Silent Anonymous Snap-Stabilizing Termination Detection

Lélia Blin
LIP6
Université d'Évry Val d'Essone
Paris, France
lelia.blin@lip6.fr

Colette Johnen
LIP6, INRIA
Université de Bordeaux
Bordeaux, France
colette.johnen@labri.fr

Gabriel Le Bouder
LIP6, INRIA
Sorbonne Université
Paris, France
gabriel.le-bouder@lip6.fr

Franck Petit
LIP6, INRIA
Sorbonne Université
Paris, France
franck.petit@lip6.fr

***Abstract*—We address the problem of Termination Detection (TD) in asynchronous networks. It is known that TD cannot be achieved in the context of self-stabilization, except in the specific case where the TD algorithm is snap-stabilizing, *i.e.*, it always behaves according to its specification regardless of the initial configuration. In this paper, we propose a generic, deterministic, snap-stabilizing, silent algorithm that detects whether an observed terminating silent self-stabilizing algorithm, $\mathcal{A}$, has converged to a configuration that satisfies an intended predicate. Our algorithm assumes that nodes know (an upper bound on) the network diameter D. However, it requires no underlying structure, nor specific topology (arbitrary network), and works in anonymous networks, *i.e.*, our algorithm uses no kind of assumption allowing to distinguish one or more nodes. Furthermore, it works under the weakest scheduling assumptions *a.k.a*, the unfair daemon. Built over any asynchronous self-stabilizing underlying unison $\mathcal{U}$, our solution adds only $O(\log D)$ bits per node. Since there exists no unison algorithm with better space complexity, the extra space of our solution is negligible *w.r.t.* the space complexity of the underlying unison algorithm. Our algorithm provides a positive answer in $O(\max(k, k', D))$ time units, where k and k' are the stabilization time complexities of $\mathcal{A}$ and $\mathcal{U}$, respectively.**

***Index Terms*—Deterministic Termination Detection, Stabilization, Anonymous Networks**

I. Introduction

Termination Detection [1] is a fundamental and widely studied problem of distributed systems. It belongs to the category of global system observation mechanisms that processes of the distributed system may need in the accomplishment of a global computation, for example, detecting the presence of a deadlock, taking a snapshot of the global system state, or maintaining a logical distributed clock. The distributed nature of systems makes these problems difficult to solve. They are subject to specific distributed algorithms dedicated to control the global state of other distributed algorithms. Regarding the termination detection (TD, for short) problem, any node of the distributed system may need to detect whether a computation has globally terminated. More precisely, upon a (local) request (for instance, from an application) a node initiates an instance of TD over the whole system to find out whether another distributed algorithm is terminated.

Supported by the French ANR project ESTATE (ANR-16 CE25-0009-03).

In this paper, we focus on distributed algorithms that are *self-stabilizing* [2]–[4]. These algorithms guarantee that starting from an arbitrary initial system configuration, provided the faults cannot corrupt the code of the algorithm, the system recovers its specification within a finite time, without any external intervention, to a configuration from which its specification is forever satisfied thereafter. Self-stabilization is well known to tolerate *transient* faults, *i.e.*, faults that occur at an unpredictable but quite rare times, and that do not result in permanent hardware damages. By contrast with other so-called robust approaches, self-stabilization does not hide the effect of faults. It means that self-stabilization is suitable for systems that tolerate temporary deviance of their specification. After transient faults cease, there is a finite period of time, called the *stabilization phase*, during which the safety properties of the system may be violated. The stabilization time (that is to say the time it takes for the system to recover its specification after the faults cease) is one of the main efficiency parameters of self-stabilizing algorithms.

Since the effect of faults cannot be hidden, it is impossible for nodes to locally decide whether the system has globally converged (except for a few trivial problems), and therefore, is terminated. Indeed, assume an application where nodes decide to use local variables (supposed to be updated by a global detection mechanism) whether the stabilization phase is over or not. Then it is always possible to build an arbitrary configuration in which the same nodes have access to exactly the same local information. As a consequence, the two situations are undistinguishable, leading to a wrong decision in the second one. Nevertheless, a self-stabilizing algorithm (at least) guarantees that by repeating global detection instances, the variables provide the correct answer at some, but in an unpredictable time. Indeed, let ssTD be a self-stabilizing TD algorithm. By initiating ssTD, it is possible that ssTD returns a wrong answer during the stabilization phase, *e.g.*, ssTD returns "*yes*" (meaning that an observed algorithm A terminated), while A actually did not terminate. In other words, ssTD can compute incorrect (or unsafe) answers several (but a finite number of) times before at the end computing true/correct answers. Self-stabilization ensures that by repeating instances of ssTD, the answer is eventually correct forever.

In [5], it is shown that the termination detection can be achieved using only one detection instance *i.e.*, at the very first request to know whether an observed algorithm is terminated or not, the returned answer is correct, even if the request was initiated during the stabilization phase. Such a self-stabilizing algorithm is said to be *snap-stabilizing* [6]. Snap-stabilization is a stronger form of self-stabilization, as after transient faults cease, a snap-stabilizing system *immediately* resumes its correct behavior, without any external intervention (still, provided the faults have not corrupted its code). By definition, snap-stabilizing algorithms are self-stabilizing algorithms whose stabilization time is null. It is important to notice that snap-stabilizing algorithms do not hide better the effects of transient faults than the self-stabilizing algorithms that do not respect this property. However, while a self-stabilizing algorithm guarantees only a finite, yet generally unbounded, number of incorrect answers after the faults cease, a snap-stabilizing algorithm offers correct answers from the first request (after the faults cease).

Note that the snap-stabilizing solutions in [5] assume *named* distributed systems, *i.e.*, systems where each node has a unique ID. Even if most of existing distributed systems are named, developing algorithms that do not use process IDs definitively makes sense in several aspects. Such algorithms are said to be *anonymous* algorithms—or, algorithms for *anonymous* systems.

Anonymity often makes the resolution of some problems harder by far. That makes anonymous approaches interesting from a computational point of view. Leader Election is the most iconic problem. Indeed, the problem is quite trivial to solve deterministically by using the total order provided by process IDs—just choose the maximum or minimum ID as the leader. By contrast, Leader Election cannot be solved in systems lacking of properties (as process IDs) that break possible symmetries [7]. Also, anonymous solutions *a priori* require less memory. Notably, they do not need process IDs, thus saving (at least) $O(\log n)$ bits, the number of bits necessary to store an ID in a system of n nodes. Anonymous approaches are also very attractive from a practical point of view. Indeed, they provide solutions that preserve user privacy, they work for systems with homonyms, where nodes can change their names or be replaced during the algorithm life time. They are also very suitable for networks made of units with weak capabilities such as wireless sensor networks, body-area networks, *etc*.

A. Related Work

The question of detecting the stability of a self-stabilizing algorithm in an anonymous system was first addressed in [8]. In this paper, the authors introduce the notion of *observer*: a local node that can detect correctness of a given algorithm, but cannot influence it. Assuming the central daemon, where only one node takes a step at each time and a prime-size uniform ring, the authors propose a deterministic distributed algorithm for the observer that detects stability in $\Theta(n^2)$ steps from the time the ring is stabilized. Located at each node, the proposed observer is not subject to any type of corruption *i.e.*, it is not self-stabilizing. In [9], the authors also propose a non-self-stabilizing observer. They propose an observer for synchronous rooted systems, where all enabled nodes take steps simultaneously and a unique node is distinguished from the others. In an synchronous and non-self-stabilizing system, the same authors remove the constraint of having a distinguished node by introducing randomization [10].

The first deterministic algorithm that solves the problem addressed in [8] that is also self-stabilizing is proposed in [5]. By contrast with the above results [8]–[10] that are not self-stabilizing, the results in [5] show the necessity to achieve snap-stabilization and not only self-stabilization. Indeed, only snap-stabilization offers the desirable property of returning the right answer to the request of knowing whether a self-stabilizing algorithm achieved stability, even during the stabilization phase of the observed algorithm. As mentioned earlier, the solution in [5] requires a named network.

In anonymous networks of arbitrary size, *unison* [11]–[14] offers a nice support to implement deterministic solutions [15]. The asynchronous unison consists of maintaining a local *logical clock* (sometimes referred to as *counter*), one for each node, such that: (i) the clock value of each node does not differ by more than 1 with any of its neighbors, and (ii) the clock value of each node is increased by 1 infinitely often. The unison principle is a strong tool to synchronize the whole system by implementing a synchronization barrier. To the best of our knowledge, this principle forms the basis of all known deterministic solutions for anonymous networks, even non-self-stabilizing, that solve global (*a.k.a.*, *Total* [16]) problems, *i.e.* problems involving all nodes of the network before a decision can be taken. TD obviously belongs to this class of algorithms. The phase algorithm in [16] and the TD algorithm in [17] are typical examples of algorithms that use an underlying unison mechanism. Both algorithms require that nodes know (an upper bound on) the network diameter D, *i.e.* the maximum distance between two nodes of the network. As far as we know, the question of the necessity of this knowledge to be able to deterministically solve total problems remains open. In [18], the author proposes a snap-stabilizing TD algorithm to characterize tasks that are solvable with snap-stabilizing algorithms in anonymous networks. This algorithm combines the synchronization technique in [17] and the self-stabilizing enumeration algorithm in [19]. The former actually uses a unison mechanism and requires that nodes know (an upper bound of) D. The latter works on a particular class of graphs (so called, non-ambiguous graphs [20]) and implements a renaming mechanism that uses an *a priori* exponential memory size.

