = \gamma_k^0\gamma_{k+1}^0 \ldots \gamma_{k+r}^0$ which 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 $\gamma_{k+r-1}^0 \rightarrow \gamma_{k+r}^0$ the clock value of q is incremented.

Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon_{b2}^0 = \gamma_{k+r}^0\gamma_{k+r+1}^0 \dots \gamma_{k+j}^0$ which 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 $\gamma_{k+j-1}^0 \rightarrow \gamma_{k+j}^0$ the clock value of q is incremented.

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

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

Since $\mathcal{A}$ is a priority unison, there exists by definition a fragment of execution $\epsilon_b^0 = \gamma_k^0\gamma_{k+1}^0 \dots \gamma_{k+r}^0$ which 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 $\gamma_{k+r-1}^0 \rightarrow \gamma_{k+r}^0$ the clock value of q increments.

If q' is enabled in the last configuration of ϵ_b^0 [2], 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 $\epsilon = \epsilon^0\epsilon^1 \dots$ which 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, that implies that an infinitely often enabled processor executes a rule in a finite time). This execution verifies 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. □

4 A protocol for chains and rings

In the following we consider some possibility results related to the asynchronous unison on chains and rings (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 for chains and rings. The proposed algorithm is both minimal and priority.

4.1 Algorithm description

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 is "locally synchronized", it modifies its clock value in a finite time in order to preserve this property. If a processor is not synchronized with at least one of its neighbors, it makes a correction in a finite time in order to correct its clock value. More precisely, each processor p has only one variable: its clock denoted by H_p. At each step, every processor p computes a set of *possible clock values*, *i.e.* the set of clock values which have a drift of at most

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

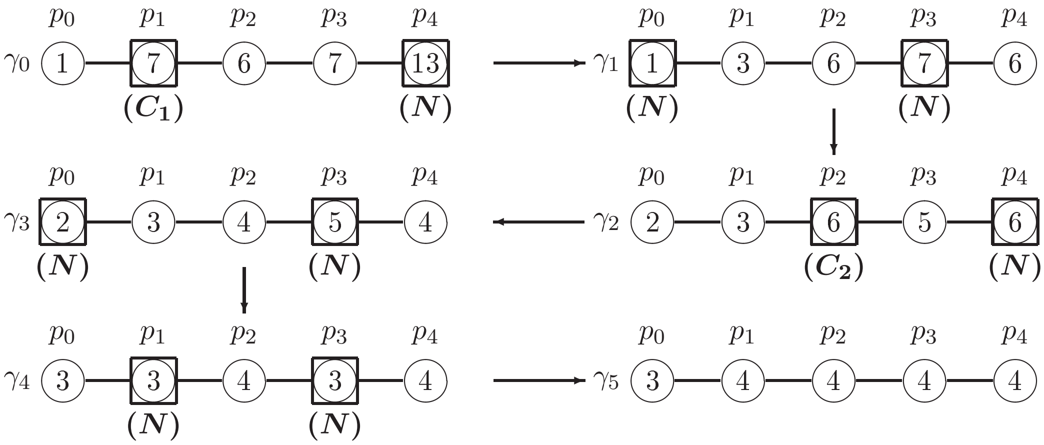


Figure 7: An example of execution of $\mathcal{UFTSS}$ on a chain with no crash (the numbers represent clock values and squared processors in γ_i executed the indicated rule during the step $\gamma_i \longrightarrow \gamma_{i+1}$).

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:

- $|Inter(N_p)| = 0$: 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.
- $|Inter(N_p)| = 1$: 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.
- $|Inter(N_p)| \geq 2$: 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 $H_p + 1 \in Inter(N_p)$, then H_p is modified to $H_p + 1$, otherwise H_p is modified to $min\{Inter(N_p)\}$.

Note that our correction rules use the average instead of maximum or minimum (which are frequently used in the literature, see e.g. [9, 11, 12, 22]) in order to not favors the clock value of a particular neighbor. That is, the chosen neighbor may be crashed and prevent the system to reach the synchronization.

The detailed description of our solution is proposed in Algorithm 1. In order to better understand our algorithm Figures from 7 to 10 propose some toy examples.

