A Self-Stabilizing Memory Efficient Algorithm for the Minimum Diameter Spanning Tree under an Omnipotent Daemon[1]

Lélia Blin[a], Fadwa Boubekeur[b], Swan Dubois[c]

[a]*Université d'Evry-Val-d'Essonne, CNRS, LIP6 UMR 7606, 4 place Jussieu 75005 Paris*
E-mail: `lelia.blin@lip6.fr`
Additional supports from the ANR project IRIS
[b]*DAVID, UVSQ, Université Paris-Saclay, 45 Avenue des Etats-Unis, 78035 Versailles*
E-mail: `fadwa.boubekeur@lip6.fr`
Additional supports from the ANR project IRIS
[c]*Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6 UMR 7606, 4 place Jussieu 75005 Paris*
E-mail: `swan.dubois@lip6.fr`

Contact author:	*Swan Dubois*
Mail address:	*UPMC Sorbonne Universités*
	LIP6 - case 26-00/207
	4 place Jussieu
	75005 Paris Cedex 5
	France
E-mail address:	*swan.dubois@lip6.fr*
Tel:	*+33 1 44 27 88 77*
Fax:	*+33 1 44 27 74 95*

[1]A preliminary version of this work appears in IPDPS 2015 (see [5]).

Abstract

Routing protocols are at the core of distributed systems performances, especially in the presence of faults. A classical approach to this problem is to build a spanning tree of the distributed system. Numerous spanning tree construction algorithms depending on the optimized metric exist (total weight, height, distance with respect to a particular process, ...) both in fault-free and faulty environments. In this paper, we aim at optimizing the diameter of the spanning tree by constructing a minimum diameter spanning tree. We target environments subject to transient faults (*i.e.* faults of finite duration).

Hence, we present a self-stabilizing algorithm for the minimum diameter spanning tree construction problem in the state model. Our protocol has the following attractive features. It is the first algorithm for this problem that operates under the *unfair and distributed* adversary (or *daemon*). In other words, no restriction is made on the asynchronous behavior of the system. Second, our algorithm needs only $O(\log n)$ bits of memory per process (where n is the number of processes), that improves the previous result by a factor n. These features are not achieved to the detriment of the convergence time, which stays polynomial.

Keywords: Self-stabilization, Spanning tree, Center, Diameter, MDST, Unfair daemon

---

1. Introduction

Self-stabilization [19, 20, 43] is one of the most versatile techniques to sustain availability, reliability, and serviceability in modern distributed systems. After the occurrence of a catastrophic failure that placed the system components in some arbitrary global state, self-stabilization guarantees recovery to a correct behavior in finite time without external (*i.e.* human) intervention. This approach is particularly well-suited for self-organized or autonomic distributed systems.

In this context, one critical task of the system is to recover efficient communications. A classical way to deal with this problem is to construct a spanning tree of the system and to route messages between processes only on this structure. Depending on the constraints required on this spanning tree (*e.g.* minimal distance to a particular process, minimum flow, ...), we obtain routing protocols that optimize different metrics.

In this paper, we focus on the minimum diameter spanning tree (MDST) construction problem [32]. The MDST problem is a particular spanning tree construction in which we require spanning trees to minimize their diameters. Indeed, this approach is natural if we want to optimize the worst communication delay between any pair of processes (since this latter is bound by the diameter of the routing tree, that is minimal in the case of the MDST).

The contribution of this paper is to present a new self-stabilizing MDST algorithm that operates in any asynchronous environment, and that improves existing solutions on the memory space required per process. Namely, we decrease the best-known space complexity for this problem by a factor of n (where n is the number of processes). Note that this does not come at the price of degrading time performance.

Related works. Spanning tree construction was extensively studied in the context of distributed systems either in a fault-free setting or presence of faults. There is an extensive, literature on self-stabilizing construction of various kinds of trees, including spanning trees (ST) [15, 39], breadth-first search (BFS) trees [1, 12, 16, 24, 34], depth- first search (DFS) trees [14, 35], minimum-weight

spanning trees (MST) [38, 6], shortest-path spanning trees [30, 36], minimum-degree spanning trees [9], Steiner Tree [8], etc. A survey on self-stabilizing distributed protocols for spanning tree construction can be found in [27].

The MDST problem is closely related to the determination of centers of the system [31]. Indeed, a center is a process that minimizes its eccentricity (*i.e.* its maximum distance to any other process of the system). Then, it is well-known that a BFS spanning tree rooted to a center is an MDST. As many self-stabilizing solutions to BFS spanning tree construction exist, we focus, in the following part on the hardest part of the MDST problem: the center computation problem.

A natural way to compute the eccentricity of processes of a distributed system (and beside, to determine its centers) is to solve first the all-pairs shortest path (APSP) problem. This problem consists in computing, for any pair of processes, the distance between them in the system. This problem was extensively studied under various assumptions. For instance, [33] provides an excellent survey on recently distributed solutions to this problem and presents an almost optimal solution in synchronous settings. Note that there also exist some approximation results for this problem, *e.g.* [40, 42], but they fall outside the scope of this work since we focus on exact algorithms. In conclusion, this approach is appealing since it allows to use well-known solutions to the APSP problem, but it yields automatically to a $O(n \log n)$ space requirement per process (due to the very definition of the problem).

In contrast, only a few works focused directly on the computation of centers of a distributed system to reduce space requirement as we do in this work. In a synchronous and fault-free environment, we can cite [37] that present the first algorithm for computing directly centers of a distributed system. In a self-stabilizing setting, some works [2, 11, 17] described solutions that are specific to tree topologies. The most related work to ours is from Butelle *et al.* [13]. The self-stabilizing distributed protocol proposed in this latter makes no assumptions on the underlying topology of the system and works in asynchronous environments. Its main drawback lies in its space complexity of $O(n \log n)$ bits per process, that is equivalent to those of APSP-based solutions.

Our contribution. At the best of our knowledge, the question whether it is possible to compute centers of any distributed system in a self-stabilizing way using only a sublinear memory per process is still open. Our main contribution is to answer positively to this question by providing a new deterministic self-stabilizing algorithm that requires only $O(\log n)$ bits per process, which improves the current results by a factor n. Moreover, our algorithm is suitable for any asynchronous environment since we do not make any assumption on the adversary (or daemon) and has a convergence time in $O(n^2)$ rounds (that is comparable to existing solutions [13]).

Organization of the paper. This paper is organized as follows. In section 2, we formalize the model used afterwards. Section 3 is devoted to the description of our algorithm while Section 4 contains its correctness proof. Finally, we discuss some open questions in Section 5.

2. Model and Definitions

State model. We model the system as an undirected connected graph $G = (V, E)$ where V is a set of processes and E is a binary relation that denotes the ability for two processes to communicate, *i.e.* $(u, v) \in E$ if and only if u and v are *neighbors*. We consider only *identified* systems (*i.e.* there exists a unique identifier $\textsc{id}_v$ for each process v taken in the set $[0, n^c]$ for some constant c). The set of all neighbors of v, called its *neighborhood*, is denoted by N_v. In the following, n denotes the number of processes of the network.

We consider the classical *state model* (see [20]) where communications between neighbors are modeled by direct reading of variables instead of an exchange of messages. Every process has a set of shared variables (henceforth, referred to as *variables*). A process v can write to its own variables only, and read its own variables and those of its neighbors. The state of a process is defined by the current value of its variables. The state of a system (*a.k.a.* the *configuration*) is the product of the states of all processes. We denote by Γ the set of all configurations of the system. The algorithm of every process is a finite set of *rules*. 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 algorithm of v is a boolean predicate involving variables of v and its neighbors. The statement of a rule of v updates one or more variables of v. A statement can be executed only if the corresponding guard is satisfied (*i.e.* it evaluates to true). The process rule is then *enabled*, and process v is *enabled* in $\gamma \in \Gamma$ if and only if at least one rule is enabled for v in γ.

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

The asynchronism of the system is modeled by an *adversary* (*a.k.a. daemon*) that chooses, at each step, the subset of enabled processes that are allowed to execute one of their rules during this step (we say that such processes are activated during the step). The literature proposed a lot of daemons depending on their characteristics (like fairness, distribution, ...), see [26] for a taxonomy of these daemons. Note that we assume an *unfair distributed daemon* in this work. This daemon is the most challenging since we made no assumption of the subset of enabled processes chosen by the daemon at each step. We only require this set to be non-empty if the set of enabled processes is not empty in order to guarantee progress of the algorithm.

To compute time complexities, we use the definition of round [23]. This definition captures the execution rate of the slowest process in any execution. The first round of $\epsilon \in \mathcal{E}$, noted ϵ', is the minimal prefix of ϵ containing the execution of one action or the neutralization of every enabled process in the initial configuration. Let ϵ'' be the suffix of ϵ such that $\epsilon = \epsilon'\epsilon''$. The second round of ϵ is the first round of ϵ'', and so on.

Self-stabilization. Let $\mathcal{P}$ be a problem to solve. A *specification* of $\mathcal{P}$ is a predicate that is satisfied by every execution in which $\mathcal{P}$ is solved. We recall the definition of self-stabilization.

Definition 1 (Self-stabilization [19]) *Let $\mathcal{P}$ be a problem, and $\mathcal{S}_\mathcal{P}$ a specification of $\mathcal{P}$. An algorithm $\mathcal{A}$ is self-stabilizing for $\mathcal{S}_\mathcal{P}$ if and only if for every configuration $\gamma_0 \in \Gamma$, for every execution $\epsilon = \gamma_0\gamma_1 \ldots$, there exists a finite prefix $\gamma_0\gamma_1 \ldots \gamma_l$ of ϵ such that every execution of $\mathcal{A}$ starting from γ_l satisfies $\mathcal{S}_\mathcal{P}$.*

3. Presentation of the Algorithm

In this section, we present our self-stabilizing algorithm for the computation of centers of the distributed system, named $\mathcal{SSCC}$ (for $\mathcal{S}$elf $\mathcal{S}$tabilizing $\mathcal{C}$enters $\mathcal{C}$omputation). We organize this section in the following way. First, we give a global overview of our algorithm in Section 3.1. Then, Sections 3.2, 3.3, 3.4, and 3.5 are devoted to the detailed presentation of each module of our algorithm, respectively a leader election module, a token circulation module, an eccentricity computation module, and finally the center computation module.

3.1. High-level Description

Our algorithm is based on several layers, each of them performing a specific task. Of course, these layers operate concurrently but, for the clarity of presentation, we present them sequentially in a "down-to-top" order.

The first layer is devoted to the construction of a *rooted spanning tree*. As we have a uniform model (that is, all processes execute the same self-stabilizing algorithm), there is no *a priori* distinguished process that may take the role of the root of the system. Therefore, we need first to elect a leader. We use an algorithm that performs such an election and constructs a spanning tree in the same time. To our knowledge, only the algorithm proposed by [18] corresponds to our criteria regarding daemon, memory requirement and convergence time. Indeed, Datta *et al.* [18] designed a self-stabilizing algorithm to construct a BFS tree rooted at the process of minimum identity. This algorithm self-stabilizes even under the distributed unfair daemon, it uses $O(\log n)$ bits of memory per

process, and it converges in $O(n)$ rounds. Throughout the rest of the paper, we call the BFS tree constructed by this first layer the `Backbone` of the system.

The second layer is a *token circulation* on the `Backbone`. Along with all existing self-stabilizing algorithms for token circulation, we choose to slightly adapt an algorithm of Petit and Villain [41]. The aim of this token circulation is to synchronize the temporal multiplexing of variables of the third layer of our algorithm that computes the eccentricity of each process. Indeed, to reduce the space requirement of our algorithm to $O(\log n)$ bits per process, all processes compute their eccentricities using the same variables, but one at a time and in a sequential fashion. To avoid conflicts, we manage this mutual exclusion by a token circulation. In more details, we distinguish, for each process, the forward token circulation (that is, the process gets the token from its parent in the `Backbone`) and the backward token circulations (that is, the process gets back the token from one of its children in the `Backbone`).

A process starts the execution of the third layer of our algorithm (that is, *computation of process eccentricity*) only on the forward token circulation. On a backward token circulation, the process sends the token to the following process in the `Backbone` in a DFS order, without performing any extra task. A process v computes its eccentricity in the following way. When it receives the forward token, v starts a self-stabilizing BFS tree construction rooted at itself. We denote this BFS by $\texttt{BFS}(v)$. Once the construction of $\texttt{BFS}(v)$ is done, the leaves of this tree start a feedback phase that consists in propagating back to v the maximum depth of a process in $\texttt{BFS}(v)$ (which is exactly the eccentricity of v). Once process v has collected its eccentricity, it releases the token to the following process in the `Backbone`.

Finally, the fourth layer aim is the *center determination*. The minimum eccentricity (computed for each process by the third layer) is collected all the time from the leaves of the `Backbone` to its root. Then, the root propagates this minimum eccentricity to all processes along the `Backbone`. The processes with the minimum eccentricity become centers. Also, among the centers, we elect the one with the highest identity to be the root of the minimum diameter spanning tree.