Many self-stabilizing algorithms aim at being *silent*, *i.e.*, after some calculations, the communication registers used by the algorithm remain fixed as long as no request is made [21], [22]. By communication variable, we mean variables shared between neighboring nodes. Silence is trivially a desired property with algorithms that terminate by building distributed fixed structures—*e.g.*, spanning trees, coloring, Maximal Independent Set, *etc*. It is also very desirable for so-called "long-

lived" algorithms—*e.g.*, mutual exclusion, unison, routing, *etc*.—to reduce communication operations and bandwidth. For instance, it is quite easy to design a self-stabilizing silent unison [12], [22] by modifying the above Condition (ii) as follows: (ii') the clock value of each node is increased by 1, provided that at least one node decides to do it.

B. Contribution

In this paper, we address asynchronous anonymous networks. We focus on *terminating and silent* self-stabilizing algorithms *i.e.*, self-stabilizing algorithms that converge in finite time to a desired global configuration from which no values of the communication variables are changed thereafter. We present a generic, deterministic, silent algorithm that detects whether an observed terminating silent self-stabilizing algorithm has converged to a configuration that satisfies an intended predicate. Our solution uses similar techniques as in [15] to achieve snap-stabilization, namely it is based on an underlying unison algorithm. However, in this paper, the latter can be any (asynchronous) unison in the literature, *e.g.*, [11]–[14].

As all existing deterministic anonymous total algorithms in the literature (*e.g.*, [15]–[18]), our algorithm requires that nodes know (an upper bound on) the diameter D of the network. It works under the weakest scheduling assumptions *a.k.a*, the unfair daemon. Built over any asynchronous self-stabilizing underlying unison $\mathcal{U}$, our solution adds only $O(\log D)$ bits per node, where D is the diameter of the network. Since there exists no unison algorithm with better space complexity — the best space complexity for the asynchronous unison is obtained in [14] with $O(\log D)$ bits —, the extra space of our solution is negligible *w.r.t.* the space complexity of the underlying unison algorithm. Time complexities are given in terms of *rounds* that captures the execution rate of the slowest node in any computation [3], [5]. The response time computes the number of rounds between the time when a request is triggered and the time when the answer to that request is returned[1]. The response time of our solution is in $O(\max(k, k', D))$ rounds, where k and k' are the stabilization time complexities of $\mathcal{A}$ and $\mathcal{U}$, respectively. In other words, once both $\mathcal{A}$ and $\mathcal{U}$ are stabilized, our solution provides an answer in optimal time, *i.e.*, $O(D)$ rounds.

Paper Outline: The remainder of the paper is organized as follows. We first formally describe notations, definitions, and the execution model in Section II. Then, in Section III, we formally define the unison problem, and establish some properties of self-stabilizing unison algorithm. In Section IV, we present and formally describe our snap-stabilizing algorithm for termination detection. Section V contains the proofs of our claims and theorems, and in specific we establish that our algorithm is snap-stabilizing for the termination detection problem. We make some concluding remarks in Section VI.

[1]An answer to a request is given when the observed terminating silent algorithm actually completed its task,*i.e.*, when the system reached a configuration satisfying an intended global predicate.

II. Model

We consider a distributed system modeled by a non-oriented connected graph $G = (V, E)$. Two nodes are neighbors in G if and only if the nodes can communicate with each other. The set of neighbors of v in G is denoted by N_v. The set of closed neighbors of v is the union of v and its neighbors, and is denoted $N[v]$. The *size* of a graph is the number of nodes it has, and is noted $n = |V|$. The longest shortest path between two nodes of G diameter D of a graph G is the size of the longest shortest path between two nodes of G.

The (local) *algorithm* (or *program*) of a node consists of a finite set of *variables* denoted by S and a finite set $R_{\mathcal{A}}$ of guarded actions (also referred to *rules*). Some variables, like N_v and the unique identity of the node, may be *constant* inputs provided by the system. An algorithm that does not use the identity of the network nodes is called *anonymous*. Each node can read its own variables and variables owned by the neighboring nodes, but can write only to its own (non-constant) variables. The communication between nodes lies in the classical *state model* [2], *i.e.*, are carried out by the variables. Each rule in $R_{\mathcal{A}}$ is of the following form:*label: guard* $\rightarrow$ *action*. *Labels* are used to identify rules in $R_{\mathcal{A}}$. A *guard* is a Boolean predicate over the variables of the node and that of its neighbors. If the guard of a rule is evaluated to true on node v, then the rule is said enabled at v. An action is the updating of some variables of a node. It can be executed only if the corresponding guard is evaluated to true.

The *state* of a node v designates the content of all the local variables of the node v. The set of states of all nodes in the system, is called *configuration* of the system represented by the symbol γ. We denote by Γ the set of all possible configurations of the system. A node is said to be *enabled* in a configuration γ if and only if at least one of its rule is enabled in γ.

When one or several enabled nodes are simultaneously activated, they all atomically read the states of their neighbors, non-deterministically pick one enabled rule, and execute the corresponding action. If no node is enabled in a configuration γ, then γ is a *terminal* configuration.

The asynchrony of the system is modeled by a map $\mathcal{D}$, commonly called *schedule*. The input of the schedule $\mathcal{D}$ is a non-empty sequence of configurations $(\gamma_0, \gamma_1, \ldots, \gamma_k)$. If γ_k is terminal, then $\mathcal{D}(\gamma_0, \gamma_1, \ldots, \gamma_k)$ is the empty set, else it is a non-empty subset of the enabled nodes of γ_k. A scheduler is a class of schedules. We assume, in this paper, the most general scheduler, where no assumption is made, commonly called the *unfair distributed scheduler*.

Let us denote by $\gamma \underset{\mathcal{A}}{\longrightarrow} \gamma'$ a computing step of Algorithm $\mathcal{A}$, where γ' is obtained from γ after the activation of one or several enabled nodes and the simultaneous execution of their action. Given a schedule $\mathcal{D}$, an execution is a finite or infinite sequence of computing steps $\gamma_0 \underset{\mathcal{A}}{\longrightarrow} \gamma_1 \underset{\mathcal{A}}{\longrightarrow} \cdots$ such that $\forall i$, the nodes that were activated between γ_i and γ_{i+1} is $\mathcal{D}(\gamma_0, \gamma_1, \ldots, \gamma_i)$. Configuration γ_0 is called the initial configuration of the system. An execution ϵ is said *maximal* if ϵ is infinite, or if $\epsilon = (\gamma_0 \underset{\mathcal{A}}{\longrightarrow} \gamma_1 \underset{\mathcal{A}}{\longrightarrow} \cdots \underset{\mathcal{A}}{\longrightarrow} \gamma_f)$ and γ_f

is a terminal configuration. If $\epsilon = (\gamma_0 \underset{\mathcal{A}}{\longrightarrow} \gamma_1 \underset{\mathcal{A}}{\longrightarrow} \cdots)$ is an execution, then we call *subexecution* of ϵ any subsequence $\epsilon' = (\gamma_i \underset{\mathcal{A}}{\longrightarrow} \gamma_{i+1} \underset{\mathcal{A}}{\longrightarrow} \cdots)$.

A node v is *neutralized* in the computing step $\gamma \underset{\mathcal{A}}{\longrightarrow} \gamma'$ if v is enabled in γ and not enabled in γ', but does not execute any action between these two configurations. Neutralization occurs when some neighbors of v changed their state between γ and γ', and this change makes the guards of all actions of v false. To evaluate the stabilization time, we use the classical notion of *round* [3], [5]. Let ϵ be an execution. The first round of an execution ϵ, noted ϵ', is the minimal prefix of ϵ in which every node that is enabled in the initial configuration either executes an action or becomes neutralized. Let ϵ'' be the suffix of ϵ starting from the last configuration of ϵ'. The second round of ϵ is the first round of ϵ'', and so forth.

The predicates considered in the following are boolean functions over the configurations of the system. Let R be a predicate and γ a configuration. We note $\gamma \in R$ when R evaluates to true in γ, and $\gamma \notin R$ otherwise. Denote by true the predicate that always evaluates to true. Given an algorithm, $\mathcal{A}$, the predicate R is *closed* for $\mathcal{A}$ if for every computing step $\gamma \underset{\mathcal{A}}{\longrightarrow} \gamma'$ such that $\gamma \in R$, then $\gamma' \in R$. Algorithm $\mathcal{A}$ converges to predicate Q from predicate R under the scheduler $\mathcal{D}$ if Q is closed and if for any execution $\epsilon = (\gamma_0 \underset{\mathcal{A}}{\longrightarrow} \gamma_1 \underset{\mathcal{A}}{\longrightarrow} \cdots)$ under $\mathcal{D}$ such that $\gamma_0 \in R$, there exists $i \geqslant 0$ such that $\gamma_i \in Q$. This is noted $R \rhd_{\mathcal{AD}} Q$. If $\mathcal{D}$ is the unfair distributed scheduler, we may simplify $R \rhd_{\mathcal{AD}} Q$ to $R \rhd_{\mathcal{A}} Q$. R is an *attractor* if $\mathsf{true} \rhd R$. When it is clear in the context, we indifferently use a predicate and the set of configurations it describes.