4.2 Correction Proof roadmap

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).*

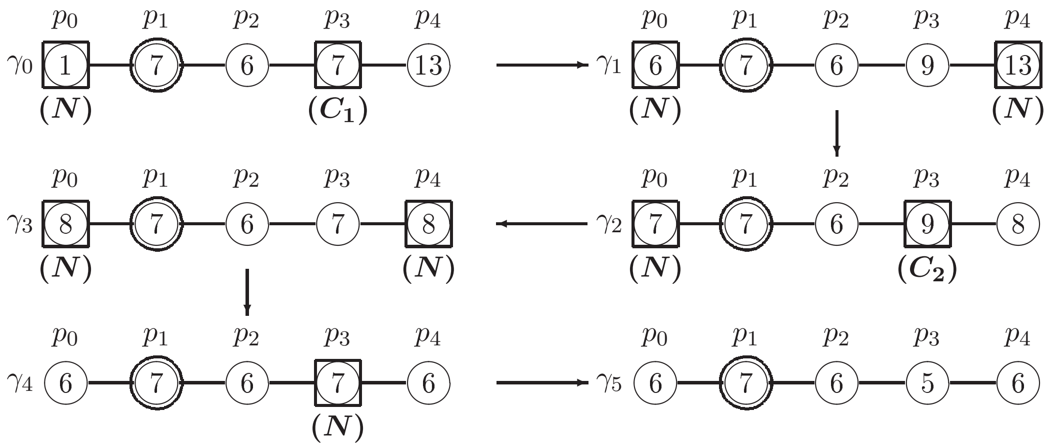
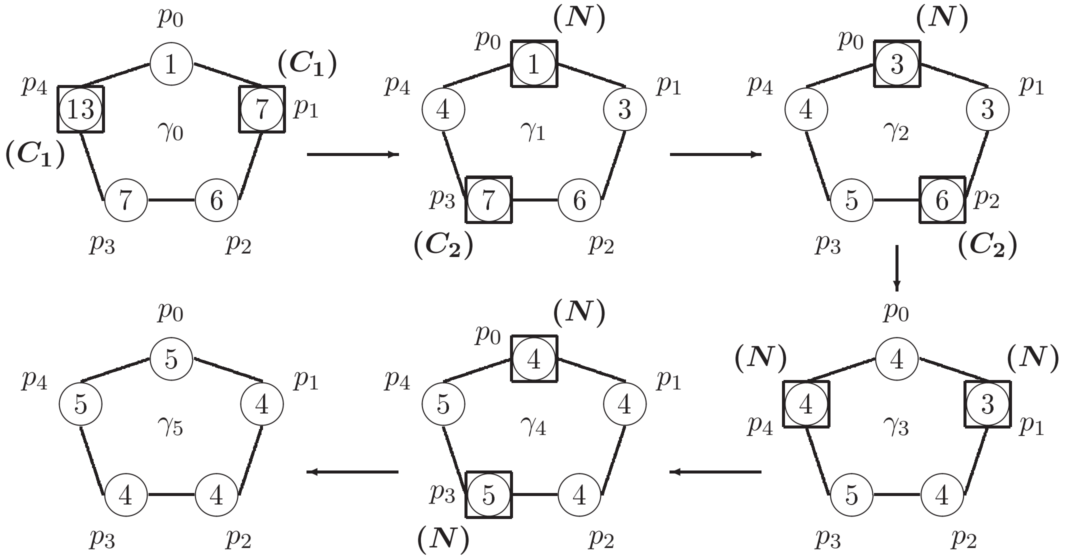


Figure 8: An example of execution of $\mathcal{UFTSS}$ on a chain with a crash (the numbers represent clock values, the double circles represent crashed processors and squared processors in γ_i executed the indicated rule during the step $\gamma_i \longrightarrow \gamma_{i+1}$).



γ_0: p_0 1, p_1 7 (C1), p_2 6, p_3 7, p_4 13 (C1)
γ_1: p_0 1 (N), p_1 3, p_2 6, p_3 7 (C2), p_4 4
γ_2: p_0 3 (N), p_1 3, p_2 6 (C2), p_3 5, p_4 4
γ_3: p_0 4, p_1 3 (N), p_2 4, p_3 5, p_4 4 (N)
γ_4: p_0 4 (N), p_1 4, p_2 4, p_3 5 (N), p_4 5
γ_5: p_0 5, p_1 4, p_2 4, p_3 4, p_4 5

Figure 9: 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 $\gamma_i \longrightarrow \gamma_{i+1}$).