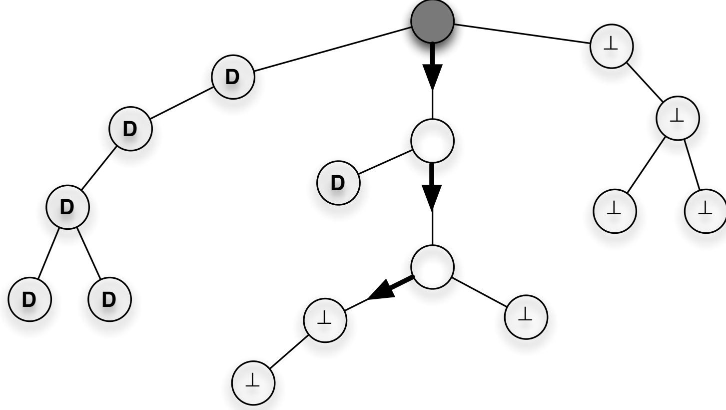

Figure 1: Illustration of the variable `next` of the token circulation layer. The gray process is the root of the `Backbone`. D stands for the value done.

We claim that each layer of this algorithm stack is self-stabilizing and that their composition self-stabilizes to a minimum diameter spanning tree of the system under the distributed unfair daemon within $O(n^2)$ rounds. As each layer of our algorithm needs at most $O(\log n)$ bits per process, we obtain the desired space complexity.

3.2. Leader Election and Spanning Tree Construction

The first layer of our algorithm executes a self-stabilizing algorithm from Datta *et al.* [18] that elects the process of smallest identity in the system and constructs a BFS spanning tree rooted at this leader. This algorithm works under the distributed unfair daemon, uses $O(\log n)$ bits of memory per process, and stabilizes within $O(n)$ rounds.

Since we use this algorithm as a black box, we do not need to present it formally here. The interested reader is referred to the original paper [18]. Remember that we call `Backbone` the BFS tree built by this layer of our algorithm. For the remainder of the presentation, we denote the parent of any process v in the `Backbone` by p_v, the set of children of v in the `Backbone` by $\text{child}(v)$, and the set of neighbors of v in the `Backbone` by $\text{N}_{\texttt{Backbone}}(v)$. That is, if the `Backbone` is defined by the 1-factor $\{(v, \mathsf{p}_v), v \in V\}$, then we have $\text{child}(v) = \{u \in V : \mathsf{p}_u = v\}$

and $\mathrm{N}_{\mathtt{Backbone}}(v) = \mathrm{child}(v) \cup \{\mathsf{p}_v\}$. We also define the predicate $\mathrm{BRoot}(v)$ over variables of this layer. This predicate is true if and only if the process v is the root of the Backbone.

It is important to note that the construction of the backbone has higher priority than the other layers of our algorithm (that is, token circulation, eccentricity computation, and centers determination). In other words, if a process v has a neighbor with a different root in the Backbone or a neighbor with an incoherent distance in the Backbone, v cannot execute a rule related to any other layer. This priority is needed for our algorithm to operate under an unfair daemon.

3.3. Token Circulation

The second layer of our algorithm is a slight adaptation of a self-stabilizing token circulation algorithm by Petit and Villain [41]. The (eventually unique) token circulates infinitely often over the Backbone in a DFS order. This algorithm operates under the distributed unfair daemon, it uses $O(\log n)$ bits of memory per process and converges in $O(n)$ rounds.

This algorithm uses only one variable for each process v: $\mathsf{next}_v \in \{\bot, \mathrm{done}, N_v\}$. This variable stores the state of the process on the current token circulation. The value $\bot$ means that the process has not already been visited by the token during the current circulation. When next_v points to a child of v in the Backbone, that means that v has been already visited during the current circulation by the token and that v sent the token to its child pointed by next_v. The value done means that the process and all its children in the Backbone have already been visited by the token during the current circulation. In other words, the token is held by the first process v with $\mathsf{next}_v = \bot$ along the path issued from the root of the Backbone following (non-$\bot$ and non-done) next variables. Refer to Figure 1 for an illustration. We define a total order $\succ$ over the values of variable next by extending the natural order $>$ over identities with the following assumption: for any process v, we have $\mathrm{done} \succ \mathrm{ID}_v \succ \bot$.

The formal presentation of this layer is provided in Algorithm 1, the presentation of functions and predicates used by this algorithm is postponed in the

detailed description (see subsection 3.3.1). This algorithm consists of two rules. The first one, $\mathbb{R}_{\texttt{ErToken}}$, is used to perform the convergence towards a unique token and the reset of the next variables at the completion of a token circulation while the second one, $\mathbb{R}_{\texttt{Backward}}$, performs the backward circulation of the token. Note that the forward circulation rule is left to the next layer of our protocol (see below for more explanations on the relationship between these two layers).

Algorithm 1 Token circulation for process v

$$\mathbb{R}_{\texttt{ErToken}} : \text{ErValues}(v) \vee \big(\neg\text{BRoot}(v) \wedge (\mathsf{next}_{\mathsf{p}_v} \neq v) \wedge (\mathsf{next}_v \in \text{child}(v) \cup \{\text{done}\})\big) \rightarrow \mathsf{next}_v := \bot;$$

$$\mathbb{R}_{\texttt{Backward}} : \neg\text{ErValues}(v) \wedge (\text{BackNd}(v) \vee \text{BackR}(v)) \wedge (\mathsf{R}_{\text{BFS}_{\text{Next}(v)}} \neq \textsc{id}(\text{Next}(v))) \rightarrow \mathsf{next}_v := \text{Next}(v);$$

We modify the original algorithm of [41] in the following way. When a process v receives the token, this latter is blocked by v until the third layer of our protocol computes the eccentricity of this process. This is done by the construction of a BFS tree rooted at v and by the gathering of the maximum distance between v and any other process in the system (refer to Section 3.4 for more details on this layer). This is the reason why the forward token circulation is not performed by a rule of Algorithm 1 but by the rule $\mathbb{R}_{\texttt{EndBFS}}$ in Algorithm 2 (that described the third layer of our algorithm). Communication between these two layers on the state of the token is performed using the predicate TokenD(v). When the token circulation layer gives the token to process v, this predicate becomes true, that allows the eccentricity computation layer to start. Once the eccentricity of process v is computed, this latter updates $next_v$ (see $\mathbb{R}_{\texttt{EndBFS}}$ in Algorithm 2) that perform the forward token circulation (exactly as in the original algorithm of [41]).

We also slightly modify the backward token circulation to ensure the convergence of the eccentricity computation layer. If the process v wants to send the token in a backward circulation to a neighbor u, v has to wait in the case where u is currently computing its own eccentricity (this situation is possible if u wrongly believes to have the token). We perform this waiting using a vari-

able $\mathsf{R}_{\mathsf{BFS}u}$ dedicated to the BFS construction (see Section 3.4 for the definition of this variable). This variable stores the identity of the root of the BFS tree construction in which u is currently involved. Then, v postpones its backward circulation to u until $\mathsf{R}_{\mathsf{BFS}u}$ is equal to u. If u wrongly believes to have the token, it can detect it locally and to correct this error within a finite time (see Section 3.4), that ensures us that the token is never infinitely blocked by v.

The composition between the backbone construction layer and the token circulation layer of our algorithm must withstand the unfairness of the daemon. Indeed, we have to ensure that the daemon cannot choose exclusively processes enabled only for the token circulation (recall that we assume that the backbone construction has priority over the token circulation) since this may lead to a starvation of the backbone construction. To deal with this issue, we choose to block the token circulation at process v if v has a neighbor that do not belong to the backbone or if v detects an inconsistency between distances in the backbone (refer to the definition and the use of predicate ErValues). Since, before the stabilization of the backbone, the overlay structure induced by variables p_v for all $v \in V$ may be composed only of subtrees or cycles, we can ensure the starvation-freedom of the backbone construction. Indeed, the daemon cannot activate infinitely the token circulation rules of processes in a given subtree because at least one of them has a neighbor that does not belong to the same subtree, which blocks the token at this process. Similarly, the daemon cannot activate infinitely the token circulation rules of processes in a given cycle because the token detects an inconsistency with distances in the backbone and then gives the priority to the backbone construction.

3.3.1. Detailed description of the Token Circulation

We start by the detailed description of the rule rule $\mathbb{R}_{\texttt{ErToken}}$. Remember that the rule $\mathbb{R}_{\texttt{ErToken}}$ is used to perform the convergence towards a unique token but also to reset the next variables to $\perp$ at the end of each token circulation.

$$\mathbb{R}_{\texttt{ErToken}}: \text{ErValues}(v) \vee \big(\neg\text{BRoot}(v) \wedge (\mathsf{next}_{\mathsf{p}_v} \neq v) \wedge (\mathsf{next}_v \in \text{child}(v) \cup \{\text{done}\})\big) \longrightarrow \mathsf{next}_v := \perp; \quad (1)$$

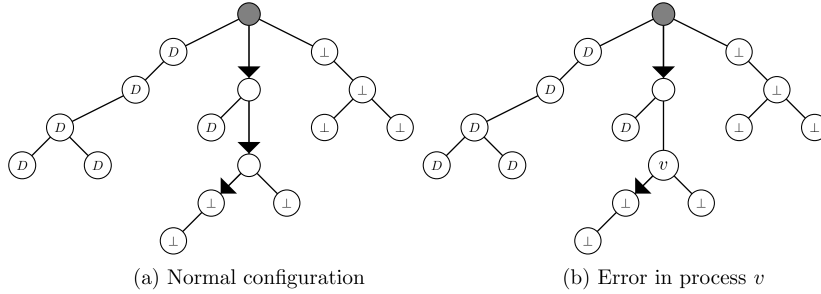
(a) Normal configuration (b) Error in process v

Figure 2: Token circulation

This rule is dedicated to the detection of an error by process v. When an error is detected by v, it deletes its variable next_v. The predicate ErValues(v) is dedicated to detecting an error in process v. The value $\perp$ means that the process has not already been visited by the token. When next_v points to a children of v in the `Backbone`, that means that v has been visited by the token and that v sent the token to its child pointed by next_v. The value done means that the process and all its children in the `Backbone` have already been visited by the token. So, if the token is not equal of one of these variables (*e.g.* towards a neighbor of v that do not belong to child(v)) then an error is detected.

$$\text{ErValues}(v) \equiv \mathsf{next}_v \notin \text{child}(v) \cup \{\perp, \text{done}\} \tag{2}$$

The second part of the guard of rule $\mathbb{R}_{\texttt{ErToken}}$ is dedicated to detecting an other inconsistency of the variables dedicated to the token circulation. Indeed, if a process v is not the root of the `Backbone` and its variable next_v points toward one of its children that mean the token is kept by one of these descendants, so its parent must point toward itself (see Figure 2(b)). Finally, this rule also detects the case where the token already explores the whole subtree rooted to the parent of the process and then set its `next` variable to $\perp$ to prepare the next token circulation. Indeed, in this case, we have: $\mathsf{next}_v = \text{done}$ and $\mathsf{next}_{\mathsf{p}_v} = \text{done}$ and the execution of the rule reset next_v to $\perp$.

The rule $\mathbb{R}_{\texttt{Backward}}$ performs the backward circulation of the token. Remem-

ber that the forward circulation rule is left to the next layer of our protocol.

$$\mathbb{R}_{\texttt{Backward}}: \neg\text{ErValues}(v) \wedge (\text{BackNd}(v) \vee \text{BackR}(v)) \wedge \\ (\mathsf{R}_{\text{BFS}_{\text{Next}(v)}} \neq \text{ID}(\text{Next}(v))) \longrightarrow \mathsf{next}_v := \text{Next}(v); \tag{3}$$

The rule is activated if and only if the process v is not in error (see predicate $\neg$ErValues(v)). To explain this rule, we need to explain first the function Next(v). This function returns the next neighbor of v to which send the token (in a round-robin fashion on identities) or done is the whole subtree rooted at the process was already explored. Note that this function also allows the root of the tree to initiate the reset of next variables to $\perp$ when it locally detects the termination of a token circulation.

$$\text{Next}(v) = \begin{cases} u & \text{if } \exists u \in \text{child}(v), \text{ID}_u = \min\{\text{ID}_w \mid w \in \text{child}(v) \wedge (\text{ID}_w \succ \mathsf{next}_v)\} \\ \perp & \text{if BRoot}(v) \wedge \mathsf{next}_v \in \text{child}(v) \wedge (\forall u \in \text{child}(v), \mathsf{next}_v \succ \text{ID}_u \vee \mathsf{next}_v = \text{ID}_u) \\ \text{done} & \text{otherwise} \end{cases} \tag{4}$$

The predicates BackR(v) and BackNd(v) used in the guard of $\mathbb{R}_{\texttt{Backward}}$ (v denotes respectively the root of the Backbone and another process) are basically used to detect when the token has finished its circulation in the subtree rooted at the process currently pointed by next_v (indicated by the done value).

$$\text{BackR}(v) \equiv \text{BRoot}(v) \wedge (\mathsf{next}_v = u) \wedge (\mathsf{next}_u = \text{done}) \wedge \text{PermR}(v) \tag{5}$$

$$\text{BackNd}(v) \equiv \neg\text{BRoot}(v) \wedge (\mathsf{next}_{\mathsf{p}_v} = v) \wedge (\mathsf{next}_v \in \text{child}(v)) \wedge \\ (\mathsf{next}_{\text{Next}(v)} = \text{done}) \wedge \text{PermNd}(v) \tag{6}$$

These predicates make respectively use of PermR(v) and PermNd(v). These latter are used to ensure that the process u returned by the application of the function Next(v) satisfies $\mathsf{next}_u = \perp$ (to avoid the sending of the token on a subtree in an illegal state).

$$\text{PermR}(v) \equiv (\text{Next}(v) = \perp) \vee (\mathsf{next}_{\text{Next}(v)} = \perp) \tag{7}$$

$$\text{PermNd}(v) \equiv (\text{Next}(v) = \text{done}) \vee (\mathsf{next}_{\text{Next}(v)} = \bot) \tag{8}$$

The communication between the token circulation and the eccentricity computation is performed using the predicate TokenD(v). When the token circulation layer gives the token to process v, this predicate becomes true, that allows the eccentricity computation layer to start. Once the eccentricity of process v is computed, this latter updates $next_v$ that performs the forward token circulation (exactly as in the original algorithm of [41]).

$$\begin{aligned}\text{TokenD}(v) \equiv (\mathsf{next}_v = \bot) \wedge \quad & \big((\text{BRoot}(v) \wedge \text{PermR}(v)) \vee \\ & (\neg\text{BRoot}(v) \wedge (\mathsf{next}_{\mathsf{p}_v} = v) \wedge \text{PermNd}(v)\big)\end{aligned} \tag{9}$$

3.4. Eccentricity Computation

The third layer of our algorithm is devoted to the computation of the eccentricity of each process. Recall that the token circulation performed by the second layer eventually ensures that at most one process computes its eccentricity at a time (that allows us to re-use the same variables). Roughly speaking, the eccentricity of each process is computed as follows. First, the process starts the construction of a BFS spanning tree rooted at itself. Once done, we gather the maximum distance in this tree (namely, the eccentricity of its root) from its leaves. Then, the process obtains its eccentricity and releases the token. This algorithm works under the distributed unfair daemon, uses $O(\log n)$ bits of memory per process, and stabilizes within $O(n)$ rounds (for the computation of the eccentricity of one process).