The specification SP_P of a problem P is a predicate over the executions, which describes a specific behavior of the system. An algorithm $\mathcal{A}$ solves a problem P under a certain scheduler if every execution of $\mathcal{A}$ under that scheduler satisfies the specification of P.

Definition 1 (Simulation)**.** *An algorithm $\mathcal{T}$ is said to simulate an algorithm $\mathcal{A}$ if the variables of $\mathcal{A}$ are a subset of the variable of $\mathcal{T}$, and if any (possibly infinite) execution of $\mathcal{T}$, $\epsilon = (\gamma_0 \underset{\mathcal{T}}{\longrightarrow} \gamma_1 \underset{\mathcal{T}}{\longrightarrow} \cdots)$ corresponds to a legitimate execution of $\mathcal{A}$, $\epsilon_{\mathcal{A}} = (\gamma_{0|\mathcal{A}} \underset{\mathcal{T}}{\longrightarrow} \gamma_{1|\mathcal{A}} \underset{\mathcal{T}}{\longrightarrow} \cdots)$ on the subset of the variables of $\mathcal{A}$, with possibly empty computing steps $\gamma_{i|\mathcal{A}} = \gamma_{i+1|\mathcal{A}}$.*

Definition 2 (Self-stabilization)**.** *Algorithm $\mathcal{A}$ is a self-stabilizing algorithm for problem P under a certain scheduler $\mathcal{D}$ if there exists a predicate R such that $\Gamma \rhd_{\mathcal{AD}} R$ and such that any execution of $\mathcal{A}$ starting from a configuration $\gamma \in R$ satisfies SP_P.*

An execution of $\mathcal{A}$ has stabilized *once R is valid. The stabilization time of $\mathcal{A}$ is the maximal number of rounds in executions of $\mathcal{A}$ starting from any configuration, before $\mathcal{A}$ stabilized.*

A self-stabilizing algorithm for problem P is silent if maximal executions that satisfy SP_P are finite.

Definition 3 (Snap-stabilization)**.** *An algorithm $\mathcal{A}$ is snap-stabilizing for a specification SP if any execution of $\mathcal{A}$ starting from any configuration satisfies SP.*

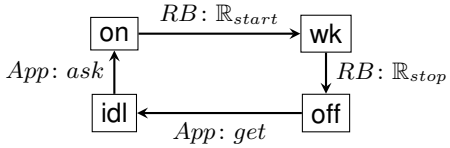

Fig. 1. Diagram for req_v

A request-based algorithm RB is an algorithm that interacts with an external application App, typically an external user or another algorithm. This interaction takes place through one shared variable, req, that can be updated by the application. The variable req has four values: $\mathsf{idl}, \mathsf{on}, \mathsf{wk}, \mathsf{off}$. The value idl means that no request is in progress on node v for the application. Node v is then said *idle*. Value on means that a request is initiated, but the computation is not launched. Once the requested computation is running, req is set to wk. The fourth value, off indicates that the requested computation is done, but the result has not been communicated to the application yet. Variable req is updated through four methods. Two of them, ask and get, are part of App. The two others, $\mathbb{R}_{start}$ and $\mathbb{R}_{stop}$, are part of RB—see Figure 1.

Snap-stabilization is a suitable paradigm for request-based algorithms. Indeed, a snap-stabilizing request-based algorithm ensures that if the application executes ask on one node v, then the following execution of get on the same node v will mark the end of a correct computation. We define the response-time of a request-based algorithm as the maximal number of rounds between any computing step in which the application executes ask on any node v, and the following computing step in which the same node v updates req_v to off.

In this paper, we consider the Snap-Stabilizing Termination Detection problem.

Definition 4 (Termination Detection)**.** *Let $\mathcal{A}$ be a silent self-stabilizing algorithm that solves a problem P, and let $\mathcal{T}$ be a request-based algorithm. $\mathcal{T}$ is a snap-stabilizing algorithm that solves the Termination Detection of $\mathcal{A}$ if:*

1) *$\mathcal{T}$ simulates $\mathcal{A}$,*
2) *$\mathcal{A}$ ultimately converges in any execution of $\mathcal{T}$,*
3) *if a request is emitted on node u (req is set from idl to on), then u ultimately answers (i.e., req is set to off),*
4) *when it answers, algorithm $\mathcal{A}$ has converged.*

The last two items can be more formally defined as follows: let $\epsilon = \gamma_0 \underset{\mathcal{T}}{\longrightarrow} \gamma_1 \underset{\mathcal{T}}{\longrightarrow} \cdots$ be an execution of $\mathcal{T}$. If $\exists t \in \mathbb{N}, \exists v \in V$ such that in $\gamma_t, \mathsf{req}_v = \mathsf{on}$, then (i) $\exists t' > t : \mathsf{req}_v = \mathsf{off}$ and (ii) $\forall t' > t : \mathsf{req}_v = \mathsf{off}$, $\mathcal{A}$ has stabilized in $\gamma_{t'}$.

III. Unison algorithms

A. Definition of the problem

Our algorithm lies in unison algorithms. Such algorithms guarantee that all the nodes in the system have a variable

clock that increases infinitely often such that two neighbors have a difference of at most one between their clocks.

Definition 5 (Unison Algorithm)**.** *Let $\mathcal{U}$ be a distributed algorithm. $\mathcal{U}$ is a self-stabilizing unison algorithm with range m if the following conditions are respected.*

1) *Definition: Every node u has a variable clock_u, with values in $\mathcal{C}$, such that $\mathcal{C} \supseteq \mathbb{Z}/m\mathbb{Z}$;*
2) *Safety: Let $\mathcal{P}(u) \equiv \mathsf{clock}_u \in \mathbb{Z}/m\mathbb{Z} \wedge \big(\forall v \in N_u, \mathsf{clock}_v \in \{\mathsf{clock}_u - 1, \mathsf{clock}_u, \mathsf{clock}_u + 1\}\big)$, and let $P_{\mathcal{U}} \equiv \forall u \in V, \mathcal{P}(u)$. Then $\mathsf{true} \rhd_{\mathcal{U}} P_{\mathcal{U}}$;*
3) *Liveness: $\forall u, \mathsf{clock}_u$ is increased infinitely many times.*

The unison algorithm has converged as soon as $P_{\mathcal{U}}$ is true.

To achive maximal genericity, we consider that $\mathcal{C}$ can contain other values than the classical clock values, in $\mathbb{Z}/m\mathbb{Z}$. Typically, $\mathcal{C}$ can contain control values such as $\perp$, nil, etc. Notice that if the value of $\mathsf{clock} \in \mathbb{Z}/m\mathbb{Z}$, then the usual arithmetic operations on clock are modulo$-m$ operations. In the following, we are interested in the operations on clock, specific to each algorithm, only if $\mathsf{clock} \in \mathbb{Z}/m\mathbb{Z}$.

Any self-stabilizing unison algorithm can be made *silent*, with as consequence the loss of the liveness property. To do so, we can prevent the clock to increase if there is no request for it and the safety property is achieved. Essentially, if $\mathcal{P}(u)$ and $\forall v \in N_u, \mathsf{clock}_v \in \{\mathsf{clock}_u - 1, \mathsf{clock}_u\}$, then u is not activatable. If the initial configuration of the system respects the safety property, then the system reaches a terminal configuration as soon as all the clock reach the highest initial value. Yet, one may want to keep a silent self-stabilizing unison algorithm running as long as some external condition is not satisfied. This *request* notion may be extended with the concept of *local request* [12]. Silent self-stabilizing unison algorithms are equipped with one additional predicate $\mathsf{LocReq}(v)$ that depends on external parameters. When this predicate is evaluated to true on one node, then it takes precedence over the termination condition, and forces the system to keep running with respect to $\mathcal{P}(u)$. In the following, we consider a silent self-stabilizing unison algorithm.

It will be useful for us to separate the rules of $\mathcal{U}$ into three distinct sets. Let $R_{\mathcal{U}}$ be the set of the rules of $\mathcal{U}$. Let us define $\mathcal{P}^+(u) \equiv \mathcal{P}(u) \wedge \forall v \in N_u, \mathsf{clock}_v \neq \mathsf{clock}_u - 1$, and $\mathcal{P}^-(u) \equiv \mathcal{P}(u) \wedge \exists v \in N_u : \mathsf{clock}_v = \mathsf{clock}_u - 1$. Note that $P^+(u)$ and $P^-(u)$ are mutually exclusive, and, joined, equate $P(u)$. We suppose without loss of generality that the guard of all the rules in $R_{\mathcal{U}}$ contains $\neg\mathcal{P}(u)$, $\mathcal{P}^+(u)$, or $\mathcal{P}^-(u)$. Since $\mathcal{U}$ is a self-stabilizing algorithm, any rule that includes $\mathcal{P}(u)$ in its guard guarantees that its action will not invalidate $\mathcal{P}(u)$. Thus, actions that include $\mathcal{P}^+(u)$ either increment clock_u by one or do not update it. In the same way, actions that include $\mathcal{P}^-(u)$ cannot update clock_u. Among all the rules that include $\mathcal{P}^+(u)$, we denote by $R_{\mathcal{U}_N}$, and call the *normal rules* the rules whose action increases by 1 the variable clock_u. These are the rules by which $\mathcal{U}$ makes progress. We denote by $R_{\mathcal{U}_T}$ and call the *transparent rules* all the other rules that include $\mathcal{P}(u)$, that do not update clock_u by definition. Transparent rules do not necessarily exist, but cannot be avoided *a priori*. Finally, we denote by $R_{\mathcal{U}_C}$ and call the *convergence rules* all the other rules of $R_{\mathcal{U}}$, rules that include $\neg\mathcal{P}(u)$ and that enable convergence. By definition, $\mathcal{U}$ has stabilized if and only if no rule of $R_{\mathcal{U}_C}$ is activatable. We define the sluggishness of $\mathcal{U}$, and denote $\mathcal{S}(\mathcal{U})$ as the maximal number of consecutive transparent rules a node can execute between two executions of normal rules, after stabilization. Sluggishness depicts how much the transparent rules might slow down clock increment. Sluggishness of algorithms presented in [11]–[14], [23] is 0.