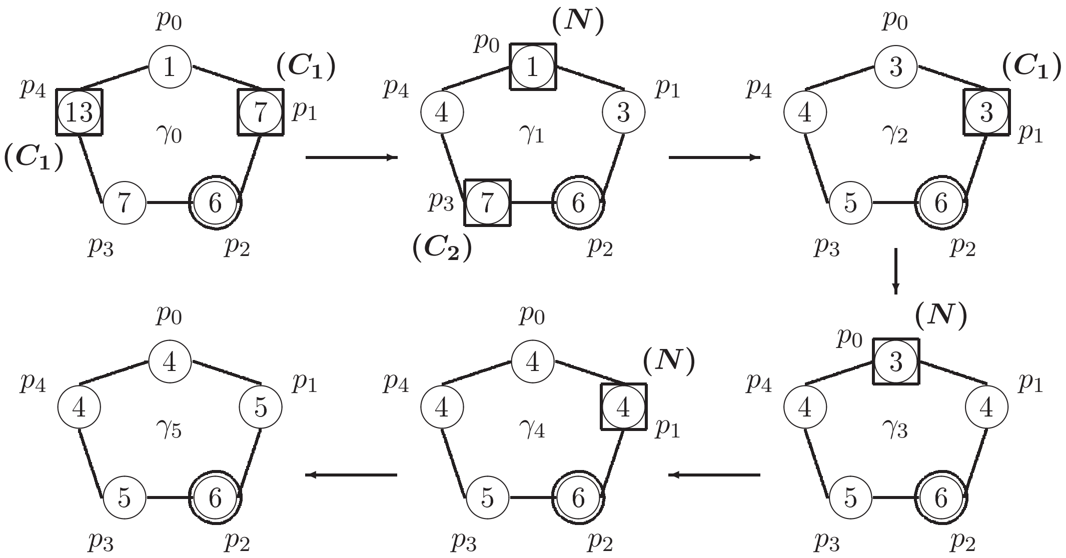


Figure 10: An example of execution of $\mathcal{UFTSS}$ on a ring with a crash (the numbers represent clock values, the double circles represent crashed processors and squared processors in γ_i executed the indicated rule during the step $\gamma_i \longrightarrow \gamma_{i+1}$).

Algorithm 1 ($\mathcal{UFTSS}$): **AU** (minimal and priority) $(1,0)$-ftss.

Data:

- N_p: set of neighbors of p.

Variable:

- H_p: natural integer representing the clock of the processor.

Macros:

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

Rules:

/* Normal rule */

$(\boldsymbol{N}) :: |Inter(N_p)| \geq 2 \longrightarrow H_p := next\,(Inter(N_p), H_p)$

/* Correction rules */

$(\boldsymbol{C_1}) :: (|Inter(N_p)| = 0) \wedge \left(H_p \neq \left\lceil \frac{\sum_{q \in N_p} H_q}{|N_p|} \right\rceil\right) \wedge \left(H_p \neq \left\lfloor \frac{\sum_{q \in N_p} H_q}{|N_p|} \right\rfloor\right) \longrightarrow H_p := \left\lfloor \frac{\sum_{q \in N_p} H_q}{|N_p|} \right\rfloor$

$(\boldsymbol{C_2}) :: (Inter(N_p) = \{h\}) \wedge (H_p \neq h) \longrightarrow H_p := h$

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

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 the sequel 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 $\mathcal{UFTSS}$ verifies 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 verify: $\gamma' \in \Gamma_1$, see respectively Lemma 5, 11, 14, and 20).

   The idea of the proof is as follows: we first prove that only the normal rule is enabled in a such configuration and then, we show that this rule respects the "locally synchronization" property.
2. Secondly, we prove that $\mathcal{UFTSS}$ verifies 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 Lemma 7, 12, 16, and 21).

   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 Lemma 10, 13, 19, and 22).

   In order to complete the proof we studying 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, $\epsilon = \gamma_0, \gamma_1 \dots$ denotes an execution of $\mathcal{UFTSS}$ in which there is no crash.