For the clarity of the presentation, let us denote by r a process that obtains the token at a given time (that is, TokenD(r)=true from this time up to the release of the token by this process). Then, our algorithm starts the construction of `BFS`(r) (the BFS spanning tree rooted at r). The first step is to inform all processes of the identity of r in order to synchronize their eccentricity computation algorithms. We do that by broadcasting the identity of r along the `Backbone` (we perform this broadcast by re-orienting temporarily the `Backbone` towards r) Then, we use a classical BFS spanning tree construction borrowed from [34] that consists, for each process, to choose as its parent in the

tree the process among its neighbors that proposes the smallest distance to the root (obviously, the process updates then its own distance to be consistent with the one of its new parent).

The delicate part is to collect the maximum distance to the root after the stabilization of BFS(r) (and not earlier). As we already said, this gathering is made by a wave from the leaves to the root of BFS(r). Each leave of BFS(r) propagates to its parent its own eccentricity value while other processes propagate to their parent the maximum between the eccentricity values of their children in BFS(r). Due to the asynchrony of the system, some difficulties may appear during this process. Indeed, if we collect an eccentricity value in a branch of BFS(r) whereas this branch is not yet stabilized (that is, some processes may still join it), we can obtain a wrong eccentricity for the process r. In this case, r may release the token earlier than expected. To prevent that, we manage the gathering of the maximum distance in the following way. When a process v changes its distance in BFS(r), this process "cleans" its eccentricity variable (that is, it erases the current value of this variable and replaces it with a specific value). Then, all processes on the path of BFS(r) between r and v clean their eccentricity variables in an upward process. In other words, we maintain at least one path in BFS(r) in which all the eccentricity variables are cleaned until BFS(r) is stabilized. The existence of this path ensures us that r does not obtain its eccentricity and release the token precociously.

We are now in measure to present formally our eccentricity computation algorithm. First, recall that, in order to broadcast the identity of the process that the algorithm currently computes the eccentricity, we need to re-orientate the Backbone to root it at the process r with TokenD(r)=true. We call this oriented tree Backbone(r). For this purpose, we introduce the function p_next(v) for each process v. This function returns the identity of the neighbor of v that belongs to the path of Backbone from v to r. More precisely, p_next is defined

as follow:

$$\text{p_next}(v) = \begin{cases} \mathsf{p}_v & \text{if TokenD}(v) = false \wedge \mathsf{next}_v \in \{\bot, \text{done}\} \\ \bot & \text{if TokenD}(v) = true \\ \mathsf{next}_v & \text{otherwise} \end{cases} \tag{10}$$

We define also the function chNext(v) that returns the set of children of v in $\texttt{Backbone}(r)$, *i.e.* neighbors u of v satisfying p_next$(u) = v$.

Our algorithm uses the following variables for each process v in order to construct $\texttt{BFS}(r)$ and to compute the eccentricity of r:

- $\mathsf{Ecc}_v \in \mathbb{N} \cup \{\bot\}$ is the eccentricity of process v;
- $\mathsf{R}_{\mathsf{BFS}_v} \in \mathbb{N}$ is the identity of the root in $\texttt{BFS}(r)$;
- $\mathsf{P}_{\mathsf{BFS}_v} \in \mathbb{N} \cup \{\bot\}$ is the identity of the parent of v in $\texttt{BFS}(r)$;
- $\mathsf{d}_{\mathsf{BFS}_v} \in \mathbb{N} \cup \{\bot, \infty\}$ is the distance between v and the root in $\texttt{BFS}(r)$;
- $\mathsf{D}_{\mathsf{BFS}_v} \in \mathbb{N} \cup \{\bot, \downarrow, \uparrow\}$ is the maximum distance between the root r and the farthest leaf in the sub-tree of v in $\texttt{BFS}(r)$;

Now, we can present rules of the third layer of our algorithm. These rules make use of some predicates and functions whose descriptions are postponed in subsection 3.4.1. For the clarity of presentation, we split the rules of our algorithm in two sets. The first one (refer to Algorithm 2) contains rules enabled for a process that holds the token (that is, a process v such that TokenD$(v) = true$) while the second one (refer to Algorithm 3) described rules for a process that does not hold the token

We discuss first of rules enabled only when the process holds the token presented in Algorithm 2. Recall that these rules are applied only when the process receives the token in a forward circulation (refer to Section 3.3). Once the process r received the token, its predicate TokenD(r) becomes true. In this state, the process r can apply only three rules.

The rule $\mathbb{R}_{\texttt{StartBFS}}$ starts the computation of $\texttt{BFS}(r)$ since r takes a state indicating that it is the root of the current BFS spanning tree. The rule $\mathbb{R}_{\texttt{CleanEcc}}$ cleans the eccentricity variable of r when needed (that is, after a fake compu-

Algorithm 2 Computation of BFS and eccentricity for a process v such that $\text{TokenD}(v) = true$

$$\mathbb{R}_{\texttt{StartBFS}} : \neg\text{RootBFS}(v) \longrightarrow (\mathsf{R}_{\mathsf{BFS}_v}, \mathsf{d}_{\mathsf{BFS}_v}, \mathsf{D}_{\mathsf{BFS}_v}) := (\text{ID}_v, 0, \bot)$$

$$\mathbb{R}_{\texttt{CleanEcc}}(v) : \text{RootBFS}(v) \wedge \text{SameBFS}(v) \wedge (\text{Down}(v) \vee \text{Bot}(v)) \longrightarrow \mathsf{D}_{\mathsf{BFS}_v} := \text{DCE}(v)$$

$$\mathbb{R}_{\texttt{EndBFS}} : \text{RootBFS}(v) \wedge \text{SameBFS}(v) \wedge \text{ChN}(v) \longrightarrow \mathsf{Ecc}_v := \text{Dis}(v); \mathsf{next}_v := \text{Next}(v); \mathsf{MinEUP}_v := \mathsf{MinEcc}(v);$$

tation of eccentricity due to the asynchrony of the system). Finally, the rule $\mathbb{R}_{\texttt{EndBFS}}$ is executed when the leaves-to-root propagation of the eccentricity is over. This rule computes the eccentricity of r, releases the token by updating the variable next_r of the token circulation layer (see Section 3.3), and updates one variable for communicating the new eccentricity to the centers determination layer (see Section 3.5 for more details on the use of this variable).

We then focus on rules enabled when the process does not hold the token presented in Algorithm 3. The first rule $\mathbb{R}_{\texttt{Tree}}$ is dedicated to flood the identity of the root r of the current BFS spanning tree along the `Backbone`. This flooding is possible since we re-orientate the `Backbone` (refer to the definition of p_next above). In the same time, the rule $\mathbb{R}_{\texttt{Tree}}$ also detects some local errors. As an example, the variable $\mathsf{D}_{\mathsf{BFS}_v}$ of a process (used to collect the eccentricity of r) must not have an integer value if one of the children (in $\texttt{BFS}(r)$) of this process is not in the same case. The rule $\mathbb{R}_{\texttt{BFS}}$ performs the BFS construction in itself. Finally, the rule $\mathbb{R}_{\texttt{Ecc}}$ deals with the tricky phase of eccentricity leaves-to-root gathering with the cleaning mechanism explained above. These rules are implemented with the help of predicates and functions defined and detailed in subsection 3.4.1.

3.4.1. Detailed description of eccentricity computation

In this section, we detail the rules of Algorithms 2 and 3. The rule $\mathbb{R}_{\texttt{StartBFS}}$ is executed only by the process v with $\text{TokenD}(v) = true$. The process v starts

Algorithm 3 Computation of BFS and eccentricity for process v with TokenD$(v) = false$

$$\mathbb{R}_{\texttt{Tree}} : \neg\text{GoodBFS}(v) \longrightarrow (\mathsf{R}_{\mathsf{BFS}v}, \mathsf{P}_{\mathsf{BFS}v}, \mathsf{d}_{\mathsf{BFS}v}, \mathsf{D}_{\mathsf{BFS}v}) := (\mathsf{R}_{\mathsf{BFS}\,\text{p_next}(v)}, \bot, \infty, \bot)$$

$$\mathbb{R}_{\texttt{BFS}} : \text{GoodBFS}(v) \wedge \text{SameBFS}(v) \wedge (\text{Best}(v) \neq \bot) \longrightarrow (\mathsf{P}_{\mathsf{BFS}v}, \mathsf{d}_{\mathsf{BFS}v}, \mathsf{D}_{\mathsf{BFS}v}) := (\text{Best}(v), \mathsf{d}_{\mathsf{BFS}\,\text{Best}(v)} + 1, \text{DBFS}(v));$$

$$\mathbb{R}_{\texttt{Ecc}} : \text{GoodBFS}(v) \wedge \text{SameBFS}(v) \wedge (\forall u \in N_v : \mathsf{d}_{\mathsf{BFS}u} \neq \infty) \wedge (\text{Best}(v) = \bot) \wedge (\mathsf{D}_{\mathsf{BFS}v} \neq \text{Dis}(v)) \longrightarrow \mathsf{D}_{\mathsf{BFS}v} := \text{Dis}(v)$$

the computation of $\texttt{BFS}(v)$, the variables dedicated to the BFS computation take values corresponding to the root. Remember that $\mathsf{R}_{\mathsf{BFS}v}$ stores the identity of the root in the current BFS, $\mathsf{d}_{\mathsf{BFS}v}$ is the distance from the root to v in the current BFS and $\mathsf{D}_{\mathsf{BFS}v}$ is used to compute the eccentricity of the root of the current BFS. To achieve that, the process v checks if the root of the BFS is itself, if it has not a parent, and if its distance is equal to zero with the following predicate.

$$\text{RootBFS}(v) \equiv (\mathsf{R}_{\mathsf{BFS}v}, \mathsf{P}_{\mathsf{BFS}v}, \mathsf{d}_{\mathsf{BFS}v}) = (\text{ID}(v), \bot, 0) \tag{11}$$

If not, the rule $\mathbb{R}_{\texttt{StartBFS}}$ assigns the variables $\mathsf{R}_{\mathsf{BFS}v}, \mathsf{P}_{\mathsf{BFS}v}, \mathsf{d}_{\mathsf{BFS}v}$ in this way.

$$\mathbb{R}_{\texttt{StartBFS}} : \neg\text{RootBFS}(v) \longrightarrow (\mathsf{R}_{\mathsf{BFS}v}, \mathsf{d}_{\mathsf{BFS}v}, \mathsf{D}_{\mathsf{BFS}v}) := (\text{ID}_v, 0, \bot) \tag{12}$$

As previously said, the rule $\mathbb{R}_{\texttt{Tree}}$ allows the flooding of the identity of the root r of the current BFS spanning tree along the `Backbone` and the detection of some local errors. To achieve the computation of the current BFS, the first step is to clean all the variables relative to the previous BFS.

$$\mathbb{R}_{\texttt{Tree}} : \neg\text{GoodBFS}(v) \longrightarrow (\mathsf{R}_{\mathsf{BFS}v}, \mathsf{P}_{\mathsf{BFS}v}, \mathsf{d}_{\mathsf{BFS}v}, \mathsf{D}_{\mathsf{BFS}v}) := (\mathsf{R}_{\mathsf{BFS}\,\text{p_next}(v)}, \bot, \infty, \bot) \tag{13}$$

The predicate GoodBFS(v) checks if the root BFS is the same as its neighbors in the `Backbone` that leads to the token. In others words, $\mathsf{R}_{\mathsf{BFS}v}$ must be equals

to the identity of the process holding the token.

$$\text{GoodBFS}(v) \equiv (\mathsf{R}_{\mathsf{BFS}v} = \mathsf{R}_{\mathsf{BFS}\mathrm{p_next}_v}) \tag{14}$$

To compute the current BFS and the eccentricity, all processes must consider as root of the BFS the process that keep the token. Hence, all other rules are enabled only when GoodBFS(v) is satisifed. The rule $\mathbb{R}_{\mathsf{BFS}}$ performs the BFS construction in itself.

$$\begin{aligned}\mathbb{R}_{\mathsf{BFS}} \,:\, &\text{GoodBFS}(v) \wedge \text{SameBFS}(v) \wedge (\text{Best}(v) \neq \bot)\\ &\longrightarrow (\mathsf{P}_{\mathsf{BFS}v}, \mathsf{d}_{\mathsf{BFS}v}, \mathsf{D}_{\mathsf{BFS}v}) := (\text{Best}(v), \mathsf{d}_{\mathsf{BFS}\text{Best}(v)} + 1, \text{DBFS}(v));\end{aligned} \tag{15}$$

$$\text{DBFS}(v) = \begin{cases} \uparrow & \text{if } \mathsf{D}_{\mathsf{BFS}v} \in \mathbb{N} \\ \downarrow & \text{otherwise} \end{cases} \tag{16}$$

The predicate SameBFS(v) is satisfied when all the neighbors of v are in the same BFS tree (*i.e.* when all its neighbors have the same root than v).

$$\text{SameBFS}(v) \equiv (\forall u \in N_v, (\mathsf{R}_{\mathsf{BFS}u} = \mathsf{R}_{\mathsf{BFS}v})) \tag{17}$$

The rule $\mathbb{R}_{\mathsf{BFS}}$ makes use of the function Best(v) that returns a neighbor of v with a better distance than it if such a neighbor exists, $\bot$ otherwise.

$$\text{Best}(v) = \begin{cases} p = \min\{\text{ID}(u) \mid \mathsf{d}_{\mathsf{BFS}u} = \min\{\mathsf{d}_{\mathsf{BFS}w} \mid w \in N_v\}\} \\ \qquad\qquad\qquad\qquad \text{if } \big((p \neq \mathsf{P}_{\mathsf{BFS}v}) \vee (\mathsf{d}_{\mathsf{BFS}v} \neq \mathsf{d}_{\mathsf{BFS}p} + 1)\big) \\ \bot \text{ otherwise} \end{cases} \tag{18}$$

If the process v has a better parent p in the current BFS, v changes its variables relative to the BFS according to p by executing the rule $\mathbb{R}_{\mathsf{BFS}}$.