B. Tools on unison

In this subsection we introduce logical tools that will be useful to reason on generic unison algorithms. Definitions 6 to 9 were introduced in [15]. Definition 10 is original work and was designed specifically for our proof.

Definition 6 (Event)**.** *Let $\epsilon = (\gamma_0 \underset{\mathcal{U}}{\rightarrow} \gamma_1 \cdots)$ be a finite or infinite execution. An event is a pair $(v, t+1)$ such that v is activated in $\gamma_t \underset{\mathcal{U}}{\rightarrow} \gamma_{t+1}$. We say that v executes a rule at time $t+1$. By convention, $(v, 0)$ is an event for all $v \in V$. An event (v, t) is said to be external if the guard of the executed rule by v depends on at least one shared register of a neighbor of v. An event (v, t) is a normal (resp. transparent, resp. convergence) event if v executes a normal (resp. transparent, resp. convergence) rule at time t.*

Definition 7 (Causal relation $\rightsquigarrow$)**.** *The causal relation is the smallest relation $\rightsquigarrow$ on the set of events such that the following two conditions hold:*

1) *Let (v, t) and (v, t') be two events such that t' is the greatest integer such that $t' < t$. Then $(v, t') \rightsquigarrow (v, t)$;*
2) *Let (v, t) and (w, t') be two events such that (v, t) is an external event, $w \in N_v$, t' is the greatest integer such that $t' < t$. Then $(w, t') \rightsquigarrow (v, t)$.*

Definition 8 (Dependance Relation $\rightsquigarrow^N$)**.** *Let (v, t_0) and (w, t') be two events. We say that (w, t') normally depends on (v, t_0), and denote $(v, t_0) \rightsquigarrow^N (w, t')$ if there exists $k \geqslant 0$ and $t_1, \ldots t_k$ such that $(v, t_0) \rightsquigarrow (v, t_1) \rightsquigarrow \cdots (v, t_k) \rightsquigarrow (w, t')$ and $(v, t_i), i > 0$ are transparent events, and (w, t') is a normal event. Denote $(v, t) \preceq^N (w, t')$ if there is a path* w.r.t *to $\rightsquigarrow^N$ from (v, t) to (w, t').*

Definition 9 (N-sequence)**.** *When a node v consecutively executes k normal rules, possibly intercut with transparent rules,*

$$\rightsquigarrow^N (v, t_0) \rightsquigarrow^N (v, t_1) \rightsquigarrow^N \cdots \rightsquigarrow^N (v, t_{k-1})$$

it executes an N-sequence of length k.

Definition 10 (Causal pyramid)**.** *Let $p = v_0 v_1 \ldots v_k$ be a path of length k. We say that p is a causal pyramid of length $d \geqslant 2k+1$ and of origin t_0^0 if $\forall i \in [0, k], \forall j \in [i, (d-1)-i], \exists t_j^i$, such that:*

- *$\forall i$, we have $t_i^i < t_{i+1}^{i+1}$ and $t_{(d-1)-(i+1)}^{i+1} < t_{(d-1)-i}^i$*
- *$\forall i, v_i$ does not execute rules of $R_{\mathcal{U}_C}$ in $(t_{i-1}^{i-1}, t_{(d-1)-(i-1)}^{i-1})$*

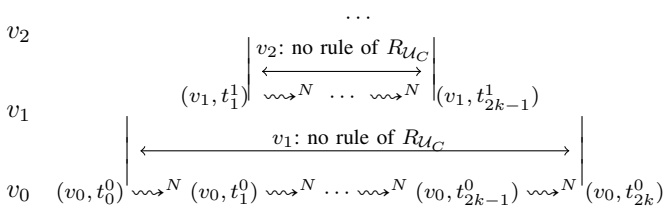

Fig. 2. Causal Pyramid Scheme

- $\forall i, v_i$ *executes an* N*-sequence* $\rightsquigarrow^N (v_i, t_i^i) \rightsquigarrow^N (v_i, t_{i+1}^i) \rightsquigarrow^N \cdots \rightsquigarrow^N (v_i, t_{(d-1)-i}^i)$

Notice that if p is a causal pyramid, then $(v_0, t_0^0) \preceq^N (v_1, t_1^1) \preceq^N \cdots \preceq^N (v_k, t_k^k) \preceq^N (v_k, t_{(d-1)-k}^k) \preceq^N \cdots \preceq^N (v_0, t_{d-1}^0)$.

C. Properties of unison algorithms

In this section, we extend and adapt some of the results of [15] to any unison algorithm.

Lemma 1. *Let v and w be two neighbors. Suppose that v is a causal pyramid of length 3 and of origin t_0, and that in (t_0, t_2), w does not execute rules of $R_{\mathcal{U}_C}$. Then vw is a causal pyramid of length 3 and of origin t_0.*

Proof. At time t_0, clock_w is equal to $p-1$ or p. If (w, t') does not exist, then at time t_2, $\mathsf{clock}_w \leqslant p$ and $\mathsf{clock}_v = p + 2$, which is contradictory. □

The following lemma directly follows by induction of Lemma 1 on the length of an N-sequence.

Lemma 2. *Let v and w be two neighbors. Suppose that v is a causal pyramid of length $x \geqslant 3$ and of origin t_0, and that in (t_0, t_{x-1}), w does not execute rules of $R_{\mathcal{U}_C}$. Then vw is a causal pyramid of length x and of origin t_0.*

Lemma 2 establishes a link between the behavior of two neighbors. By induction on the distance between v and any other process, Theorem 1 follows.

Theorem 1. *Let v_0 and v_k be two nodes, and let $p = v_0 v_1 \cdots v_k$ be a path. Suppose that $v_0 \dots v_{k-1}$ is a causal pyramid of length $d \geqslant 2k+1$, and that in $(t_{k-1}^{k-1}, t_{(d-1)-(k-1)}^{k-1})$, v_k does not execute rules of $R_{\mathcal{U}_C}$. Then $v_0 \cdots v_k$ is a causal pyramid of length $2k+1$.*

Corollary 1. *Let v_0 and v_k be two nodes, and let $p = v_0 v_1 \cdots v_k$ be a path. Suppose that v_0 is a causal pyramid of length $2k+1$, and that $\forall i, v_i$ does not execute rules of $R_{\mathcal{U}_C}$ in (t_0, t_{2k}). Then $v_0 \cdots v_k$ is a causal pyramid of length $2k+1$.*

IV. Algorithm

Let us consider a silent self-stabilizing algorithm $\mathcal{A}$ that solves a problem P under the unfair distributed scheduler. We introduce a generic mechanism that builds an anonymous silent snap-stabilizing request-based algorithm $\mathcal{T}$ that solves the Termination Detection of $\mathcal{A}$. Based on an anonymous self-stabilizing unison algorithm $\mathcal{U}$, it follows the request-based mechanism described in Section II. More specifically, for each node v, the algorithm communicates with the application App by means of req_v—refer to Figure 1. To know whether $\mathcal{A}$ has terminated or not, App triggers a request to $\mathcal{T}$ by executing $App\colon ask$ on some idle nodes $v_1, v_2, \dots, v_k$ of the system, setting $\mathsf{req}_{v_{i,i\in[1,k]}}$ from idl to on. Next, $\mathcal{T}$ answers to the request only after algorithm $\mathcal{A}$ has terminated, by setting req_v to off. Then, App may execute $App\colon get$ that sets req_v to idl.

A. Variables

The variables of node v in algorithm $\mathcal{T}$ are:

- $\mathsf{S}(\mathcal{A}, v)$ the set of all variables of node v in algorithm $\mathcal{A}$.
- $\mathsf{S}(\mathcal{U}, v)$ the set of all variables of node v in algorithm $\mathcal{U}$, including clock_v.
- $\mathsf{doi}_v \in [0, 2D+2]$, for duration of inactivity. This variable is used to store the number of steps since the last time a convergence rule (for $\mathcal{A}$ or $\mathcal{U}$) was executed by node v. Variable doi propagates through the whole system with the following rule: if the maximum value of doi among the neighbors of v is p, then v cannot set doi_v under $p-1$.
- $\mathsf{cnt}_v \in [0, 2D+1]$ is a countdown to 0, initiated at $2D+1$ when the application asks for the termination.
- $\mathsf{req}_v \in \{\mathsf{on}, \mathsf{wk}, \mathsf{off}, \mathsf{idl}\}$, for request, is the interface between App and $\mathcal{T}$. req_v may be updated according to Figure 1.