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

Lemma 5 *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 $\forall p \in V$, $(Inter(N_p))^{\gamma_i} \neq \emptyset$ and then the rule $\boldsymbol{(C_1)}$ is not enabled in γ_i. Assume rule $\boldsymbol{(C_2)}$ is enabled in γ_i. This implies that $(Inter(N_p))^{\gamma_i} = \{h\}$ and that $(H_p)^{\gamma_i} \neq h$. Then, we have $\gamma_i \notin \Gamma_1$ (since if $(H_p)^{\gamma_i} \neq h$, then the following holds: $\exists q \in N_p, |\,(H_p)^{\gamma_i} - (H_q)^{\gamma_i}\,| \geq 2$). This contradiction allows us to conclude that the enabled processors in γ_i are only enabled for rule $\boldsymbol{(N)}$.

Let p be a processor which 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: $\exists q \in N_p, |\,(H_p)^{\gamma_{i+1}} - (H_q)^{\gamma_{i+1}}\,| \geq 2$. By construction of rule $\boldsymbol{(N)}$, $(H_p)^{\gamma_{i+1}} \in (Inter(N_p))^{\gamma_i}$. By construction, $(Inter(N_p))^{\gamma_i} \subseteq \{(H_q)^{\gamma_i} - 1, (H_q)^{\gamma_i}, (H_q)^{\gamma_i} + 1\}$. It follows that $\forall q \in N_p, |\,(H_p)^{\gamma_{i+1}} - (H_q)^{\gamma_{i+1}}\,| < 2$ for each processor p which executes a rule (since $\forall q \in N_p, (H_q)^{\gamma_i} = (H_q)^{\gamma_{i+1}}$). Overall, $\gamma_{i+1} \in \Gamma_1$. □

Secondly, we prove the liveness of our algorithm.

Lemma 6 *$\forall \gamma_0 \in \Gamma_1, \forall p \in V$, p executes the rule $\boldsymbol{(N)}$ in a finite time in any execution starting from γ_0.*

Proof. Let $\gamma \in \Gamma_1$. Following Lemma 5, the only enabled rule is $\boldsymbol{(N)}$. We prove this property by induction. To this end, we define the following property (where p denotes a processor):
$\boldsymbol{(P_d)}$: If d is the distance between p and the nearest end of the chain, then p executes the rule $\boldsymbol{(N)}$ 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 $\boldsymbol{(N)}$ since $(Inter(N_p))^{\gamma'} \supseteq \{(H_q)^{\gamma'}, (H_q)^{\gamma'} + 1\}$ 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 $\boldsymbol{(P_{d-1})}$ is true. Denote q the neighbor of p which is on the half-chain starting with p which realize d. Assume by absurd that p is never enabled for rule $\boldsymbol{(N)}$ in an execution ϵ starting from $\gamma_0 \in \Gamma_1$. This implies that, for each configuration γ' which is contained in ϵ, we have $|\,(Inter(N_p))^{\gamma'}\,| = 1$ (since if $|\,(Inter(N_p))^{\gamma'}\,| = 0$, then $\gamma' \notin \Gamma_1$). Let us study the following cases:

Case 1: $\bar{q}$ never executes a rule in ϵ.
It follows that: $\forall \gamma' \in \epsilon, (H_q)^{\gamma'} = (H_{\bar{q}})^{\gamma'} + 2$ or $(H_q)^{\gamma'} = (H_{\bar{q}})^{\gamma'} - 2$. By construction of $(Inter(N_q))^{\gamma'}$ and of rule $\boldsymbol{(N)}$, the clock of q can not move from a value to the other in a step (recall that only rule $\boldsymbol{(N)}$ can be enabled for q since $\gamma' \in \Gamma_1$ by lemma 5), this implies that q never executes the rule $\boldsymbol{(N)}$, which contradicts $\boldsymbol{(P_{d-1})}$.

Case 2: $\bar{q}$ executes a rule in a finite time in ϵ.
Let $\gamma \rightarrow \gamma'$ be the first step in which $\bar{q}$ executes the rule $\boldsymbol{(N)}$. It is known that, for any $\gamma \in \Gamma_1$:

$$|\,(Inter(N_p))^{\gamma}\,| = 1 \Rightarrow \begin{cases} (H_{\bar{q}})^{\gamma} = ((H_p)^{\gamma} - 1) \wedge (H_q)^{\gamma} = ((H_p)^{\gamma} + 1)\ \boldsymbol{(A)} \\ or \\ (H_{\bar{q}})^{\gamma} = ((H_p)^{\gamma} + 1) \wedge (H_q)^{\gamma} = ((H_p)^{\gamma} - 1)\ \boldsymbol{(B)} \end{cases}$$

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 absurd.