The rule $\mathbb{R}_{\mathsf{Ecc}}$ deals with the tricky phase of eccentricity leaves-to-root gath-

ering.

$$\mathbb{R}_{\mathsf{Ecc}} : \text{GoodBFS}(v) \wedge \text{SameBFS}(v) \wedge (\forall u \in N_v : \mathsf{d}_{\mathsf{BFS}_u} \neq \infty) \\ \wedge (\text{Best}(v) = \bot) \wedge (\mathsf{D}_{\mathsf{BFS}_v} \neq \text{Dis}(v)) \\ \longrightarrow \mathsf{D}_{\mathsf{BFS}_v} := \text{Dis}(v) \tag{19}$$

Due to the asynchrony of the system, the process v can store a fake eccentricity. Let us first describe how our algorithm manage this from a global point of view. We add three values to $\mathsf{D}_{\mathsf{BFS}_v}$: the values $\bot$, $\uparrow$, and $\downarrow$. The value $\bot$ is dedicated to resetting the variable $\mathsf{D}_{\mathsf{BFS}_v}$ in a root-to-leave wave either when the root of the BFS changes or when a fake eccentricity is detected. In our algorithm, a process can compute its eccentricity only when its eccentricity variable and the ones of its ancestors have the value $\downarrow$ and all its descendants have already computed their eccentricities (inducing a leave-to-root computation of the eccentricities). Hence, the reset wave is followed by a root-to-leave wave that transitions the $\mathsf{D}_{\mathsf{BFS}_v}$ variable forms the value $\bot$ to $\downarrow$. When a process v detects an inconsistency in eccentricity computation, it puts its variable $\mathsf{D}_{\mathsf{BFS}_v}$ to $\uparrow$. This value is then propagated to the whole system from neighbors to neighbors, suspending the eccentricity computation. When all the neighbors of the root reach the value $\uparrow$, the root restarts the reset root-to-leave wave (*i.e* propagate the value $\bot$).

We are now in measure to present in more details predicates and functions used to implement this mechanism. The eccentricity of a process v is the farther distance from it to its descendants. Then, we compute these distances the leaves of the BFS tree to its root. When a process v is a leaf or all its children in the current BFS have computed their farther distances, the process v can compute its farther distance $\mathsf{D}_{\mathsf{BFS}_v}$. The function Ch_bfs$(v)$ returns the set of children of v in the current BFS (*i.e.* the neighbors u with the same BFS root such that the variable parent of u points to v and the distance of u is the distance of v plus one).

$$\text{Ch_bfs}(v) = \{u \mid u \in N_v \wedge (\mathsf{R}_{\mathsf{BFS}u}, \mathsf{P}_{\mathsf{BFS}u}, \mathsf{d}_{\mathsf{BFS}u}) = (\mathsf{R}_{\mathsf{BFS}v}, v, \mathsf{d}_{\mathsf{BFS}v} + 1)\} \tag{20}$$

The farther distance of process v is computed thanks to the function $\mathsf{MaxD}(v)$ that returns the largest value between its own current distance and the farther distance provided by its children in the current BFS.

$$\mathsf{MaxD}(v) = \max\{\mathsf{d}_{\mathsf{BFS}v}, \max\{\mathsf{D}_{\mathsf{BFS}u} \mid u \in \text{Ch_bfs}(v)\}\} \tag{21}$$

We present now predicates designed to check inconsistencies in eccentricity computation. To be considered coherent, a process v must satisfies the following set of properties: (i) all its children in the current `BFS` tree have already computed their farther distance, (ii) its variable $\mathsf{D}_{\mathsf{BFS}v}$ must be equal to the farther distance of the current root (computed by the predicate $\mathsf{MaxD}(v)$), (iii) either its parent has not yet computed its farther distance ($\mathsf{D}_{\mathsf{BFS}\mathsf{P}_{\mathsf{BFS}v}} =\downarrow$) or its father has already computed its own farther distance ($\mathsf{D}_{\mathsf{BFS}\mathsf{P}_{\mathsf{BFS}v}} \in \mathbb{N}$), and (iv) its variable $\mathsf{D}_{\mathsf{BFS}v}$ is coherent with $\mathsf{D}_{\mathsf{BFS}\mathsf{P}_{\mathsf{BFS}v}}$, (the one of its parent in the current BFS). The three first properties are gathered in the $\mathsf{CohD}_E(v)$ predicate while the fourth one is captured by the predicate $\mathsf{CohD}_{\bar{N}}(v)$.

$$\begin{aligned}\mathsf{CohD}_E(v) \equiv\ & (\mathsf{D}_{\mathsf{BFS}v} \in \mathbb{N}) \wedge (\mathsf{d}_{\mathsf{BFS}v} = \min\{\mathsf{d}_{\mathsf{BFS}u} : \forall u \in N_v\} + 1) \\ & \wedge (\forall u \in \text{Ch_bfs}(v), \mathsf{D}_{\mathsf{BFS}u} \in \mathbb{N}) \wedge (\mathsf{D}_{\mathsf{BFS}v} = \mathsf{MaxD}(v)) \wedge (\mathsf{D}_{\mathsf{BFS}\mathsf{P}_{\mathsf{BFS}v}} \in \{\downarrow, \mathbb{N}\})\end{aligned} \tag{22}$$

$$\mathsf{CohD}_{\bar{N}}(v) \equiv (\mathsf{D}_{\mathsf{BFS}\mathsf{P}_{\mathsf{BFS}v}}, \mathsf{D}_{\mathsf{BFS}v}) \in \{(\bot, \bot), (\bot, \uparrow), (\downarrow, \bot), (\uparrow, \uparrow), (\downarrow, \downarrow)\} \tag{23}$$

As a consequence, a process v is considered coherent (notion described by the predicate $\mathsf{CohD}(v)$) if it satisfies both $\mathsf{CohD}_{\bar{N}}(v)$ and $\mathsf{CohD}_E(v)$.

$$\mathsf{CohD}(v) \equiv (\mathsf{CohD}_{\bar{N}}(v) \wedge \mathsf{CohD}_E(v)) \tag{24}$$

A process is allowed to compute its eccentricity only when all its neighbors have computed their distance from the root in `BFS`. The predicate $\text{Gd}(v)$

formalizes this configuration.

$$\mathrm{Gd}(v) \equiv (\forall u \in N(v), \mathsf{d}_{\mathsf{BFS}_u} \in \{\mathsf{d}_{\mathsf{BFS}_v} - 1, \mathsf{d}_{\mathsf{BFS}_v}, \mathsf{d}_{\mathsf{BFS}_v} + 1\}) \tag{25}$$

All these predicates and functions allow us to write the $\mathrm{Dis}(v)$ function that is the heart of all waves used by the eccentricity computation. This function returns to the process v the new value of its $\mathsf{D}_{\mathsf{BFS}_v}$ variable depending on the current wave.

$$\mathrm{Dis}(v) = \begin{cases} \uparrow & \text{if } \neg\mathsf{CohD}(v) \vee \big((\mathsf{D}_{\mathsf{BFS}_v} \neq \bot) \wedge \mathsf{CohD}(v) \\ & \qquad \wedge(\exists u \in N_v \mid \mathsf{D}_{\mathsf{BFS}_u} = \uparrow)\big) \\ \bot & \text{if } \mathsf{CohD}(v) \wedge (\mathsf{D}_{\mathsf{BFS}_{\mathsf{P}_{\mathsf{BFS}_v}}}, \mathsf{D}_{\mathsf{BFS}_v}) = (\bot, \uparrow) \\ & \qquad \wedge(\forall u \in \mathrm{Ch_bfs}(v), \mathsf{D}_{\mathsf{BFS}_u} = \uparrow) \\ \downarrow & \text{if } \mathsf{CohD}(v) \wedge (\mathsf{D}_{\mathsf{BFS}_v} = \bot) \wedge (\forall u \in \mathrm{Ch_bfs}(v), \mathsf{D}_{\mathsf{BFS}_u} = \bot) \wedge \\ & \qquad (\mathsf{D}_{\mathsf{BFS}_{\mathsf{P}_{\mathsf{BFS}_v}}} = \downarrow) \\ \mathsf{MaxD}(v) & \text{if } \mathsf{CohD}(v) \wedge (\mathsf{D}_{\mathsf{BFS}_v} = \downarrow) \wedge (\forall u \in \mathrm{Ch_bfs}(v), \mathsf{D}_{\mathsf{BFS}_u} \in \mathbb{N}) \\ & \qquad \wedge \mathrm{Gd}(v) \end{cases} \tag{26}$$

The rule $\mathbb{R}_{\mathtt{Ecc}}$ is responsible for the propagation of the different waves but does not handle their initialization at the root r of the BFS. This is the aim of the rule $\mathbb{R}_{\mathtt{CleanEcc}}$. There are two types of waves to initialize: the $\bot$ wave and the $\downarrow$ wave. They are respectively started when all neighbors v of r satisfy $\mathsf{D}_{\mathsf{BFS}_v} = \uparrow$ (see predicate Down) and when all neighbors v of r satisfy $\mathsf{D}_{\mathsf{BFS}_v} = \bot$ (see predicate Bot).

$$\mathrm{Down}(v) \equiv (\mathsf{D}_{\mathsf{BFS}_v} = \downarrow) \wedge (\forall u \in N_v, \mathsf{D}_{\mathsf{BFS}_u} = \uparrow) \tag{27}$$

$$\mathrm{Bot}(v) \equiv (\mathsf{D}_{\mathsf{BFS}_v} = \bot) \wedge (\forall u \in N_v, \mathsf{D}_{\mathsf{BFS}_u} = \bot) \tag{28}$$

The execution of the rule $\mathbb{R}_{\mathtt{CleanEcc}}$ initiates a wave in both cases and the type of this wave is determined by the function $\mathrm{DCE}(v)$ above.

$$\mathrm{DCE}(v) = \begin{cases} \bot & \text{if } \mathrm{Down}(v) = \mathit{true} \\ \downarrow & \text{if } \mathrm{Bot}(v) = \mathit{true} \end{cases} \tag{29}$$

$$\mathbb{R}_{\texttt{CleanEcc}} \quad : \quad \text{RootBFS}(v) \wedge \text{SameBFS}(v) \wedge (\text{Down}(v) \vee \text{Bot}(v)) \\ \longrightarrow \mathsf{D}_{\mathsf{BFS}_v} := \text{DCE}(v) \tag{30}$$

Finally, the rule $\mathbb{R}_{\texttt{EndBFS}}$ is executed by the root of the BFS when the leaves-to-root propagation of the eccentricity is over. This termination is checked by the predicate $\text{ChN}(v)$. This predicate is satisfied once all the children of the root have computed their farther distance.

$$\text{ChN}(v) \equiv (\forall u \in N_v, \mathsf{D}_{\mathsf{BFS}_u} \in \mathbb{N}) \tag{31}$$

The execution of this rule computes the eccentricity of r (with the help of the function $\text{Dis}(v)$), releases the token by updating the variable next_r of the token circulation layer (see Section 3.3), and updates one variable for communicating the new eccentricity to the centers determination layer (see Section 3.5 for more details on the use of the variable $\mathsf{MinEcc}(v)$).

$$\mathbb{R}_{\texttt{EndBFS}} : \text{RootBFS}(v) \wedge \text{SameBFS}(v) \wedge \text{ChN}(v) \\ \longrightarrow \mathsf{Ecc}_v := \text{Dis}(v); \mathsf{next}_v := \text{Next}(v); \\ \mathsf{MinEUP}_v := \mathsf{MinEcc}(v); \tag{32}$$

3.5. Centers Computation

The fourth and last layer of our algorithm aims to identify the centers of the system. As each process computes its own eccentricity with the three first layers of our algorithm, it remains only to compute the minimal one. In this goal, we use the `Backbone` (oriented towards the leader elected by the first layer). First, the root gathers the minimal eccentricity in the system in a leaves-to-root wave. Then, the root floods the `Backbone` with it in a root-to-leaves wave containing this minimum eccentricity. This algorithm works under the distributed unfair daemon, uses $O(\log n)$ bits of memory per process, and stabilizes within $O(n)$ rounds.