The space complexity of the variables of $\mathcal{T}$ is $O(\log D)$ bits per node, where D is an upper bound of the diameter of the graph. Consequently, the space complexity of the whole system is $O(\mathcal{C}(\mathcal{A}) + \mathcal{C}(U) + \log D)$ bits per node, where $\mathcal{C}(\mathcal{A})$ (resp. $\mathcal{C}(\mathcal{U})$) is the space complexity of algorithm $\mathcal{A}$ (resp. of algorithm $\mathcal{U}$) in bits per node.

B. Overview of the algorithm

The detail of the rules of algorithm $\mathcal{T}$ is presented in Algorithm 1. Algorithm $\mathcal{T}$ simulates both algorithms $\mathcal{A}$ and $\mathcal{U}$, independently. When an enabled node v is activated, v atomically executes the rule of $\mathcal{A}$ for which it is enabled, if such rule exists, the rule of $\mathcal{U}$ for which it is enabled, if such rule exists, and updates the proper variables of algorithm $\mathcal{T}$.

Since algorithm $\mathcal{T}$ performs two distinct and independent tasks: the simulation of $\mathcal{A}$ and $\mathcal{U}$ on the one hand, the detection of termination on the other hand, it is natural to divide the set of the rules of $\mathcal{T}$, $R_{\mathcal{T}}$, in two disjoint sets: $\mathcal{R}^S_{\text{simul}}$ and $\mathcal{R}^S_{\text{proper}}$. A node is enabled for $\mathcal{T}$ if it enabled for at least one rule of $\mathcal{R}^S_{\text{simul}}$ or $\mathcal{R}^S_{\text{proper}}$. If an enabled node is activated, then it atomically executes the rule of $\mathcal{R}^S_{\text{simul}}$ for which it is activatable, if such a rule exists, and the rule of $\mathcal{R}^S_{\text{proper}}$ for which it is activatable, if such a rule exists.

$\mathcal{R}^S_{\text{simul}}$ contains the rules that simulate algorithms $\mathcal{A}$ and $\mathcal{U}$, and also updates the variable doi. If the activated node v has not yet converged for both $\mathcal{A}$ and $\mathcal{U}$, then it executes a rule for at least one of those algorithms and sets its variable doi_v to $2D + 2$. The convergence rules are

$\mathbb{R}_{cvg} = \{\mathbb{R}^{\mathcal{U}_C}_{cvg}, \mathbb{R}^{N}_{cvg}, \mathbb{R}^{\mathcal{A}}_{cvg}\} \subset \mathcal{R}^{S}_{\text{simul}}$, whose different guards correspond to the different possible ways for a node to be activatable for $\mathcal{A}$ and/or $\mathcal{U}$. Be aware that the rules of $\mathbb{R}_{cvg}$ are not convergence rules in the sense of unison algorithms, since $\mathbb{R}^{N}_{cvg}$ is a normal rule, and $\mathbb{R}^{\mathcal{A}}_{cvg}$ is a transparent rule.

Otherwise, if w is only activatable for a transparent rule of $\mathcal{U}$, then it executes $\mathbb{R}_{trans} \in \mathcal{R}^{S}_{\text{simul}}$ and nothing else.

Finally, if v is only activatable for a normal rule of $\mathcal{U}$ then v executes the rule $\mathbb{R}_{wait} \in \mathcal{R}^{S}_{\text{simul}}$, that sets doi_v to one less than the maximum value of doi of the closed neighbors of v. As a consequence, as long as one node v has not converged for $\mathcal{A}$, variable doi is maintained at $2D+2$ on v. When activated, the neighbors of v set their variable doi to $2D+1$ (or $2D+2$ if one rule of $\mathbb{R}_{cvg}$ is activated). After that, if one neighbor of a neighbor of v is activated, then it sets its variable doi to at least $2D$, and so on. Thus, if one node is activatable for a convergence rule, then the variable doi will propagate at high value along the graph. Yet, this property requires that all the nodes are activated, in a specific order.

Fortunately, since $\mathcal{U}$ is a unison algorithm, no node, and no subset of nodes, may compute independently of the rest of the system, and decrease its doi variable down to 0 regardless of what happens in the whole system. This guarantees that, starting from any configuration, after a node v executes $2D+1$ computing steps, the variable doi_v is under the influence of all the other nodes of the system. Consequently, after $2D+1$ computing steps of node v, the variable doi_v cannot be 0 unless all the nodes in the graph have converged for both $\mathcal{A}$ and $\mathcal{U}$. This property allows us to design our procedure thanks to the variable cnt.

Whenever App asks one node v for the termination of the algorithm, v executes rule $\mathbb{R}_{start}$, it sets its variable cnt_v to $2D+1$ and updates its variable req_v from on to wk. Variable cnt_v is decreased by one each time a normal rule of $\mathcal{U}$ is executed, and is updated to $2D+1$ as soon as $\mathsf{doi}_v \neq 0$. Since $\mathcal{U}$ is a unison algorithm, in a legitimate execution cnt_v reaches 0 only after all nodes have converged for $\mathcal{A}$. As a desired consequence, algorithm $\mathcal{T}$ is snap-stabilizing for the detection of termination for algorithm $\mathcal{A}$.

C. Predicates

Let $RS \in R_{\mathcal{A}}, R_{\mathcal{U}}, R_{\mathcal{U}_C}, R_{\mathcal{U}_T}, R_{\mathcal{U}_N}$ be the set of the rules of one algorithm, and let v be a node. $\mathsf{Act}(RS, v)$ returns true if and only if node u is activatable by a rule of RS.

$$\mathsf{Act}(RS, v) \equiv v \text{ is activatable by one rule of } RS \quad (1)$$

As described in Section III-A, we consider a silent self-stabilizing unison algorithm $\mathcal{U}$. In $\mathcal{T}$, the normal rules of $\mathcal{U}$, denoted $R_{\mathcal{U}_N}$, are not activatable on node v unless the following predicate $\mathsf{LocReq}(v)$ is evaluated to true.

$$\mathsf{LocReq}(v) \equiv \bigvee \begin{array}{l}(\exists u \in N_v : \mathsf{clock}_u = \mathsf{clock}_v + 1)\\ \mathsf{Act}(R_{\mathcal{A}}, v)\\ (\mathsf{doi}_v \neq 0)\\ (\mathsf{req}_v \in \{\mathsf{on}, \mathsf{wk}\})\end{array} \quad (2)$$

When both algorithm $\mathcal{A}$ and $\mathcal{U}$ have converged on node v, the only simulation rules that v may execute are the normal rules of algorithm $\mathcal{U}$, which permit the liveness of $\mathcal{U}$. This situation is described by the predicate $\mathsf{UnisonOnly}$.

$$\mathsf{UnisonOnly}(v) \equiv \neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \mathsf{Act}(R_{\mathcal{U}_N}, v) \quad (3)$$

D. Actions

Let $RS \in R_{\mathcal{A}}, R_{\mathcal{U}}, R_{\mathcal{U}_C}, R_{\mathcal{U}_T}, R_{\mathcal{U}_N}$ be the set of the rules of one algorithm, and let v be a node. Let $\mathsf{SimulR}(RS, v)$ be a procedure that executes the activatable rule in RS on node v if such rule exists and which does nothing otherwise. Procedure $\mathsf{Simul}(v)$ sequentially executes one rule of Algorithm $\mathcal{A}$ if possible, then one rule of Algorithm $\mathcal{U}$, again if possible. Formally:

$$\mathsf{SimulR}(RS, v) \equiv \begin{cases} v \text{ updates its state executing} & \\ \text{the enabled rules in } RS & \text{if } \mathsf{Act}(RS, v)\\ v \text{ does nothing} & \text{otherwise}\end{cases} \quad (4)$$

$$\mathsf{Simul}(v) \equiv \mathsf{SimulR}(R_{\mathcal{A}}, v); \mathsf{SimulR}(R_{\mathcal{U}}, v) \quad (5)$$

Procedure $\mathsf{Propagate\_doi}(v)$ updates the variable doi_v to one less than the maximal value of doi of the closed neighbors of v:

$$\mathsf{Propagate\_doi}(v) \equiv \mathsf{doi}_v := \max(0, \max_{w \in N[v]}(\mathsf{doi}_w - 1)) \quad (6)$$

V. Correctness of Algorithm $\mathcal{T}$

In this section, we establish that $\mathcal{T}$ is a snap-stabilizing procedure for the detection of the termination of Algorithm $\mathcal{A}$. This proof is divided in three subsections. In Subsection V-A, we prove that $\mathcal{T}$ satisfies both conditions 1 and 2 of Definition 4. In Subsection V-B, we prove that $\mathcal{T}$ satisfies Condition 3 of Definition 4. Finally, in Subsection V-C, we prove that $\mathcal{T}$ satisfies Condition 4 of Definition 4.

A. Simulation properties of $\mathcal{T}$

In this subsection, we establish Theorem 2, which ensures that $\mathcal{T}$ satisfies Condition 1 and Condition 2 of Definition 4. Due to the lack of place, the detail of the proof is omitted. We will describe the scheme of the proof only.