Since the two cases are absurd, 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 $\boldsymbol{(P_d)}$ 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 7 *If $\gamma \in \Gamma_1$, then any processor increments its clock in a finite time in any execution starting from γ.*

Proof. Assume by 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 be $\alpha = (H_p)^{\gamma_0}$. By Lemma 6, p executes infinitely often ***(N)***. But, it never increments, that implies that $next((Inter(N_p))^{\gamma}, (H_p)^{\gamma}) = min\{(Inter(N_p)^{\gamma})\}$ at each execution of a rule by p (in a configuration γ). Since $\forall \gamma \in \Gamma_1, \forall q \in N_p, \mid (H_p)^{\gamma} - (H_q)^{\gamma} \mid < 2$ and $\forall q \in N_p, (Inter(N_p))^{\gamma} \subseteq \{(H_q)^{\gamma} - 1, (H_q)^{\gamma}, (H_q)^{\gamma} + 1\}$, we have: $min\{(Inter(N_p))^{\gamma}\} \leq (H_p)^{\gamma}$.

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

If $\bar{q}$ does not exist or if $(H_{\bar{q}})^{\gamma} \in \{(H_p)^{\gamma}, (H_p)^{\gamma} + 1\}$, then $(H_p)^{\gamma} + 1 \in (Inter(N_p))^{\gamma}$. This contradicts $next((Inter(N_p))^{\gamma}, (H_p)^{\gamma}) = min\{(Inter(N_p)^{\gamma})\}$. We deduce that $\bar{q}$ exists and that $(H_{\bar{q}})^{\gamma} = (H_p)^{\gamma} - 1$. This implies that ***(N)*** is not enabled for p.

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

In the following we prove the convergence of our algorithm.

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

$$\forall e = \{p, q\} \in E, \omega(e, \gamma) = \mid (H_p)^{\gamma} - (H_q)^{\gamma} \mid$$
$$\forall p \in V, \varpi(p, \gamma) = \max_{e \in E / p \in e} \{\omega(e, \gamma)\}$$
$$\forall i \in \mathbb{N}, p(i, \gamma) = |\{e \in E / \omega(e, \gamma) = i\}|$$

Consider the following potential function:

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

We compare two values of P by lexicographic order. The following properties are verified:

$$\forall \gamma \in \Gamma, P(\gamma) \geq (\ldots 0, 0)$$
$$\forall \gamma \in \Gamma, \gamma \in \Gamma_1 \Leftrightarrow P(\gamma) = (\ldots, 0, 0)$$
$$\forall \gamma \in \Gamma, \gamma \in \Gamma \setminus \Gamma_1 \Leftrightarrow P(\gamma) > (\ldots, 0, 0)$$

Lemma 8 *If $\gamma \in \Gamma \setminus \Gamma_1$, then every step $\gamma \rightarrow \gamma'$ which contains the execution of a rule by a processor p such that $\varpi(p) \geq 2$ verifies $P(\gamma') < P(\gamma)$.*

Proof. Let $\gamma \in \Gamma \setminus \Gamma_1$. Let $\gamma \rightarrow \gamma'$ be a step which 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 $j = \omega(\{p,q\},\gamma) = |\,(H_p)^\gamma - (H_q)^\gamma\,|$. $(Inter(N_p))^\gamma = \{(H_q)^\gamma - 1, (H_q)^\gamma, (H_q)^\gamma + 1\}$. It follows that p executed rule $\boldsymbol{(N)}$. So, we have $|\,(H_p)^{\gamma'} - (H_q)^{\gamma'}\,| \leq 1$. 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 $\gamma \rightarrow \gamma'$.

By construction of $(Inter(N_p))^\gamma$, 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 $\gamma \rightarrow \gamma'$.

This case is similar to the case 2.1.

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

Let us study the following cases:

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

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

$$(H_p)^\gamma \geq \frac{(H_q)^\gamma + (H_{\bar{q}})^\gamma}{2}$$

1) Assume that $(H_p)^\gamma > (H_{\bar{q}})^\gamma + \frac{(H_q)^\gamma + (H_{\bar{q}})^\gamma}{2}$.