This layer makes use of the following variables. The variable Ecc_v (maintained by the third layer, refer to Section 3.4) stores the eccentricity of the

process v. The variable MinEUP_v is used to collect the minimal eccentricity in the leave-to-root wave while the variable MinE_v is used to store the minimal eccentricity of the system and to broadcast it in the root-to-leaves wave. Moreover, we add a variable Center_v that represents the center of the graph, Center_v is $true$ if and only if $\mathsf{MinE}_v = (\mathsf{Ecc}_v, \mathrm{ID}_v)$, otherwise Center_v is $false$. We define the following function:

$$\mathsf{MinEcc}(v) = \min\{(\mathsf{Ecc}_v, \mathrm{ID}_v), \min\{\mathsf{MinEUP}_u \mid u \in \mathrm{child}(v)\}\}$$

note that $\mathsf{MinEcc}(v)$ returns the minimum eccentricity and among the nodes with the minimum eccentricity the one with minimum identity.

Algorithm 4 Computation of the minimum eccentricity for process v

$\mathbb{R}_{\texttt{MinEUp}}$: $\mathsf{MinEUP}_v \neq \mathsf{MinEcc}(v) \longrightarrow \mathsf{MinEUP}_v := \mathsf{MinEcc}(v);$

$\mathbb{R}_{\texttt{MinERoot}}$: $\mathrm{BRoot}(v) \wedge (\mathsf{MinE}_v \neq \mathsf{MinEUP}_v) \longrightarrow \mathsf{MinE}_v := \mathsf{MinEUP}_v;$
$\mathsf{Center}_v := \texttt{IsCenter}(v);$

$\mathbb{R}_{\texttt{MinEDown}}$: $\neg\mathrm{BRoot}(v) \wedge (\mathsf{MinE}_v \neq \mathsf{MinE}_{\mathsf{p}_v})) \longrightarrow \mathsf{MinE}_v := \mathsf{MinE}_{\mathsf{p}_v};$
$\mathsf{Center}_v := \texttt{IsCenter}(v);$

The function $\texttt{IsCenter}(v)$ returns $true$ if the node v is the center of the graph, $false$ otherwise. Remember that, among the centers node we choose the one with the minimum identity.

$$\texttt{IsCenter}(v) = \begin{cases} true & \text{if } \mathsf{MinE}_v = (\mathsf{Ecc}_v, \mathrm{ID}_v) \\ false & \text{otherwise} \end{cases} \tag{33}$$

Formal presentation of this algorithm is done in Algorithm 4. The rule $\mathbb{R}_{\texttt{MinEUp}}$ manages the leaves-to-root gathering of minimal eccentricity while rules $\mathbb{R}_{\texttt{MinERoot}}$ and $\mathbb{R}_{\texttt{MinEDown}}$ ensures its root-to-leaves propagation (respectively for the root of `Backbone` and for other processes).

Once this algorithm stabilizes, each process knows its eccentricity (thanks to the third layer) and the minimal eccentricity in the system (thanks to the fourth

layer). Then, it is trivial for a process to decide if it is a center or not. As we assume that processes have unique identifiers, it is easy to elect the center with the minimal identity in the case where the system admits more than one center in order to construct a single minimum diameter spanning tree (note that this phase requires $O(\log n)$ bits of memory per process and stabilizes within $O(n)$ rounds).

In conclusion, the composition of these four layers provides us a self-stabilizing algorithm for centers computation or minimum diameter spanning tree construction under the distributed unfair daemon that needs $O(\log n)$ bits of memory per process and stabilizes within $O(n)$ rounds.

4. Proof of the Algorithm

As our algorithm is composed of several layers, one would be tempted to use some existing generic tool to ease its proof. Unfortunately, none of the existing methods of self-stabilizing algorithms composition is suitable in our case. Indeed, we cannot use methods that restrict the power of the adversary since we consider a fully asynchronous environment (*e.g.* the fair composition of [21, 22] that assumes a central and/or fair daemon or the parallel composition of [25] that works only with a fair daemon). Some other methods are relevant only to specific composition that are not used by our algorithm (*e.g.* the cross-over composition of [4] —composition of a daemon transformer with another algorithm—, the dependency graph composition of [3] —every composed algorithm must converge to a fixed point—, or the adaptive composition of [28] —a variable is used to select only one of the composed algorithms—). Finally, the more relevant methods (*e.g.* the module composition of [44] or the convergence stairs composition of [29]) are not suitable because they require each layer of the algorithm to converge to a *closed* predicate, that is not the case of some layers of our algorithm (*e.g.* the token circulation layer modifies the root of the BFS tree each time this latter is computed by the above layer, hence violating the predicate to which the BFS tree layer converges).

In consequence, we have to come back to a custom proof for our algorithm

using classical approaches. In particular, we use the property that, for a self-stabilizing algorithm, the set of legitimate configurations (*i.e.* configurations satisfying the specification of the problem) is an *attractor* of Γ for this algorithm. Given two sets of configurations $\Gamma_2 \subseteq \Gamma_1 \subseteq \Gamma$, we say that Γ_2 is an attractor of Γ_1 for algorithm $\mathcal{A}$ (denoted by $\Gamma_1 \rhd \Gamma_2$) if any execution of $\mathcal{A}$ starting from any configuration of Γ_1 reaches in a finite time a configuration of Γ_2 and if Γ_2 is closed under $\mathcal{A}$.

Some of our proofs are performed by exhibiting *potential functions*. A potential function is a bounded function that associates to each configuration of the system value that strictly decreases at each application of a rule by the algorithm. Hence, if the minimal value of the function is associated with a legitimate configuration of the system, the existence of such a function is sufficient to prove the convergence of the algorithm. The difficulty is obviously to exhibit the potential function that captures precisely the behavior of the self-stabilizing algorithm. Regarding the BFS spanning tree construction algorithm we use here, [34] proposed a potential function but this one is not suitable in our case (because of a too weak characterization of algorithm's effects on configurations and of the use of a *a priori* knowledge of processes). It is why we propose a more involved potential function for our algorithm to prove the lemma 7.

Theorem 1 *The algorithm $\mathcal{SSCC}$ is a self-stabilizing algorithm that computes one of the center of the system under the distributed unfair daemon. It uses $O(\log n)$ bits of memory per process and stabilizes within $O(n^2)$ rounds.*

Definition 2 ***Blocking path**: Let v be the process with one token. Define a* blocking path *of v as a maximal length path in $\texttt{BFS}(v)$ from the root v of the $\texttt{BFS}(v)$ to another process element of $\texttt{BFS}(v)$ such that every process u on this path satisfies $\mathsf{D}_{\mathsf{BFS}_u} \notin \mathbb{N}$.*

Note that, the rules $\mathbb{R}_{\texttt{EndBFS}}$, $\mathbb{R}_{\texttt{CleanEcc}}$, $\mathbb{R}_{\texttt{BFS}}$ and $\mathbb{R}_{\texttt{Ecc}}$ need the predicate SameBFS to true. The predicate SameBFS(v) is true for v if v and all these neighbors in G share the same root for the BFS (variable $\mathsf{R}_{\mathsf{BFS}}$), only the rule $\mathbb{R}_{\texttt{Tree}}$ modify this variable. Moreover, when a process v executes the rule $\mathbb{R}_{\texttt{Tree}}$, it puts its

variable $\mathsf{D}_{\mathsf{BFS}_v}$ at $\perp$. As a consequence if v was in a blocking path it maintains the blocking path and if v was not in a blocking path but its parent is in a blocking path then v reaches the blocking path. Let γ be a configuration, we introduce a potential function denoted by λ as follow $\Gamma \times V \times V \to \mathbb{N}$ be the function defined by:

$$\lambda(\gamma, r, v) = \begin{cases} 1 & \text{if } v \text{ belongs to a blocking path of } r \text{ in } \gamma \\ 1 & \text{if } v \notin \text{SameBFS}(r) \text{ and } v \in N_r \text{ in } \gamma \\ 0 & \text{otherwise} \end{cases}$$

Now, we define our potential function Λ: $\Gamma \to \mathbb{N}$ as follows:

$$\Lambda(\gamma, r) = \sum_{v \in V} \lambda(\gamma, r, v)$$

Claim 1 $\Lambda(\gamma, r) > 0$ *implies the token is blocked in* r*.*

Proof of the claim. The token is released when the process r executes the rule $\mathbb{R}_{\texttt{EndBFS}}$(32), but this rule is activatable only if all the neighbors of r are in $\texttt{BFS}(r)$ (see predicate SameBFS(v)(17)) and all the children of r in the $\texttt{BFS}(r)$ have already compute the distance maximum in their subtree (see predicate ChN(v) (31) in rule $\mathbb{R}_{\texttt{EndBFS}}$). If it is not the case then the token is blocked and by definition of Λ we have $\lambda(\gamma, r, v) = 1$ so $\Lambda(\gamma, r) > 0$. ∎

Let denoted by $\texttt{T}$ a spanning tree shaped by the variable p_v (namely a sub-spanning tree or the $\texttt{Backbone}$). Let B a set of enabled process that the scheduler does not want to execute. The process r is the root of $\texttt{T}$, either a process with a blocked token or a process in B. Let us define $\Gamma_{\texttt{Tree}}$ as the set of configurations such that for all $\gamma \in \Gamma_{\texttt{Tree}}$ we have $\Lambda(\gamma, r) > 0$ or $r \in B$ and the rule $\mathbb{R}_{\texttt{Tree}}$(13) is enabled by no process.

Claim 2 *If* $\Lambda(\gamma, r) > 0$ *then the activation of* $\mathbb{R}_{\texttt{Tree}}$ *by any process* $v \in V$ *in configuration* γ *gives a configuration* γ' *such that* $\Lambda(\gamma', r) = \Lambda(\gamma, r)$*.*

Proof of the claim. Let us consider a configuration γ with $\Lambda(\gamma, r) > 0$ and A the set of process activated by the rule $\mathbb{R}_{\texttt{Tree}}$(13). More precisely, the rule

$\mathbb{R}_{\texttt{Tree}}$(13) is enabled when the predicate GoodBFS(v) (14) is not true, that means $\mathsf{R}_{\mathsf{BFS}_v} \neq \mathsf{R}_{\mathsf{BFS}_{\mathrm{p_next}_v}}$. We distinguish two cases:

1. Let us consider, v be a neighbor of r, if $\mathbb{R}_{\texttt{Tree}}$ is activatable by v that means $v \notin \mathrm{SameBFS}(r)$ so $\lambda(\gamma, r, v) = 1$, the execution of rule $\mathbb{R}_{\texttt{Tree}}$ by v in γ gives $\mathsf{D}_{\mathsf{BFS}_v} = \perp$ in γ', so that maintain $\lambda(\gamma', r, v) = 1$.

2. Let us consider now, v be not a neighbor of r, if $\mathbb{R}_{\texttt{Tree}}$ is activatable by v that means $\mathsf{R}_{\mathsf{BFS}_v} \neq \mathsf{R}_{\mathsf{BFS}_{\mathrm{p_next}_v}}$ so either $\mathsf{R}_{\mathsf{BFS}_{\mathrm{p_next}_v}} \neq r$ as a consequence $\lambda(\gamma, r, v) = \lambda(\gamma', r, v) = 0$, or $\mathsf{R}_{\mathsf{BFS}_{\mathrm{p_next}_v}} = r$ so $\lambda(\gamma, r, v) = 0$ and $\lambda(\gamma', r, v) = 0$ as the execution of rule $\mathbb{R}_{\texttt{Tree}}$ by v in γ gives $\mathsf{P}_{\mathsf{BFS}_v} = \perp$ (v is not in $\texttt{BFS}(v)$ in γ' since v has not parent) .

■

We denote by $p(r, v)$ the path in spanning tree $\texttt{T}$ from the root r of $\texttt{T}$ to v and by $d(v)$ the distance between r and v in $\texttt{T}$. Let ξ be the following potential function $\Gamma \times V \times V \to \mathbb{N}$ be the function defined by:

$$\xi(\gamma, r, v) = \sum_{u \in p(r,v) \wedge \mathsf{R}_{\mathsf{BFS}_u} \neq \mathsf{R}_{\mathsf{BFS}_{\mathrm{p_next}(u)}}} (n - d(u))$$

And Ξ the potential function: $\Gamma \to \mathbb{N}$ as follows:

$$\Xi(\gamma, r) = \sum_{v \in V} \xi(\gamma, r, v)$$

Note that, $\Xi(\gamma, r) = 0$ means $\mathbb{R}_{\texttt{Tree}}$ is enabled by no process and $\forall v \in V$ we have $\mathsf{R}_{\mathsf{BFS}_v} = r$.