Theorem 2. *$\mathcal{T}$ is a simulation of both $\mathcal{A}$ and $\mathcal{U}$, and in any execution of $\mathcal{T}$, both $\mathcal{A}$ and $\mathcal{U}$ ultimately converge.*

We first establish by syntaxical analysis Lemma 3 that characterizes the terminal configurations of $\mathcal{T}$. These configurations are the one in which $\mathcal{U}$ converged, and in which on all the nodes v of the system, $\mathsf{LocReq}(v)$ is not verified (this includes the termination of $\mathcal{A}$).

Lemma 3. *The terminal configurations of $\mathcal{T}$ are the configurations such that $\forall v \in V$:*

$$\neg\mathsf{Act}(R_{\mathcal{U}}, v) \wedge \neg\mathsf{LocReq}(v)$$

According to the design of Algorithm $\mathcal{T}$, it is pretty obvious that any execution of $\mathcal{T}$ is a simulation of both $\mathcal{A}$ and $\mathcal{U}$. Since $\mathcal{A}$ is a silent self-stabilizing algorithm, there does not exists infinite executions of $\mathcal{A}$. Thus, there only exists a finite number

Algorithm 1: Algorithm $\mathcal{T}$

During a step, if a rule of set $\mathcal{R}^S_{\text{simul}}$ and a rule of set $\mathcal{R}^S_{\text{proper}}$ are enabled then v executes these 2 rules.

$\mathcal{R}^S_{\textbf{simul}}$ **:: rules to update** doi

$\mathbb{R}^{\mathcal{U}_C}_{cvg}$	:	$\mathsf{Act}(R_{\mathcal{U}_C}, v)$	$\longrightarrow \mathsf{Simul}(v); \mathsf{doi}_v := 2D+2$	$\in R_{\mathcal{T}_C}$
$\mathbb{R}^{\mathcal{A}}_{cvg}$	:	$\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \neg(\mathsf{Act}(R_{\mathcal{U}_C}, v) \vee \mathsf{Act}(R_{\mathcal{U}_N}, v))$	$\longrightarrow \mathsf{Simul}(v); \mathsf{doi}_v := 2D+2$	$\in R_{\mathcal{T}_T}$
$\mathbb{R}_{trans}$	:	$\neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \mathsf{Act}(R_{\mathcal{U}_T}, v)$	$\longrightarrow \mathsf{Simul}(v)$	$\in R_{\mathcal{T}_T}$
$\mathbb{R}^{N}_{cvg}$	:	$\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \mathsf{Act}(R_{\mathcal{U}_N}, v)$	$\longrightarrow \mathsf{Simul}(v); \mathsf{doi}_v := 2D+2$	$\in R_{\mathcal{T}_N}$
$\mathbb{R}_{wait}$	:	$\mathsf{UnisonOnly}(v)$	$\longrightarrow \mathsf{Simul}(v); \mathsf{Propagate\_doi}(v)$	$\in R_{\mathcal{T}_N}$

$\mathcal{R}^S_{\textbf{proper}}$ **:: rules to update** cnt **and** req

$\mathbb{R}_{start}$	:	$\mathsf{req}_v = \mathsf{on}$	$\longrightarrow \mathsf{req}_v := \mathsf{wk}; \mathsf{cnt}_v := 2D+1$
$\mathbb{R}_{stop}$	:	$\mathsf{req}_v = \mathsf{wk} \wedge \mathsf{UnisonOnly}(v) \wedge \mathsf{cnt}_v = 0 \wedge \mathsf{doi}_v = 0$	$\longrightarrow \mathsf{req}_v = \mathsf{off}$
$\mathbb{R}_{cpt}$	:	$\mathsf{req}_v = \mathsf{wk} \wedge \mathsf{UnisonOnly}(v) \wedge \mathsf{cnt}_v \neq 0 \wedge \mathsf{doi}_v = 0$	$\longrightarrow \mathsf{cnt}_v := \mathsf{cnt}_v - 1$
$\mathbb{R}_{end}$	:	$\mathsf{req}_v = \mathsf{wk} \wedge \mathsf{UnisonOnly}(v) \wedge \mathsf{doi}_v \neq 0$	$\longrightarrow \mathsf{cnt}_v := 2D+1$

of activation of $\mathcal{A}$ in an execution of $\mathcal{T}$, and since $\mathcal{U}$ is a self-stabilizing algorithm, it ultimately converges in any execution of $\mathcal{T}$. This is stated in Lemma 4.

Lemma 4. *$\mathcal{U}$ converges in any maximal execution of $\mathcal{T}$.*

Finally, since $\mathcal{U}$ is a unison algorithm, all the nodes are regularly activated in any exececution, which implies that the convergence of $\mathcal{A}$ necessarily occurs in any maximal execution of $\mathcal{T}$. This completes the proof of Theorem 2.

Remark 1. *Since $\mathcal{T}$ is a simulation of $\mathcal{U}$, we can extend the results of Section III-C to the executions of $\mathcal{T}$. To do this, we extend to $\mathcal{T}$ the concepts of normal, transparent, and convergence rule, normal dependance relation, and N-sequence. $\mathcal{T}$ has one convergence rule $\mathbb{R}^{\mathcal{U}_C}_{cvg}$, two transparent rules $\mathbb{R}^{\mathcal{A}}_{cvg}$ and $\mathbb{R}_{trans}$, and two normal rules $\mathbb{R}^{N}_{cvg}$ and $\mathbb{R}_{wait}$. Specifically, Theorem 1 and Corollary 1 remain valid on $\mathcal{T}$.*

B. Termination of $\mathcal{T}$

In this subsection, we establish Theorem 3, that states that $\mathcal{T}$ satisfies Condition 3 of Definition 4.

We define $\Gamma_{\mathsf{cvg}} \subset \Gamma$ the set of configurations in which $\mathcal{A}$ has stabilized and $\mathcal{U}$ has converged, and Γ_{doi} the set of configuration in which, in addition, $\forall v, \mathsf{doi}_v = 0$. Lemma 5 states that Γ_{doi} is an attractor.

Definition 11. *Let $\Gamma_{\mathsf{doi}} \subset \Gamma_{\mathsf{cvg}} \subset \Gamma$ be*

Γ_{cvg} : $\forall v \in V, (\neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \neg\mathsf{Act}(R_{\mathcal{U}_C}, v))$

Γ_{doi} : $\forall v \in V, (\neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \neg\mathsf{Act}(R_{\mathcal{U}_C}, v) \wedge \mathsf{doi}_v = 0)$

Lemma 5. $\Gamma \rhd_{\mathcal{T}} \Gamma_{\mathsf{cvg}} \rhd_{\mathcal{T}} \Gamma_{\mathsf{doi}}$

Due to the lack of place, the proof of Lemma 5 is omitted.

Finally, we define Γ^w_{td} where $w \in V$, as the set of the configurations of Γ_{doi} such that $\mathsf{req}_w \in \{\mathsf{off}, \mathsf{idl}\}$. Theorem 3 states that any execution that starts in Γ_{doi} eventually reaches a configuration of Γ^w_{td}.

Definition 12. *Let $w \in V$. Let $\Gamma^w_{td} \subset \Gamma_{\mathsf{doi}}$ be the set of configurations such that, $\forall v \in V$:*

$$(\neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \neg\mathsf{Act}(R_{\mathcal{U}_C}, v) \wedge \mathsf{doi}_v = 0) \wedge \mathsf{req}_w \in \{\mathsf{off}, \mathsf{idl}\}$$

Theorem 3. *Let $w \in V$ be a node and $\epsilon = (\gamma_0 \underset{\mathcal{T}}{\longrightarrow} \gamma_1 \underset{\mathcal{T}}{\longrightarrow} \cdots)$ be a maximal execution such that $\gamma_0 \in \Gamma_{\mathsf{doi}}$. There exists $i \geqslant 0 : \gamma_i \in \Gamma^w_{td}$.*

Proof. Suppose $\gamma_0 \notin \Gamma^w_{td}$, *i.e.* $\gamma_0(\mathsf{req}_w) \in \{\mathsf{on}, \mathsf{wk}\}$.

Case 1: in $\gamma_0, \mathsf{req}_w = \mathsf{wk}$. Since Γ_{doi} is closed, the only rules of $\mathcal{R}^S_{\text{proper}}$ activatables by w are $\mathbb{R}_{stop}$ and $\mathbb{R}_{cpt}$. As long as $\mathsf{req}_w \in \{\mathsf{on}, \mathsf{wk}\}$, $\mathsf{LocReq}(w)$ is satisfied, and since $\mathcal{U}$ is a unison algorithm, it makes progress, which means that $\mathbb{R}_{wait}$ is regularly activated on all nodes. Thus, as long as $\mathsf{cnt}_w \neq 0$, $\mathbb{R}_{cpt}$ is regularly activated by w. Since each activation of $\mathbb{R}_{cpt}$ by w decreases cnt_w by 1, it ultimately reaches 0, afterwhat $\mathbb{R}_{stop}$ is activated by w, and then, the system has reached Γ^w_{td}.

Case 2: in $\gamma_0, \mathsf{req}_w = \mathsf{on}$. Then the previous guarantees as well that node w will eventually be activated, through rule $\mathbb{R}_{start}$, after what $\mathsf{req}_w = \mathsf{wk}$, which is the case above. □

The previous results ensure that, whatever the initial configuration is, the system globally converges to Γ_{doi}, and each node is infinitely often in an availability state for the app.