We can say that:

$$\begin{aligned} \omega(e,\gamma) &> (H_{\bar{q}})^\gamma - (H_q)^\gamma + \tfrac{(H_q)^\gamma + (H_{\bar{q}})^\gamma}{2} \\ \omega(e,\gamma') &= \left\lfloor \tfrac{(H_q)^\gamma + (H_{\bar{q}})^\gamma}{2} \right\rfloor \end{aligned}$$

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

On the other hand,

$$\begin{aligned} \omega(\bar{e},\gamma) &> \tfrac{(H_q)^\gamma + (H_{\bar{q}})^\gamma}{2} \\ \omega(\bar{e},\gamma') &= (H_{\bar{q}})^\gamma - \left\lfloor \tfrac{(H_q)^\gamma + (H_{\bar{q}})^\gamma}{2} \right\rfloor \end{aligned}$$

Then: $\omega(\bar{e},\gamma') \leq \omega(\bar{e},\gamma)$.
In conclusion, we have: $P(\gamma') < P(\gamma)$.
2) Assume that $(H_p)^\gamma \leq (H_{\bar{q}})^\gamma + \frac{(H_q)^\gamma+(H_{\bar{q}})^\gamma}{2}$.
We have then:

$$\begin{array}{r} \omega(e,\gamma) > \frac{(H_q)^\gamma+(H_{\bar{q}})^\gamma}{2} \\ \omega(e,\gamma') = \left\lfloor \frac{(H_q)^\gamma+(H_{\bar{q}})^\gamma}{2} \right\rfloor \end{array}$$

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 $(H_p)^\gamma > (H_{\bar{q}})^\gamma$, due to the fact that $(H_p)^\gamma > \left\lceil \frac{(H_q)^\gamma+(H_{\bar{q}})^\gamma}{2} \right\rceil$ in the contrary case).
In conclusion, we have: $P(\gamma') < P(\gamma)$.

Case 2.3.2: We have $(H_q)^\gamma > (H_{\bar{q}})^\gamma$.
This case is similar to the case 2.3.1 when we permute q and $\bar{q}$.

That proves the result. □

Lemma 9 *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 reason by absurd. 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 verifying $\varpi(p,\gamma_0) \geq 2$.

In a first time, assume that one of the end p of the chain verify: $\varpi(p,\gamma_0) \geq 2$. 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 $\forall i \in \mathbb{N}, (Inter(N_p))^{\gamma_i} = \{(H_q)^{\gamma_0} - 1, (H_q)^{\gamma_0}, (H_q)^{\gamma_0}+1\}$, p is so always enabled for rule $\boldsymbol{(N)}$. Since the daemon is strongly fair, p executes a rule in a finite time, that is contradictory. We can deduce that the two ends of the chain verifies: $\varpi(p,\gamma_0) < 2$.

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 verifying $\varpi(p,\gamma_0) \geq 2$ never execute a rule, that implies that the network verify:

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

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, that implies that its clock value never changes). All these constraints allows us to say:

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

By a reasoning similar to these of the proof of Lemma 7, we can prove that all processors between p_0 and p_{i-1} executes infinitely often the rule $\boldsymbol{(N)}$ in every execution starting from γ_k

even if p_i never execute a rule (this is the case by hypothesis). By a reasoning similar to these of the proof of Lemma 7, we can state that $H_{p_{i-1}}$ not remains constant. The construction of $Inter(N_{p_{i-1}})$ implies that $\left(Inter(N_{p_{i-1}})\right)^{\gamma_j} \subseteq \{(H_{p_i})^{\gamma_k} - 1, (H_{p_i})^{\gamma_k}, (H_{p_i})^{\gamma_k} + 1\}$ for each $j \geq k$ (since H_{p_i} does not change by hypothesis).

If we are in the case $\boldsymbol{(A)}$, we can deduce that $H_{p_{i-1}}$ takes infinitely often the value $(H_{p_i})^{\gamma_k} - 1$ or $(H_{p_i})^{\gamma_k}$. We can see that p_i is enabled by $\boldsymbol{(N)}$ and $\boldsymbol{(C_1)}$ respectively. This contradicts the construction of k (recall that p_i is never enabled in ϵ from γ_k).