Claim 3 *Starting from a configuration $\gamma \in \Gamma$ with $\Lambda(\gamma, r) > 0$ or $r \in B$ the number of activation of $\mathbb{R}_{\texttt{Tree}}$ is bounded.*

Proof of the claim. The rule $\mathbb{R}_{\texttt{Tree}}$ is based on the predicate GoodBFS (14), if its predicate it false ($\mathsf{R}_{\mathsf{BFS}_v} \neq \mathsf{R}_{\mathsf{BFS}_{\mathrm{p_next}_v}}$) then the rule $\mathbb{R}_{\texttt{Tree}}$ is enabled for the process v. The variable $\mathrm{p_next}_v$ is modified by the circulation of the Token. So in a spanning tree without token or with blocked the token this variable

never changes. Let us consider a process v such that $\xi(\gamma, r, v) > 0$, and the processes u such that $u \in p(r, v)$ and u is enabled by rule $\mathbb{R}_{\texttt{Tree}}$ and activated by the scheduler in γ, Moreover u have not ancestor in $p(r, v)$ activates by the scheduler. The process u is involved in $\xi(\gamma, r, v)$ with a value $(n - d(u))$. Let γ' be the configuration given by the execution of rule $\mathbb{R}_{\texttt{Tree}}$ by u, we obtain in γ' $\mathsf{R}_{\mathsf{BFS}u} = \mathsf{R}_{\mathsf{BFS}\text{p_next}_u}$. Moreover in γ', if the child w of u in the path $p(r, v)$ is not enabled by rule $\mathbb{R}_{\texttt{Tree}}$ then $\xi(\gamma, r, v)$ is decreased by at $(n - d(u))$. If w is enabled by rule $\mathbb{R}_{\texttt{Tree}}$, either it was enabled by rule $\mathbb{R}_{\texttt{Tree}}$ in γ and $\xi(\gamma, r, v)$ is decreased by at $(n - d(u))$ or it was not enabled by rule $\mathbb{R}_{\texttt{Tree}}$ like $d(u) = d(w) + 1$ then $\xi(\gamma, r, v)$ is decreased by at one. As direct consequence we obtain $\xi(\gamma, r, v) > \xi(\gamma', r, v)$ and until $\Lambda(\gamma, r) > 0$ we have $\Xi(\gamma, r) > \Xi(\gamma', r)$. ∎

Let us define $\Gamma_{\texttt{Backbone}} \subseteq \Gamma$ as the set of configurations of Γ such that no rule of the first layer of $\mathcal{SSCC}$ (leader election and `Backbone` construction) is enabled.

Lemma 1 *$\Gamma \rhd \Gamma_{\texttt{Backbone}}$ in $O(n)$ rounds and $\Gamma_{\texttt{Backbone}}$ is closed.*

Proof. Let be $\gamma_0 \notin \Gamma_{\texttt{Backbone}}$ the initial configuration of the system. Let us denote by B the enabled processes by rules of the construction of the `Backbone`. As the scheduler want to block the construction of the `Backbone`, it does not activate processes in B, as a consequence until the scheduler executes an enabled process of `Backbone` no round is completed. In γ_0 two types of spanning structures can occur, namely, sub spanning trees and cycles. In a cycle, at least one process is an element of B. So if we consider the graph G_0 the subgraph induced by the processes $V \setminus B$ then only one type of structure occurs, sub spanning trees. Let us denote by $\Gamma_{\texttt{GToken}} \subseteq \Gamma$ the set of configurations in which no process is enabled by rule $\mathbb{R}_{\texttt{ErToken}}$. Then, we have:

Claim 4 $\Gamma \rhd \Gamma_{\texttt{GToken}}$ in $O(1)$ rounds and $\Gamma_{\texttt{GToken}}$ is closed.

Proof of the claim. The algorithm of [41] assures the convergence and the closure for $\Gamma_{\texttt{GToken}}$ for each sub-trees. ∎

The authors of [41] assure the uniqueness of the token in a tree. Remark that in the absence of token, it is the root of the tree that assures the creation of a token. In our case, if the root of the subtree is a process in B then no token is created, so in $\gamma \in \Gamma_{\texttt{GToken}}$ and G_0 there exist sub spanning trees with only one token or no token. Let us consider a γ a configuration in $\Gamma_{\texttt{GToken}}$ and the graph G_0. We do not modify the backward token circulation (rule $\mathbb{R}_{\texttt{Backward}}$). Let $\texttt{T}$ be a sub spanning tree of G_0, if $\texttt{T}$ has a token then the number of activation of $\mathbb{R}_{\texttt{Backward}}$ is limited by the depth of $\texttt{T}$. Note that, if the root r of $\texttt{T}$ is an element of B the circulation of the token is blocked by r. So w.l.g consider a spanning tree $\texttt{T}$ with a root $r \notin B$, the tree $\texttt{T}$ has only one token thanks to the algorithm [41]. We proof that starting from γ the circulation of the token in $\texttt{T}$ reaches a configuration γ' where the circulation of the token is blocked.

The forward circulation is closely linked to the rules $\mathbb{R}_{\texttt{StartBFS}}$ and $\mathbb{R}_{\texttt{EndBFS}}$. When a process receives the token in forwarding circulation, it executes the rule $\mathbb{R}_{\texttt{StartBFS}}$, it releases the token thanks to the rule $\mathbb{R}_{\texttt{EndBFS}}$.

Claim 5 *$\Gamma_{\texttt{GToken}} \rhd \Gamma_{\texttt{Tree}}$ in $O(1)$ rounds and $\Gamma_{\texttt{Tree}}$ is closed.*

Proof of the claim. By claim 3 we obtain that the number of activation of rule $\mathbb{R}_{\texttt{Tree}}$ is bounded in a spanning tree $\texttt{T}$ if the token is blocked in the root r or $r \in B$. Let now consider a sub spanning tree $\texttt{T}$ where the token is not blocked, like $\gamma \not\subset \Gamma_{\texttt{Backbone}}$, $\texttt{T}$ has at least one process v with a neighbor u such that $u \notin \texttt{T}$, let denote by $\texttt{T}'$ the spanning tree such that $u \in \texttt{T}'$. Remark that, like $\gamma \not\subset \Gamma_{\texttt{Backbone}}$ either $\texttt{T}$ or $\texttt{T}'$ have one process in B, w.l.g suppose it is $\texttt{T}'$ so the circulation of the token in $\texttt{T}'$ if it exists is blocked. Let denoted by r' the root of $\texttt{T}'$, by claim 3 we obtain that the system reaches a configuration γ' where $\mathsf{R}_{\mathsf{BFS}_u} = \mathsf{R}_{\mathsf{BFS}_{r'}}$ and this value never change. If the token was in a process w in $\texttt{T}$, w can release the token in process w'. All the processes in path between w' and v in $\texttt{T}$ execute the rule $\mathbb{R}_{\texttt{Tree}}$, and all put their variable $\mathsf{d}_{\mathsf{BFS}}$ to $\perp$, that result to blocking path between w' and v. The process u cannot have $\mathsf{R}_{\mathsf{BFS}_u}$ equal to w' because u is not $\texttt{T}$, so v will never have $\text{SameBFS}(v)$ true as a consequence the rule $\mathbb{R}_{\texttt{Tree}}$(15), $\mathbb{R}_{\texttt{Ecc}}$(19) and $\mathbb{R}_{\texttt{CleanEcc}}$ (30) are note enabled for any node in $\texttt{T}$, moreover the token is blocked. To conclude, the system reaches a configuration

γ' such that $\Lambda(\gamma', w) > 0$ and only the processes in B are enabled, that is the rule involved in the construction of the `Backbone`. The convergence is given in [18]. ■

The closure is given by [18]. □

Let us denote by $\Gamma_{\texttt{1-Token}} \subseteq \Gamma$ the set of configurations in which there exists exactly one token (*i.e.* no process is enabled by rule $\mathbb{R}_{\texttt{ErToken}}$).

Lemma 2 $\Gamma_{\texttt{Backbone}} \rhd \Gamma_{\texttt{1-Token}}$ *in* $O(n)$ *rounds and* $\Gamma_{\texttt{1-Token}}$ *is closed.*

Proof. Let us consider $\gamma \in \Gamma_{\texttt{Backbone}}$ and B denote the enabled processes by rule $\mathbb{R}_{\texttt{ErToken}}$. As the scheduler want to block the convergence toward one token, it does not activate processes in B. The proof mimics the proof of claim 5, the broadcast of the variable $\mathsf{R}_{\mathsf{BFS}}$ is blocked by the process in B so the circulation of the token if exists, will be blocked. As a consequence, the system reaches a configuration γ' such that $\Lambda(\gamma', w) > 0$ and only the processes in B are enabled. We do not modify the algorithm of [41], the authors proof the closure of the algorithm in their article. □

Let us define $\Gamma_{\texttt{e}}$ as the set of configuration $\gamma \in \Gamma_{\texttt{BFS}}$ where the every process $v \in V \setminus \{r\}$ has $\mathsf{D}_{\mathsf{BFS}_v} = \max\{\mathsf{d}_{\mathsf{BFS}_v}, \max\{\mathsf{D}_{\mathsf{BFS}_u} \mid u \in \text{Ch_bfs}(v)\}\}$.

Lemma 3 *If* $\Lambda(\gamma, r) > 0$ *and* $\gamma \in \Gamma_{\texttt{1-Token}}$ *then we obtain* $\Gamma_{\texttt{1-Token}} \rhd \Gamma_{\texttt{e}}$ *in* $O(n)$ *rounds.*

Proof.

Corollary 1 *By claims 3 and 2 we have, starting from a configuration* $\gamma \in \Gamma_{\texttt{1-Token}}$ *with* $\Lambda(\gamma, r) > 0$ *the system reaches a configuration* $\Gamma_{\texttt{Tree}}$ *with* $\Lambda(\gamma', r) > 0$ *and* $\forall v \in V$ *we have* $\mathsf{R}_{\mathsf{BFS}_v} = r$*.*

Claim 6 *If* $\Lambda(\gamma, r) > 0$ *then the activation of* $\mathbb{R}_{\texttt{BFS}}$ *by any process* $v \in V$ *in configuration* γ *gives a configuration* γ' *such that* $\Lambda(\gamma', r) \geq \Lambda(\gamma, r)$*.*

Proof of the claim. The activation of rule $\mathbb{R}_{\texttt{BFS}}$(15) by process v assigns $\mathsf{D}_{\mathsf{BFS}_v} \in \{\downarrow, \uparrow\}$, if v was in a blocking path that keeps v in a blocking path, otherwise if v

was a neighbor of a process u element of a blocking path the process v reaches the blocking path and $\Lambda(\gamma', r)$ increases by at least one compare to $\Lambda(\gamma, r)$.

■

Let us define $\Gamma_{\mathtt{BFS}}$ a configuration γ where $\forall v \in V$, $\mathsf{d_{BFS}}_v = d_G(r, v)$ and $\Lambda(\gamma) > 0$. In other words, the rule $\mathbb{R}_{\mathtt{BFS}}$ is enabled by no process ($\forall v \in V \setminus \{r\}$ we have $\mathsf{d_{BFS}}_v = \min\{\mathsf{d_{BFS}}_u : u \in N(v)\} + 1$).

Claim 7 Starting from a configuration $\gamma \in \Gamma_{\mathtt{Tree}}$ with $\Lambda(\gamma, r) > 0$ the number of activation of $\mathbb{R}_{\mathtt{BFS}}$ is bounded.

The computation of BFS tree with only one rule is well known, unfortunately to our knowledge all the proofs do not use a potential function, to be consistent we propose a new proof based on potential function. The authors in [34] proposed a potential function but this one is not suitable in our case (because of a too weak characterization of algorithm's effects on configurations and of the use of a *a priori* knowledge of processes). Moreover, the algorithm in [34] use 3 rules and the proof is based on the parent of each process, we generalize this proof for our algorithm that uses only one rule for the BFS tree.
Proof of the claim. Let γ a configuration in $\Gamma_{\mathtt{Tree}}$ and $A(\gamma)$ be the set of all activated processes by rule $\mathbb{R}_{\mathtt{BFS}}$ in γ. After the activations of processes of $A(\gamma)$ the system reaches the configuration γ'. Notice that, only the rule $\mathbb{R}_{\mathtt{BFS}}$ can change the distance on the BFS tree (e.i variable $\mathsf{d_{BFS}}$), to clarify our purpose we denote by $\mathsf{d_{BFS}}_v(\gamma)$ the distance on the BFS tree of process v in configuration γ. To compute a BFS, a process v waits that all its neighbors are in a same tree of it (see SameBFS 17). Moreover, the rule $\mathbb{R}_{\mathtt{BFS}}$ is activatable only when the distance of process v can be improved (see Best 18), for convenance we introduce $\mathsf{min}^{+1}(v, \gamma) = \min\{\mathsf{d_{BFS}}_u(\gamma) \mid u \in N_v\} + 1$ that is the result of predicate Best(v). Let $P(\gamma)$ be the following potential function:

$$P(\gamma) = (F(\gamma), S(\gamma))$$

where

$$S(\gamma) = \sum_{v \in V} \mathsf{d}_{\mathsf{BFS}_v}(\gamma)$$

and

$$F(\gamma) = (f_1(\gamma), f_2(\gamma), \ldots, f_{2n+\max\{\mathsf{d}_{\mathsf{BFS}_v}(\gamma_0)|v \in V\}}(\gamma))$$

where $\mathtt{f}_k(\gamma) = \{v|\mathsf{d}_{\mathsf{BFS}_v}(\gamma) = k \wedge \mathsf{d}_{\mathsf{BFS}_v}(\gamma) \leq \mathsf{min}^{+1}(v,\gamma)\}$ and $f_k(\gamma) = |\mathtt{f}_\mathtt{k}(\gamma)|$. Let us denote by K the set $\{1, \ldots, 2n + \max\{\mathsf{d}_{\mathsf{BFS}_v}(\gamma_0)|v \in V\}\}$. Note that, by definition all the processes v in $\mathtt{f}_\mathtt{k}(\gamma)$ for all $k \in K$ are enabled by rule $\mathbb{R}_{\mathtt{BFS}}$. Let denote by m the distance such that $m = \min\{k \mid k \in K \wedge \exists v \in A(\gamma) \text{ s.t } v \in \mathtt{f}_k(\gamma)\}$. The comparaison between $F(\gamma)$ and $F(\gamma')$ is by lexical order. We denote by γ' the configuration after activation of the processes in $A(\gamma)$. We can now prove the following result $P(\gamma') < P(\gamma)$ for all configuration γ and $A(\gamma)$ not empty.