C. Snap-stabilization

In this subsection, we establish Theorem 5, that states that $\mathcal{T}$ satisfies Condition 4) of Definition 4.

The activation of $\mathbb{R}_{start}$ by w corresponds to the request by w to detect the termination of Algorithm $\mathcal{A}$. The activation of $\mathbb{R}_{stop}$ by w corresponds to the detection of the termination of Algorithm $\mathcal{A}$ by w. Lemma 6 states that w will eventually execute the rule $\mathbb{R}_{stop}$ along any execution starting by a termination detection request by w (*i.e.* the activation of $\mathbb{R}_{start}$ by w). This lemma allows us to define, for a maximal execution starting by a termination detection request by w, the response time to w's request: $\mathsf{f}(w, \epsilon)$, the time of the first computing step in which w executes the rule $\mathbb{R}_{stop}$. This is stated in Definition 13.

Lemma 6. *Let $w \in V$ and let $\epsilon = (\gamma \underset{\mathcal{T}}{\longrightarrow} \gamma_0 \underset{\mathcal{T}}{\longrightarrow} \gamma_1 \underset{\mathcal{T}}{\longrightarrow} \cdots)$ be a maximal execution such that w executes $\mathbb{R}_{start}$ in $\gamma \underset{\mathcal{T}}{\longrightarrow} \gamma_0$. $\exists i > 0$ such that in $\gamma_{i-1} \underset{\mathcal{T}}{\longrightarrow} \gamma_i$, w executes $\mathbb{R}_{stop}$.*

Proof. Lemma 5 ensure that there exists i such that $\gamma_i \in \Gamma_{\mathsf{doi}}$. Theorem 3 applied to $\epsilon' = (\gamma_i \underset{\mathcal{T}}{\longrightarrow} \gamma_{i+1} \underset{\mathcal{T}}{\longrightarrow} \cdots)$ ensures that there exists $j \geqslant i : \gamma_j \in \Gamma^w_{td}$.

In $\gamma_0, \mathsf{req}_w = \mathsf{wk}$, and in $\gamma_j, \mathsf{req}_w \in \{\mathsf{off}, \mathsf{idl}\}$. This is only possible if between γ_0 and γ_j, w executes $\mathbb{R}_{stop}$. □

Definition 13 (Final Descent)**.** *Let $w \in V$ and let $\epsilon = (\gamma \underset{\tau}{\longrightarrow} \gamma_0 \underset{\tau}{\longrightarrow} \gamma_1 \underset{\tau}{\longrightarrow} \cdots)$ be a maximal execution such that w executes $\mathbb{R}_{start}$ in $\gamma \underset{\tau}{\longrightarrow} \gamma_0$. We denote by $\mathsf{f}(w,\epsilon)$ the* time of response *of w in ϵ: the smallest $i > 0$ such that during $\gamma_{i-1} \underset{\tau}{\longrightarrow} \gamma_i$, w executes $\mathbb{R}_{stop}$. If w executes $\mathbb{R}_{end}$ along $\epsilon_f = (\gamma \underset{\tau}{\longrightarrow} \gamma_0 \underset{\tau}{\longrightarrow} \cdots \underset{\tau}{\longrightarrow} \gamma_{\mathsf{f}(w,\epsilon)-1})$ then let $\mathsf{cs}_s = \gamma_{s-1} \underset{\tau}{\longrightarrow} \gamma_s$ be the latest computing step of ϵ_f in which w executes $\mathbb{R}_{end}$. Otherwise we set $s = 0$ and $\mathsf{cs}_s = \gamma \underset{\tau}{\longrightarrow} \gamma_0$. Remark that cs_s is the latest computing step of ϵ_f in which w executes a rule that sets cnt_w at $2D + 1$.*

We denote by ϵ_s the final descent *of w in ϵ, $\epsilon_s = \gamma_s \underset{\tau}{\longrightarrow} \gamma_{s+1} \underset{\tau}{\longrightarrow} \cdots \underset{\tau}{\longrightarrow} \gamma_{\mathsf{f}(w,\epsilon)-1}$.*

In the sequel of this subsection, we study the properties of ϵ_s. We establish that $\gamma_s \in \Gamma_{\mathsf{cvg}}$. As Γ_{cvg} is closed, $\gamma_{\mathsf{f}(w,\epsilon)} \in \Gamma_{\mathsf{cvg}}$: a terminal configuration of $\mathcal{A}$ is reached.

Lemma 7 will allow us to use Corrolary 1.

Lemma 7. *Let w be a node and let $\epsilon = (\gamma \underset{\tau}{\longrightarrow} \gamma_0 \underset{\tau}{\longrightarrow} \gamma_1 \underset{\tau}{\longrightarrow} \cdots)$ be a maximal execution such that w executes $\mathbb{R}_{start}$ in $\gamma \underset{\tau}{\longrightarrow} \gamma_0$. Then w does not execute any rule of $\mathbb{R}_{cvg}$ in ϵ_s, and w executes an N-sequence of length at least $2D + 1$ in ϵ_s.*

The first assertion is proved by contradiction, as a consequence of Definition 13. The second one comes from the fact that $\mathbb{R}_{cpt}$ is executed $2D + 1$ times in ϵ_s, thus so does $\mathbb{R}_{wait}$.

Definition 14 (Anchor of an execution)**.** *Let $v \in V$ and let $\epsilon = (\gamma \underset{\tau}{\longrightarrow} \gamma_0 \underset{\tau}{\longrightarrow} \cdots)$ be an execution. We say that v is an anchor of ϵ if during ϵ, v executes a rule that is not $\mathbb{R}_{start}$, and if the first rule different from $\mathbb{R}_{start}$ executed by v is a rule of $\mathbb{R}_{cvg}$.*

We now introduce Theorem 4 that establishes that if an execution does not respect the snap property of Definition 4 then there exists an anchor of that execution. Due to the lack of place, the detail of the proof is omitted. We give only the main steps of the proof.

Theorem 4. *Let w be a node and let $\epsilon = (\gamma \underset{\tau}{\longrightarrow} \gamma_0 \underset{\tau}{\longrightarrow} \gamma_1 \underset{\tau}{\longrightarrow} \cdots)$ be a maximal execution such that w executes $\mathbb{R}_{start}$ in $\gamma \underset{\tau}{\longrightarrow} \gamma_0$. If $\gamma_s \notin \Gamma_{\mathsf{cvg}}$, then there exists a node v that is an anchor of ϵ_s.*

We first establish by contradiction that, if ϵ is an execution that starts in a configuration where $\mathcal{A}$ or $\mathcal{U}$ has not stabilized, and such that all nodes are activated, then at least one node executes a convergence rule for $\mathcal{A}$ of $\mathcal{U}$ during ϵ. This first result allows us to establish that if an execution does not respect the snap property of Definition 4 then there exists a node that executes a convergence rule for $\mathcal{A}$ of $\mathcal{U}$ during a final descent ϵ_s. Once again, we reason by contradiction, and observe that we can use Lemma 7 and Corollary 1. Finally, we prove that if there exists a node that executes a convergence rule for $\mathcal{A}$ of $\mathcal{U}$ during a final descent ϵ_s, there exists an anchor of that execution. Indeed, consider cs_a the first computing step of ϵ_s where a node executes a convergence rule, and consider v a node that executes a convergence rule during cs_a. Then, necessarily, the first activation of v during ϵ_s happens at cs_a. The combination of that last result and of the previous one terminates the proof of Theorem 4.

Theorem 4 allows us to prove Lemma 8 by contradiction: if there exists an anchor of ϵ_s then a contradiction is raised.

Lemma 8. *Let w be a node, and let $\epsilon = (\gamma \underset{\tau}{\longrightarrow} \gamma_0 \underset{\tau}{\longrightarrow} \gamma_1 \underset{\tau}{\longrightarrow} \cdots)$ be a maximal execution such that w executes $\mathbb{R}_{start}$ in $\gamma \underset{\tau}{\longrightarrow} \gamma_0$. We have: $\gamma_s \in \Gamma_{\mathsf{cvg}}$.*

In the proof of Lemma 8, we consider a causal pyramid of maximal lenght with origin w that ends near a node v, such that v executes a convergence rule before the action of the last node of the pyramid. We then prove that variable doi spreads down the pyramid from v to w, which leads to the conclusion that w executes $\mathbb{R}_{end}$ during ϵ_s, absurd.