If we are in the case $\boldsymbol{(B)}$, we can deduce that $H_{p_{i-1}}$ takes infinitely often the value $(H_{p_i})^{\gamma_k} + 1$ or $(H_{p_i})^{\gamma_k}$. We can see that p_i is enabled by $\boldsymbol{(N)}$ and $\boldsymbol{(C_1)}$ respectively. This contradicts the construction of k (recall that p_i is never enabled in ϵ from γ_k).

This finishes the proof. □

Lemma 10 *There exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$.*

Proof. The result follows directly from Lemmas 8 and 9. □

Finally, we can conclude:

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

Proof. Lemmas 5, 7, and 10 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 11 *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 5 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 12 *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 7 taking in account a processor $p \in V^*$.

In order to prove the property of Lemma 6, we take d as the distance between p and the end e of the chain which verifies: no processor between p and e is crashed. This implies that the processor q is not crashed. The case in which $\bar{q}$ is crashed appear in the case 1 of the induction.

We can repeat the reasoning of the proof of Lemma 7 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 13 *There exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$.*

Proof. We repeat the reasoning of Lemma 10 taking in account a processor $p \in V^*$.

We can repeat the reasoning of the proof of the property of Lemma 8 since the fact that a processor is crashed or not does not modify the proof.

In order to prove the property of Lemma 9, we take a numbering of processors which ensure 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, that implies that there exists at least two processors p such that $\varpi(p, \gamma_k) \geq 2$, that allows us to choose one which is not crashed. The case in which p_{i+1} is crashed does not modify the proof since we assumed that this processor never execute 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 11, 12, and 13 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 $\mathcal{UFTSS}$ in which there is no crash.

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

Lemma 14 *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 5 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 15 *$\forall \gamma_0 \in \Gamma_1, \forall p \in V$, p executes rule $(\boldsymbol{N})$ in a finite time in every execution starting from γ_0.*

Proof. Let be $\gamma_0 \in \Gamma_1$ (we have seen in the proof of Lemma 5 that implies that only rule $(\boldsymbol{N})$ 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, that 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 verify: $\exists q \in N_p, (H_p)^{\gamma_j} = (H_q)^{\gamma_j} + 1$ and $(H_p)^{\gamma_j} = (H_{\bar{q}})^{\gamma_j} - 1$. Let i be the smallest integer greater than k such that the step $\gamma_i \rightarrow \gamma_{i+1}$ 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 $\gamma_i \rightarrow \gamma_{i+1}$.

Since p is not enabled in γ_{i+1} (by hypothesis) and that the execution of rule $(\boldsymbol{N})$ always modifies the clock values (*cf.* proof of Lemma 7),we have:

$$\begin{cases} (H_p)^{\gamma_i} = (H_q)^{\gamma_i} + 1 \text{ and } (H_p)^{\gamma_i} = (H_{\bar{q}})^{\gamma_i} - 1 \\ and \\ (H_p)^{\gamma_{i+1}} = (H_q)^{\gamma_{i+1}} - 1 \text{ and } (H_p)^{\gamma_{i+1}} = (H_{\bar{q}})^{\gamma_{i+1}} + 1 \end{cases}$$

The clock move of $\bar{q}$ contradicts the construction of rule $\boldsymbol{(N)}$ and $(Inter(N_p))^{\gamma_i}$. 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 $\boldsymbol{(N)}$, $(Inter(N_q))^{\gamma_i}$, and the fact that the execution of this rule must change the clock value, we have: $(H_q)^{\gamma_{i+1}} \in \{(H_p)^{\gamma_i}, (H_p)^{\gamma_i} - 1\}$. Processor p is then enabled for rule $\boldsymbol{(N)}$ (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 contradiction, 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, that 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 $(H_p)^{\gamma_k} + 2$ and if the second neighbor of $\bar{q}$ have a clock value equals to $(H_p)^{\gamma_k} - 2$, etc..