1. $\exists v \in V$ such that $v \notin \mathtt{f}_k(\gamma), \forall k \in K$, the only way for a process v to become element of $\mathtt{f}_s(\gamma')$ is to have a neighbor $u = \mathsf{min}^{+1}(v,\gamma')$ such that $u \in \mathtt{f}_t(\gamma)$ and $u \in A(\gamma)$. Let denote by w the neighbor of u such that $w = \mathsf{min}^{+1}(u,\gamma)$, w.l.g let us consider that $t = m$, by definition of processes u, v, w we have $\mathsf{d}_{\mathsf{BFS}_v}(\gamma) > m \leq \mathsf{d}_{\mathsf{BFS}_w}(\gamma)$. After activation of u we obtain, $\mathsf{d}_{\mathsf{BFS}_v}(\gamma') \leq \mathsf{d}_{\mathsf{BFS}_u}(\gamma') = \mathsf{dfs}_w(\gamma) + 1$. If $v \notin A(\gamma)$ we have $\mathsf{d}_{\mathsf{BFS}_v}(\gamma') = \mathsf{d}_{\mathsf{BFS}_v}(\gamma)$, otherwise if $v \in A(\gamma)$ after activation of v we achieve that $\mathsf{d}_{\mathsf{BFS}_v}(\gamma') = m{+}1$ so $\mathsf{d}_{\mathsf{BFS}_v}(\gamma') \geq m{+}1$ as a consequence f_m is decreased by at least one in γ' and the system reaches $F(\gamma') < F(\gamma)$.

2. $\exists v \in A(\gamma)$ such that $v \in \mathtt{f}_k(\gamma)$, so $\mathsf{d}_{\mathsf{BFS}_v}(\gamma) \leq \mathsf{d}_{\mathsf{BFS}_u}(\gamma)$ with $u = \mathsf{min}^{+1}(v,\gamma)$ and after activation of v we have $\mathsf{d}_{\mathsf{BFS}_v}(\gamma') = \mathsf{d}_{\mathsf{BFS}_u}(\gamma'){+}1$. W.l.g let suppose $k = m$ and let w be the process such that $w = \mathsf{min}^{+1}(v,\gamma')$.

   (a) Let us prove that if $u = w$ we obtain $F(\gamma') < F(\gamma')$:

      i. If $u \notin A(\gamma)$, we obtain $\mathsf{d}_{\mathsf{BFS}_v}(\gamma') > \mathsf{d}_{\mathsf{BFS}_u}(\gamma')$, so $v \notin f_s(\gamma') \forall s \in K$, as a consequence f_m is decreased by at least one.

      ii. $u \in A(\gamma)$ s.t. $u \in \mathtt{f}_\mathtt{s}(\gamma)$: $m \leq s \leq \mathsf{d}_{\mathsf{BFS}_z}(\gamma)$ with $z = \mathsf{min}^{+1}(u,\gamma)$, after activation of v and u we obtain $\mathsf{d}_{\mathsf{BFS}_v}(\gamma') = s{+}1 \leq \mathsf{d}_{\mathsf{BFS}_u}(\gamma') =$

$\mathsf{d_{BFS}}_z(\gamma) + 1$ so $v \in \mathtt{f_{s+1}}(\gamma')$ and f_m is decreased by at least one.

iii. $u \in A(\gamma)$ and $u \notin \mathtt{f}_s(\gamma) \forall s \in K$ so $\mathsf{d_{BFS}}_u(\gamma) > \mathsf{d_{BFS}}_z(\gamma)$ with $z = \mathsf{min}^{+1}(u, \gamma)$. After activation of v and u we reach $\mathsf{d_{BFS}}_v(\gamma') = \mathsf{d_{BFS}}_u(\gamma) + 1 > \mathsf{d_{BFS}}_u(\gamma') = \mathsf{d_{BFS}}_z(\gamma) + 1$ as a consequence $v \notin \mathtt{f}_s(\gamma') \forall s \in K$ and f_m is decreased by at least one.

(b) Let us prove now that if $u \neq w$ we obtain $F(\gamma') < F(\gamma')$:

i. $u \notin A(\gamma)$ and $w \notin A(\gamma)$ it is impossible because $u \neq w$.

ii. $u \in A(\gamma)$ and $w \notin A(\gamma)$ or $u \in A(\gamma)$ and $w \in A(\gamma)$: by definition of u and w we have $\mathsf{d_{BFS}}_u(\gamma) = m \leq \mathsf{d_{BFS}}_w(\gamma)$ and we obtain in after activation of the processes that $\mathsf{d_{BFS}}_v(\gamma') = m + 1 \leq \mathsf{d_{BFS}}_u(\gamma') = \mathsf{d_{BFS}}_w(\gamma') + 1$ so f_m is decreased by at least one.

To conclude this part if $\exists v \in A(\gamma)$ such that $v \in \mathtt{f}_k(\gamma)$ the function F decreases and as a consequence $P(\gamma') < P(\gamma)$.

3. $\forall v \in A(\gamma)$: $v \notin \mathtt{f}_k(\gamma), \forall k \in K$. A direct consequence of the first item of this proof is the activation of the processes maintains $F(\gamma') = F(\gamma)$, moreover this activation decreases the distance of v so $S(\gamma') < S(\gamma)$ and $P(\gamma') < P(\gamma)$.

■

Let $E(\gamma)$ be the set of enabled processes in γ and $\mathsf{notCoh}(\gamma) \subseteq \mathsf{E}(\gamma)$ a set of processes such that if v in $\mathsf{notCoh}(\gamma)$ implies $\mathsf{CohD}(v) = false$ (see 24, 22 and 23) and $\mathsf{D_{BFS}}(v) \neq \perp$ (otherwise v is not enabled).

Claim 8 Starting from a configuration $\gamma \in \Gamma_{\mathtt{BFS}}$ with $\Lambda(\gamma, r) > 0$ and if the scheduler never actives the elements of notCoh *and never decreases Λ then the system converges to a configuration γ' such that* $\mathsf{notCoh}(\gamma') = \mathsf{E}(\gamma')$.

Proof of the claim. Note that, this part of algorithm runs by up and down waves, if the scheduler decide does not activate a process v ($\mathsf{CohD}(v) = false$) the waves are blocked. More precisely, in a down waves ($\perp$ and $\downarrow$), the descendants u of v in the BFS with $\mathsf{CohD}(u) = true$ are not enabled until v are executed

(see reference to $\mathsf{P}_{\mathsf{BFS}}$ in lines 2 and 3 of predicate 26). In an up wave of the computation of the eccentricity, for a ancestors u of v with $\mathsf{CohD}(u) = true$ then the ancestor of v are not enabled until v is executed (see reference to Ch_bfs in line 4 of predicate 26). The up wave of $\uparrow$ a little bit different because this wave does not use all the bfs children but the existence of one neighbor of the process (see line 1 of predicate 26), but line 2 of predicate 26 chess if all the children u of v in the $\mathtt{BFS}$ have $\mathsf{D}_{\mathsf{BFS}_u} = \downarrow$ so the down wave of $\bot$ is blocked until the up wave of $\uparrow$ is complete. ■

Let us define $\Gamma_{\mathtt{GoodE}}$ the set of configurations $\gamma \in \Gamma_{\mathtt{BFS}}$ where $\forall v \in V$, $\mathsf{CohD}(v) = true$.

Claim 9 Starting from a configuration $\gamma \in \Gamma_{\mathtt{BFS}}$ *with* $\Lambda(\gamma, r) > 0$ *and* $\mathsf{notCoh}(\gamma) = \mathsf{E}(\gamma)$ *then the system reaches a configuration* $\gamma' \in \Gamma_{\mathtt{GoodE}}$.

Proof of the claim. Our algorithm use down and up waves, if the scheduler want slow down the convergence, it actives the processes v with $\mathsf{CohD}(v) = false$ one by one and this from the leaves of $\mathtt{BFS}(r)$ to the root of $\mathtt{BFS}(r)$. We already saw in proof of claim 8, that v and all the descendants of v in $\mathtt{BFS}(r)$ have their variable $\mathsf{D}_{\mathsf{BFS}}$ different of $\bot$. So when the scheduler actives v, v puts $\mathsf{D}_{\mathsf{BFS}_v} = \bot$ therefore all the children w of v in $\mathtt{BFS}(r)$ have now, $\mathsf{CohD}(v) = false$. A direct consequence of this is now all w is elements of $\mathsf{notCoh}(\gamma)$ and enabled, so the system reaches a configuration γ' with $\mathsf{notCoh}(\gamma') = \mathsf{E}(\gamma')$, remark that v is not enabled until its children takes the value $\uparrow$. Moreover, $\mathsf{CohD}(v)$ is becomes true. So with the same argument, this process is repeated until the leaves of the subtree of v in the $\mathtt{BFS}(r)$. When all the descendants u of v have $\mathsf{D}_{\mathsf{BFS}_u} = \bot$, a down wave of value $\uparrow$ is started until this wave reaches a process w in notCoh. We repeat the same proof until the extension of notCoh. Notice that, during all the process, the scheduler can maintain Λ positive. ■

To establish the claim, let $\phi : \Gamma_{\mathtt{GoodE}} \rightarrow \mathbb{N}$ be the potential function defined

as:

$$\phi(\gamma, v, r) \quad : \quad \begin{cases} n^2 & \text{if } \mathsf{D}_{\mathsf{BFS}_v} = \uparrow \\ n & \text{if } \mathsf{D}_{\mathsf{BFS}_v} = \bot \\ 1 & \text{if } \mathsf{D}_{\mathsf{BFS}_v} = \downarrow \\ 0 & \text{if } \mathsf{D}_{\mathsf{BFS}_v} \in \mathbb{N} \end{cases}$$

and we define the potential function Φ: $\Gamma_{\texttt{GoodE}} \to \mathbb{N}$ as follows:

$$\Phi(\gamma, r) = \sum_{v \in V \setminus \{r\}} \mu(\gamma, v, r)$$

Claim 10 *Let* $\gamma \in \Gamma_{\texttt{GoodE}}$ *be a configuration where* $\Lambda(\gamma, r) > 0$ *then the activation of* $\mathbb{R}_{\texttt{Ecc}}$ *by any process* $v \in V$ *in configuration* γ *gives a configuration* γ' *such that* $\Phi(\gamma, r) > \Phi(\gamma', r)$.

The proof is direct by definition of rule $\mathbb{R}_{\texttt{Ecc}}$ and definition of Φ.

Claim 11 *Let* $\gamma \in \Gamma_{\texttt{GoodE}}$ *be a configuration where* $\Lambda(\gamma, r) > 0$ $\Phi(\gamma, r) < n$ *then the activation of* $\mathbb{R}_{\texttt{Ecc}}$ *by any process* $v \in V$ *in configuration* γ *gives a configuration* γ' *such that* $\Lambda(\gamma, r) > \Lambda(\gamma', r)$.

Proof of the claim. In a configuration $\gamma \in \Gamma_{\texttt{GoodE}}$ if $\Phi(\gamma, r) < n$ then all process v have either $\mathsf{D}_{\mathsf{BFS}_v} = \downarrow$ or $\mathsf{D}_{\mathsf{BFS}_v} \in \mathbb{N}$, moreover, if $\Lambda(\gamma, r) > 0$ that means there exist a blocking path only composed by value $\downarrow$, and only the leaves of this blocking path are enabled, so at each activation of process v with $\mathsf{D}_{\mathsf{BFS}_v} = \downarrow$ the function Λ decreases by one. ■

Corollary 2 $\Gamma_{\texttt{BFS}} \rhd \Gamma_{\texttt{e}}$ *in* $O(n)$ *rounds.*

□

Let us denote by $e^*(v)$ the eccentricity of process v and by E^* the minimal eccentricity of the network G. We define $\Gamma_{\texttt{Center}}$ the set of configurations such that, for all v in V we have $\mathsf{Ecc}_v = e^*(v)$ and $\mathsf{MinE}_v = E^*$. The centers of the network are the processes with $\mathsf{Ecc}_v = \mathsf{MinE}_v$.

Lemma 4 $\Gamma_{\texttt{1-Token}} \rhd \Gamma_{\texttt{Center}}$ *in* $O(n^2)$ *rounds, and* $\Gamma_{\texttt{Center}}$ *is closed.*

Proof.