Theorem 5. *$\mathcal{T}$ is a snap-stabilizing algorithm for the termination detection problem.*

Proof. Let ϵ be a maximal execution of $\mathcal{T}$. Suppose there exists a node w and a computing step $(cs_{ask} = \gamma_{ask-1} \underset{\tau}{\longrightarrow} \gamma_{ask}) \in \epsilon$ such that in cs_{ask}, App executes $App\colon ask$ on node w. According to Lemma 4, and since $\mathcal{U}$ is a unison algorithm, w is activated again in the ϵ-subexecution $\epsilon' = (\gamma_{ask} \underset{\tau}{\longrightarrow} \cdots)$. Let $\mathsf{cs}_0 = (\gamma \underset{\tau}{\longrightarrow} \gamma_0)$ be the first computing step of ϵ' in which w is activated. In cs_0, w executes rule $\mathbb{R}_{start}$. Thus, the ϵ'-subexecution $\epsilon'' = (\gamma \underset{\tau}{\longrightarrow} \gamma_0 \underset{\tau}{\longrightarrow} \cdots)$ corresponds to the premises of Lemma 8 that, combined with Lemma 5, guarantee that $\gamma_{\mathsf{f}(w,\epsilon)} \in \Gamma_{\mathsf{cvg}}$. By definition, w do not execute $\mathbb{R}_{stop}$ between γ_{ask} and γ. Thus, $\gamma_{\mathsf{f}(w,\epsilon)-1} \underset{\tau}{\longrightarrow} \gamma_{\mathsf{f}(w,\epsilon)}$ is the first computing step subsequent to γ_{ask} in which w executes $\mathbb{R}_{stop}$. This proves that w positively answers to the request of App, and that when it does, $\mathcal{A}$ has indeed terminated. □

D. Time complexity

Definition 15 (Full Round)**.** *Recall the sluggishness of $\mathcal{U}$ us the maximal number of transparent rules a node can execute between two normal rules, in a stabilized execution. A full round of an execution is defined as $1 + \mathcal{S}(\mathcal{U})$ rounds.*

The notion of full round is the suitable notion to evaluates the time of response of our snap-stabilizing algorithm, since Algorithm $\mathcal{U}$ is for us a blackbox. Recall that, for unison algorithms presented in [11]–[14], the notion of full round is identical to the more classical notion of round.

Theorem 6. *Let w be a node, and let $\epsilon = (\gamma \underset{\tau}{\longrightarrow} \gamma_0 \underset{\tau}{\longrightarrow} \cdots)$ be a maximal execution such that $\gamma \in \Gamma_{\mathsf{cvg}}$ and in $\gamma \underset{\tau}{\longrightarrow} \gamma_0$, w executes $\mathbb{R}_{start}$. Then $\mathsf{f}(w,\epsilon)$ occurs in $O(D)$ full rounds after γ.*

Proof. Theorem 2 guarantees that all the rounds are finite.

During one full round, all nodes v such that $\forall u \in N_v, \mathsf{clock}_u \in \{\mathsf{clock}_v, \mathsf{clock}_v + 1\}$ are activated and execute rule $\mathbb{R}_{wait}$. These activations imply that $\min_{v \in V} \mathsf{clock}_v$ increases by at least one each full round. Since at any moment

the maximal difference between the clocks of two nodes is D, we obtain that $\forall k \in \mathbb{N}$, during $D+k$ full rounds all the nodes are activated at least k times. After $D+1$ full rounds, all the nodes are activated, and the following property holds for any node v in any configuration: $\mathsf{doi}_v = \max_{u \in V} \mathsf{doi}_u \wedge \mathsf{doi}_v > 0 \Rightarrow \mathsf{UnisonOnly}(v)$. In other words, the nodes with maximal value of doi are activatable for the rule $\mathbb{R}_{wait}$. This implies that the maximal value of doi decreases by at least one in each full round. This ensures that after at most $3D+3$ full rounds, the system is in Γ_{doi}. Moreover after $D+2D+2$ more full rounds, all the nodes are activated at least $2D+2$ times, so w executes $\mathbb{R}_{cpt}$ $2D+1$ times, after what it executes rule $\mathbb{R}_{stop}$.

We established that $\mathsf{f}(w, \epsilon)$ occurs in at most $6D+5 = O(D)$ full rounds after the execution of $\mathbb{R}_{start}$ by w.

□

VI. Conclusion

We proposed a generic, deterministic, snap-stabilizing, silent algorithm that solves Termination Detection in asynchronous networks. Our solution works assuming an unfair scheduler. It has the nice feature of working in anonymous networks, but requires that each node knows (an upper bound on) the network diameter D. The space complexity of our solution is $O(\log D)$ bits per node, and provides an answer in $O(\max(k, k', D))$ rounds, where k and k' are the stabilization time complexities of the observed and the unison algorithms, respectively.

We have endeavored to provide a generic algorithm that works with any self-stabilizing unison algorithm in the literature, *e.g.*, [11]–[14]. However, the proposed solution returns positive answers only, *i.e.*, no answer is given while the observed algorithm does not effectively terminate according to its specification. Adapting our solution to provide snap-stabilizing negative answers (*i.e.*, without false negative answers) is not so easy to achieve. It seems to be dependant on stabilizing time of the unison algorithm. Therefore, we should be able to achieve this goal by developing an *ad-hoc* solution, but likely at the expense of genericity. Also, two other issues remain open in anonymous settings, namely (i) the question of necessity for nodes to known the knowledge of (an upper bound on) the diameter of the network (no matter the solution being self- or non-self-stabilizing), and (ii) the optimality in both spaces and time to solve snap-stabilizing TD in anonymous networks.

References

[1] N. Francez, "Distributed termination," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, p. 42–55, 1980. [Online]. Available: https://doi.org/10.1145/357084.357087

[2] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[3] S. Dolev, *Self-Stabilization*. MIT Press, 2000.

[4] K. Altisen, S. Devismes, S. Dubois, and F. Petit, Eds., *Introduction to Distributed Self-Stabilizing Algorithms*, ser. Synthesis Lectures on Distributed Computing. Morgan & Claypool Publishers, 2019.

[5] A. Cournier, A. K. Datta, S. Devismes, F. Petit, and V. Villain, "The expressive power of snap-stabilization," *Theor. Comput. Sci.*, vol. 626, pp. 40–66, 2016. [Online]. Available: https://doi.org/10.1016/j.tcs.2016.01.036

[6] A. Bui, A. K. Datta, F. Petit, and V. Villain, "Snap-stabilization and PIF in tree networks," *Distributed Computing*, vol. 20, no. 1, pp. 3–19, 2007.

[7] D. Angluin, "Local and global properties in networks of processors," in *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC '80)*, 1980, pp. 82–93.

[8] C. Lin and J. Simon, "Observing self-stabilization," in *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1992*, N. C. Hutchinson, Ed. ACM, 1992, pp. 113–123. [Online]. Available: https://doi.org/10.1145/135419.135444

[9] J. Beauquier, L. Pilard, and B. Rozoy, "Observing locally self-stabilization," *J. High Speed Networks*, vol. 14, no. 1, pp. 3–19, 2005. [Online]. Available: http://content.iospress.com/articles/journal-of-high-speed-networks/jhs252

[10] ——, "Observing locally self-stabilization in a probabilistic way," *J. Aerosp. Comput. Inf. Commun.*, vol. 3, no. 10, pp. 516–537, 2006. [Online]. Available: https://doi.org/10.2514/1.19858

[11] J. Couvreur, N. Francez, and M. Gouda, "Asynchronous unison," in *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems (ICDCS'92)*, 1992, pp. 486–493.

[12] C. Boulinier, F. Petit, and V. Villain, "When graph theory helps self-stabilization," in *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004*. ACM, 2004, pp. 150–159. [Online]. Available: https://doi.org/10.1145/1011767.1011790

[13] S. Devismes and C. Johnen, "Self-stabilizing distributed cooperative reset," in *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019*. IEEE, 2019, pp. 379–389. [Online]. Available: https://doi.org/10.1109/ICDCS.2019.00045

[14] Y. Emek and E. Keren, "A thin self-stabilizing asynchronous unison algorithm with applications to fault tolerant biological networks," in *PODC '21: ACM Symposium on Principles of Distributed Computing*. ACM, 2021, pp. 93–102.

[15] C. Boulinier, M. Levert, and F. Petit, "Snap-stabilizing waves in anonymous networks," in *Distributed Computing and Networking, 9th International Conference, ICDCN 2008*, ser. Lecture Notes in Computer Science, vol. 4904. Springer, 2008, pp. 191–202. [Online]. Available: https://doi.org/10.1007/978-3-540-77444-0_17

[16] G. Tel, "Total algorithms," in *Concurrency 88*, V. F. e. Springer, Ed., vol. LNCS 335. Springer-Verlag, 1988, pp. 277–291.

[17] B. K. Szymanski, Y. Shi, and N. S. Prywes, "Terminating iterative solution of simultaneous equations in distributed message passing systems," in *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985*, M. A. Malcolm and H. R. Strong, Eds. ACM, 1985, pp. 287–292.

[18] E. Godard, "Snap-stabilizing tasks in anonymous networks," *Theory Comput. Syst.*, vol. 63, no. 2, pp. 326–343, 2019.

[19] ——, "A self-stabilizing enumeration algorithm," *Inf. Process. Lett.*, vol. 82, no. 6, pp. 299–305, 2002.

[20] A. W. Mazurkiewicz, "Distributed enumeration," *Inf. Process. Lett.*, vol. 61, no. 5, pp. 233–239, 1997.

[21] S. Dolev, M. Gouda, and M. Schneider, "Memory requirements for silent stabilization," in *PODC96 Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, 1996, pp. 27–34.

[22] M. Gouda and F. Haddix, "The alternator," in *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*. IEEE Computer Society Press, 1999, pp. 48–53.

[23] C. Boulinier, F. Petit, and V. Villain, "Synchronous vs. asynchronous unison," *Algorithmica*, vol. 51, no. 1, pp. 61–80, 2008. [Online]. Available: https://doi.org/10.1007/s00453-007-9066-x