Since the ring has a finite length n, we obtain (following the same reasoning) there exists two neighboring processors p_1, p_2 such that $(H_{p_1})^{\gamma_k} = (H_p)^{\gamma_k} + \alpha$ and $(H_{p_2})^{\gamma_k} = (H_p)^{\gamma_k} - \beta$ (with α and β integers greater or equal to 1 depending on the parity of n). Therefore, $|\,(H_{p_1})^{\gamma_k} - (H_{p_2})^{\gamma_k}\,| = \alpha + \beta \geq 2$. Then, we obtain that $\gamma_k \notin \Gamma_1$, which contradicts Lemma 14 and proves the lemma. □

Lemma 16 *If $\gamma_0 \in \Gamma_1$, then every processor increments its clock in a finite time in ϵ.*

Proof. The proof is similar to these of Lemma 7 using Lemma 15 (instead of Lemma 6) 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 10.

Lemma 17 *If $\gamma \in \Gamma \setminus \Gamma_1$, then every step $\gamma \to \gamma'$ which contains the execution of a rule of a processor p such that $\varpi(p) \geq 2$ verifies $P(\gamma') < P(\gamma)$.*

Proof. The proof is similar to the proof of Lemma 8 since the topology of the network has no impact on the proof (note that the case 1 is impossible on a ring). □

Lemma 18 *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, by contradiction, that there exists an execution $\epsilon = \gamma_0\gamma_1 \ldots$ starting from γ_0 which contains no execution of a rule by any processor p which verifies $\varpi(p, \gamma_0) \geq 2$. 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 verifying $\omega(\{p,q\},\gamma_k)=\varpi(p,\gamma_k)$. By hypothesis, q never executes a rule. Therefore, its clock value remains constant. Let us study the following cases:

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

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

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

$$\forall j\geq k,\begin{cases}(Inter(N_p))^{\gamma_j}=\emptyset\\ and\\ (H_p)^{\gamma_j}\in\left\{\left\lceil\frac{(H_q)^{\gamma_j}+(H_{\bar{q}})^{\gamma_j}}{2}\right\rceil,\left\lfloor\frac{(H_q)^{\gamma_j}+(H_{\bar{q}})^{\gamma_j}}{2}\right\rfloor\right\}\end{cases}$$

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, that contradicts the hypothesis). It follows: $(H_q)^{\gamma_j}<(H_p)^{\gamma_j}<(H_{\bar{q}})^{\gamma_j}$ or $(H_q)^{\gamma_j}>(H_p)^{\gamma_j}>(H_{\bar{q}})^{\gamma_j}$.

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

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

Lemma 19 *There exists $i\geq 0$ such that $\gamma_i\in\Gamma_1$.*

Proof. The result follows directly from Lemmas 17 and 18. □

Finally, we can conclude:

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

Proof. Lemmas 14, 16, and 19 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.

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

Lemma 20 *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 14 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 21 *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 16. Note that the crash of a processor is possible only for the case 4. □

In the following we prove the convergence of our algorithm.

Lemma 22 *There exists $i \geq 0$ such that $\gamma_i \in \Gamma_1$.*

Proof. This proof is similar to the proof of Lemma 19 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 20, 21, and 22 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, 8, 9, and 10. □

5 Conclusion

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.

There 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 [16] (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 could induce *probabilistic* self-stabilizing ones, this raises the open question of the possibility of *probabilistic* FTSS for dynamic tasks in asynchronous systems.

References

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

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

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

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

[5] Christian Boulinier, Franck Petit, and Vincent Villain. Synchronous vs. asynchronous unison. In Ted Herman and Sébastien Tixeuil, editors, *Self-Stabilizing Systems*, volume 3764 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2005.

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

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

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

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

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

[11] Shlomi Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Real-Time Systems*, 12(1):95–107, 1997.

[12] Shlomi Dolev and Jennifer L. Welch. Wait-free clock synchronization. *Algorithmica*, 18(4):486–511, 1997.

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

[14] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[15] Ajei S. Gopal and Kenneth J. Perry. Unifying self-stabilization and fault-tolerance (preliminary version). In *PODC*, pages 195–206, 1993.

[16] Mohamed G. Gouda. The theory of weak stabilization. In Ajoy Kumar Datta and Ted Herman, editors, *WSS*, volume 2194 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2001.

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

[18] Ezra N. Hoch, Danny Dolev, and Ariel Daliot. Self-stabilizing byzantine digital clock synchronization. In Ajoy Kumar Datta and Maria Gradinariu, editors, *SSS*, volume 4280 of *Lecture Notes in Computer Science*, pages 350–362. Springer, 2006.

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

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

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

[22] Marina Papatriantafilou and Philippas Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, 1997.