Claim 12 *In configurations* $\Gamma_{\texttt{e}}$ *the token is not blocked infinitely.*

Proof of the claim. Let us suppose that, the scheduler can block the token infinitely in process r, in other words starting for a configuration γ_0 with $\Lambda(\gamma_0, v) > 0$ the scheduler maintains for all configurations $\gamma > \gamma_0$ that $\Lambda(\gamma, v) > 0$. We saw in the claim 2 the use of rule $\mathbb{R}_{\texttt{Tree}}$ cannot decrease Λ, but the scheduler cannot boost indefinitely the rule $\mathbb{R}_{\texttt{Tree}}$ (see claim 3). Similarly, we saw in the claim 6 the use of rule $\mathbb{R}_{\texttt{BFS}}$ cannot decrease Λ but the scheduler cannot boost indefinitely the rule $\mathbb{R}_{\texttt{Tree}}$ (see claim 7). The last rule available if the token is block, it is the rule $\mathbb{R}_{\texttt{Ecc}}$. The proof of claim 8 proof that the scheduler cannot maintain infinitely processes with an error in the $\mathsf{D}_{\mathsf{BFS}}$ variable (see predicate CohD). As a consequence, the system reaches configurations without error of variable $\mathsf{D}_{\mathsf{BFS}}$. To conclude, the claims 9, 10 and claim 11 proof that the application of rule $\mathbb{R}_{\texttt{Ecc}}$ decrease the function Λ as a consequence the process r becomes the only process enabled, and the activation of rule $\mathbb{R}_{\texttt{EndBFS}}$(32) by r releases the token. Note that, in configuration $\gamma \in \Gamma_{\texttt{e}}$ we have for all $v \in V \setminus \{r\}$ has $\mathsf{D}_{\mathsf{BFS}_v} = \max\{\mathsf{d}_{\mathsf{BFS}_v}, \max\{\mathsf{D}_{\mathsf{BFS}_u} \mid u \in \text{Ch_bfs}(v)\}\}$, so when r executes $\mathbb{R}_{\texttt{EndBFS}}$, it puts in $\mathsf{R}_{\mathsf{BFS}_r}$ the value $e^*(r)$.

■

Let us define $\Gamma_{\texttt{TokDown}} \subseteq \Gamma_{\texttt{1-Token}}$ the set of configurations where the token circulate downward and $\Gamma_{\texttt{TokUp}} \subseteq \Gamma_{\texttt{1-Token}}$ the set of configurations where the token circulate upward. A direct consequence of claim 12 is the following:

Claim 13 $\Gamma_{\texttt{e}} \rhd \Gamma_{\texttt{TokDown}} \cup \Gamma_{\texttt{TokUp}}$ *in one round.*

Proof of the claim. In a configuration $\gamma \in \Gamma_{\texttt{e}}$ only the process r is enabled by rule $\mathbb{R}_{\texttt{EndBFS}}$. After activation of process r, $\mathsf{Ecc}_r = e^*(r)$ where $e^*(r)$ is the eccentricity of process r, and the token is released. ■

By lemma 3 we obtain the following corollary:

Corollary 3 $\Gamma_{\texttt{1-Token}} \rhd \Gamma_{\texttt{TokDown}} \cup \Gamma_{\texttt{TokUp}}$ *in* $O(n)$ *rounds, and* $\Gamma_{\texttt{TokDown}} \cup \Gamma_{\texttt{TokUp}}$ *is closed under* $\mathcal{SSCC}$*.*

Claim 14 $\Gamma_{\texttt{TokDown}} \cup \Gamma_{\texttt{TokUp}} \rhd \Gamma_{\texttt{Ecc}}$ *in* $O(n^2)$ *rounds, and* $\Gamma_{\texttt{Ecc}}$ *is closed under* $\mathcal{SSCC}$*.*

Proof of the claim. Starting from a configuration $\gamma \in \Gamma_{\texttt{TokDown}} \cup \Gamma_{\texttt{TokUp}}$ the system reaches a configuration $\gamma \in \Gamma_{\texttt{e}}$ in $O(n)$ rounds, in this configuration r is the only process enabled by rule $\mathbb{R}_{\texttt{EndBFS}}$, this rule puts $\mathsf{Ecc}_r = e^*(r)$. Moreover, r update the minimum eccentricity of the network (see variable $\mathsf{NewMinE}_v$), and releases the token. Thanks to claim 12, $\Gamma_{\texttt{TokDown}} \cup \Gamma_{\texttt{TokUp}}$ is closed all process v receiving the token in down circulation reaches $\mathsf{Ecc}_v = e^*(v)$. The token circulation takes $O(n)$ rounds, the computation $e^*(v)$ takes $O(n)$ for each process, therefore in $O(n^2)$ rondes the system reaches $\Gamma_{\texttt{Ecc}}$ and is $\Gamma_{\texttt{Ecc}}$ closed under $\mathcal{SSCC}$. ■

Let us denote by E^* the minimal eccentricity of the network and. Let us define $\Gamma_{\texttt{Center}} \subseteq \Gamma_{\texttt{Ecc}}$ as the set of configurations $\gamma \in \Gamma_{\texttt{Ecc}}$ such that, for all v in V and $\mathsf{MinE}_v = E^*$. The center of the network is the process with $\mathsf{MinE}_v = (\mathsf{Ecc}_v, \text{ID}_v)$.

Claim 15 $\Gamma_{\texttt{Ecc}} \rhd \Gamma_{\texttt{Center}}$ *in* $O(n^2)$ *rounds, and* $\Gamma_{\texttt{Center}}$ *is closed under* $\mathcal{SSCC}$*.*

Proof of the claim. The fourth layer of $\mathcal{SSCC}$ is dedicated to the gathering of the minimum of eccentricity of the network. This gathering is made in the same time of the token circulation and the eccentricity computation. When a process v change its eccentricity (variable $\mathsf{Ecc}(v)$) a down wave is triggered, from v to the root of the `Backbone`, for this the algorithm use the variable MinEUP. When the root of the `Backbone` receives a new minimum eccentricity, it broadcast the new minimum (see variable MinE_v). When a process v receives a new minimum eccentricity if $\mathsf{MinE}_v = (\mathsf{Ecc}_v, \text{ID}_v)$ then v knows that it is the center; otherwise v is not a center. ■ □

Lemma 5 *The algorithm* $\mathcal{SSCC}$ *uses* $O(\log n)$ *bits of memory per process.*

Proof. The construction of `Backbone` and the token circulations use $O(\log)$ bits of memory per process (see articles [18] and [41]). The computation of the eccentricity uses pointeur variables that use $O(\log n)$ bits of memory by process (see variables $\mathsf{R}_{\mathsf{BFS}}$ and $\mathsf{P}_{\mathsf{BFS}}$) and distance variables that use also $O(\log n)$ bits of memory by process (see variables Ecc, $\mathsf{d}_{\mathsf{BFS}}$ and $\mathsf{D}_{\mathsf{BFS}}$). So the algorithm $\mathcal{SSCC}$ uses $O(\log n)$ bits of memory per process.

□

This complete the proof of theorem 1

5. Conclusion

In this paper, we present the first self-stabilizing algorithm for the minimum diameter spanning tree construction that tolerates any asynchronous environment (captured by a distributed unfair daemon) and uses $O(\log n)$ bits of memory per process. Our algorithm achieves a stabilization time in $O(n^2)$ rounds. This contribution improves the existing results by a factor n regarding the memory requirement.

This work opens some challenging questions that follow. These questions are focused on the optimality of memory requirement. The answer depends whether we want to obtain a *silent* self-stabilizing algorithm or not. A silent self-stabilizing algorithm is a self-stabilizing algorithm such that processes are enabled only on a finite prefix of any execution. As our algorithm is based on a token circulation, it is not a silent self-stabilizing algorithm. The first open question is to decide if our non silent self-stabilizing algorithm is optimal with respect to memory requirement. A recent work [10], which presents a non silent BFS-based leader election self-stabilizing algorithm requiring $O(\log \log n)$ bits of memory per process, leads us to think that we can improve the memory requirement of our algorithm. This naturally opens another question about the optimality of a silent self-stabilizing algorithm for minimum diameter spanning tree construction. An appealing way to answer this question is suggested in [7]. Using their solution, the question is reduced to provide a proof-labeling

scheme for this problem requiring $O(\log n)$ bits of memory per process. If such a labeling scheme exists, a straightforward adaptation of our self-stabilizing algorithm would be an optimal silent self-stabilizing algorithm for the minimum diameter spanning tree construction.

[1] Y. Afek and A. Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Chicago J. Theor. Comput. Sci.*, 1998, 1998.

[2] G. Antonoiu and P. Srimani. A self-stabilizing distributed algorithm to find the center of a tree graph. *Parallel Algorithms and Applications*, 10(3-4):237–248, 1997.

[3] Anish Arora, Paul C. Attie, Michael Evangelist, and Mohamed G. Gouda. Convergence of iteration systems. *Distributed Computing*, 7(1):43–53, 1993.

[4] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *WSS'01*, pages 19–34, 2001.

[5] L. Blin, F. Boubekeur, and S. Dubois. A self-stabilizing memory efficient algorithm for the minimum diameter spanning tree under an omnipotent daemon. In *IPDPS'15*, pages 1–10, 2015.

[6] L. Blin, S. Dolev, M. Potop-Butucaru, and S. Rovedakis. Fast self-stabilizing minimum spanning tree construction. In *DISC'10*, pages 480–494, 2010.

[7] L. Blin, P. Fraigniaud, and Boaz Patt-Shamir. On proof-labeling schemes versus silent self-stabilizing algorithms. In *SSS'14*, pages 18–32, 2014.

[8] L. Blin, M. Potop-Butucaru, and S. Rovedakis. A superstabilizing log (n)-approximation algorithm for dynamic steiner trees. In *SSS'09*, pages 133–148, 2009.

[9] L. Blin, M. Potop-Butucaru, and S. Rovedakis. Self-stabilizing minimum degree spanning tree within one from the optimal degree. *J. of Parallel and Distributed Computing*, 71(3):438–449, 2011.

[10] L. Blin and S. Tixeuil. Compact deterministic self-stabilizing leader election - the exponential advantage of being talkative. In *DISC'13*, pages 76–90, 2013.

[11] S. Bruell, S. Ghosh, M. Karaata, and S. Pemmaraju. Self-stabilizing algorithms for finding centers and medians of trees. *SIAM Journal of Computing*, 29(2):600–614, 1999.

[12] J. Burman and S. Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *DISC'07*, pages 92–107, 2007.

[13] F. Butelle, C. Lavault, and M. Bui. A uniform self-stabilizing minimum diameter tree algorithm (extended abstract). In *WDAG'95*, pages 257–272, 1995.

[14] Z. Collin and S. Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

[15] A. Cournier. A new polynomial silent stabilizing spanning-tree construction algorithm. In *SIROCCO'09*, pages 141–153, 2009.

[16] A. Cournier, S. Rovedakis, and V. Villain. The first fully polynomial stabilizing algorithm for BFS tree construction. In *OPODIS'11*, pages 159–174, 2011.

[17] A. Datta and L. Larmore. Leader election and centers and medians in tree networks. In *SSS'13*, pages 113–132, 2013.

[18] A. Datta, L. Larmore, and P. Vemula. An o(n)-time self-stabilizing leader election algorithm. *J. Parallel Distrib. Comput.*, 71(11):1532–1544, 2011.

[19] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, 1974.

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

[21] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *(PODC'90*, pages 103–117, 1990.

[22] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[23] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997.

[24] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):424–440, 1997.

[25] Shlomi Dolev and Ted Herman. Parallel composition of stabilizing algorithms. In *WSS'99*, pages 25–32, 1999.

[26] S. Dubois and S. Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.

[27] F. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report ic/2003/38, EPFL, 2003.

[28] Mohamed G. Gouda and Ted Herman. Adaptive programming. *IEEE Trans. Software Eng.*, 17(9):911–921, 1991.

[29] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991.

[30] S. Gupta, A. Bouabdallah, and P. Srimani. Self-stabilizing protocol for shortest path tree for multi-cast routing in mobile networks (research note). In *Euro-Par'00*, pages 600–604, 2000.

[31] R. Hassin and A. Tamir. On the minimum diameter spanning tree problem. *Information Processing Letters*, 53(2):109–111, 1995.

[32] J.-M. Ho, D. Lee, C.-H. Chang, and C. Wong. Minimum diameter spanning trees and related problems. *SIAM Journal of Computing*, 20(5):987–997, 1991.

[33] S. Holzer and R. Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *PODC'12*, pages 355–364, 2012.

[34] S.-T. Huang and N.-S. Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, 1992.

[35] S.-T. Huang and N.-S. Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993.

[36] T. Huang. A self-stabilizing algorithm for the shortest path problem assuming read/write atomicity. *J. Comput. Syst. Sci.*, 71(1):70–85, 2005.

[37] E. Korach, D. Rotem, and N. Santoro. Distributed algorithms for finding centers and medians in networks. *ACM Transactions on Programming Languages and Systems*, 6(3):380–401, 1984.

[38] A. Korman, S. Kutten, and T. Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an MST. In *PODC'11*, pages 311–320, 2011.

[39] A. Kosowski and L. Kuszner. A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In *PPAM'05*, pages 75–82, 2005.

[40] D. Peleg, L. Roditty, and E. Tal. Distributed algorithms for network diameter and girth. In *ICALP'12*, pages 660–672, 2012.

[41] F. Petit and V. Villain. Time and space optimality of distributed depth-first token circulation algorithms. In *WDAS'99*, pages 91–106, 1999.

[42] L. Roditty and V. Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *STOC'13*, pages 515–524, 2013.

[43] S. Tixeuil. *Algorithms and Theory of Computation Handbook*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall. CRC Press, Taylor & Francis Group, November 2009.

[44] George Varghese. Compositional proofs of self-stabilizing protocols. In *WSS'97*, pages 80–94, 1997.