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

Lélia Blin[1] Fadwa Boubekeur[2] Swan Dubois[3]

***Abstract*—The diameter of a network is one of the most fundamental network parameters. Being able to compute the diameter is an important problem in the analysis of large networks, and moreover this parameter has many important practical applications in real networks. As a consequence, it is natural to study this problem in a distributed system, and more specifically in a distributed system tolerant to transient faults. More specifically, we are interested in the problem to identify one of the centers of graph. Once done, we construct a minimum diameter spanning tree rooted in this center. Of course, the challenging problem is to compute one center of the graph.**

We present a uniform self-stabilizing algorithm for the minimum diameter spanning tree construction problem in the state model. Our protocol has several attractive features that makes it suitable for practical purposes. It is the first algorithm for this problem that operates under the *unfair adversary* (also called *unfair daemon*). In other words, no restriction is made on the distributed behavior of the system. Consequently, it is the hardest adversary to deal with. Moreover, 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 improvements are not achieved to the detriment of the convergence time, that stays reasonable with $O(n^2)$ rounds.

***Keywords*—Self-stabilization; Spanning tree; Center; Diameter; MDST; Unfair daemon;**

I. Introduction

Self-stabilization [1], [2], [3] 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.

[1] Université d'Evry-Val-d'Essonne F-91000, Evry, France
CNRS, UMR 7606, LIP6, F-75005, Paris, France
E-mail: lelia.blin@lip6.fr
Additional supports from the ANR project IRIS
[2] Sorbonne Universités, UPMC Université Paris 6, F-75005, Paris, France
CNRS, UMR 7606, LIP6, F-75005, Paris, France
E-mail: fadwa.boubekeur@lip6.fr
Additional supports from the ANR project IRIS
[3] Sorbonne Universités, UPMC Université Paris 6, F-75005, Paris, France
CNRS, UMR 7606, LIP6, F-75005, Paris, France
Inria, Équipe-projet REGAL, F-75005, Paris, France
E-mail: swan.dubois@lip6.fr

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 [4]. 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 bounded 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 with respect to 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 process). Note that this does not come at the price of degrading time performance.

a) Related works: Spanning tree construction was extensively studied in the context of distributed systems either in a fault-free setting or in presence of faults. There is a huge literature on the self-stabilizing construction of various kinds of trees, including spanning trees (ST) [5], [6], breadth-first search (BFS) trees [7], [8], [9], [10], [11], depth- first search (DFS) trees [12], [13], minimum-weight spanning trees (MST) [14], [15], shortest-path spanning trees [16], [17], minimum-degree spanning trees [18], Steiner Tree [19], etc. A survey on self-stabilizing distributed protocols for spanning tree construction can be found in [20].

The MDST problem is closely related to the determination of centers of the system [21]. 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 a MDST. As there exists many self-stabilizing solutions to BFS spanning tree construction, we focus in the following 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 by the way, to determine its centers) is to solving 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, [22] provides a good survey on recent distributed solutions to this problem and presents an almost optimal solution in synchronous settings. Note that there exist also some approximation results for this problem, *e.g.* [23], [24], 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 in order to reduce space requirement as we do in this work. In a synchronous and fault-free environment, we can cite [25] that presents the first algorithm for computing directly centers of a distributed system. In a self-stabilizing setting, some works [26], [27], [28] described solutions that are specific to tree topologies. The most related work to ours is from Butelle *et al.* [29]. 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.

b) 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, that improves the existing 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 reasonable time of convergence in $O(n^2)$ rounds (that is comparable with existing solutions [29]).

c) Organization of the paper: This paper is organized as follows. In section II, we formalize the model used afterwards. Section III is devoted to the description of our algorithm while Section IV contains a sketch of its correctness proof. Finally, we discuss some open questions in Section V.

II. Model and Definitions

d) 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 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 [2]) where communications between neighbors are modeled by direct reading of variables instead of exchange of messages. Every process has of 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 of 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* $\gamma \rightarrow \gamma'$ is defined as an atomic execution of a non-empty subset of enabled rules in γ that transitions the system from γ to γ'. An execution of an algorithm $\mathcal{A}$ is a maximal sequence of configurations $\epsilon = \gamma_0\gamma_1 \ldots \gamma_i\gamma_{i+1} \ldots$ such that, $\forall i \geq 0, \gamma_i \rightarrow \gamma_{i+1}$ is a step if γ_{i+1} exists (else γ_i is a *terminal* configuration). *Maximality* means that the sequence is either finite (and no action of $\mathcal{A}$ 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 $\gamma_i \rightarrow \gamma_{i+1}$ if v is enabled in γ_i and is *not* enabled in γ_{i+1}, yet did not execute any rule in step $\gamma_i \rightarrow \gamma_{i+1}$.

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 of their characteristics (like fairness, distribution, ...), see [30] 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 choosen 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 [31]. 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.

e) Self-stabilization: Let $\mathcal{P}$ be a problem to solve. A *specification* of $\mathcal{P}$ is a predicate that is satisfied by every algorithm solving $\mathcal{P}$. We recall the definition of self-stabilization.

Definition 1 (Self-stabilization [1]): 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 $\mathcal{S}_\mathcal{P}$.

III. 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 III-A. Then, Sections III-B, III-C, III-D, and III-E 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.

A. 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 an 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 [32] corresponds to our criteria in terms of daemon, memory requirement and convergence time. Indeed, Datta *et al.* [32] 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 all existing self-stabilizing algorithms for token circulation, we choose to slightly adapt an algorithm of Petit and Villain [33]. 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, in order 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 sequentially 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 at 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. In addition, among the centers, we elect the one with the highest identity to be the root of the minimum diameter spanning tree.

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.

B. Leader Election and Spanning Tree Construction

The first layer of our algorithm executes a self-stabilizing algorithm from Datta *et al.* [32] 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 [32]. 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 $\mathrm{N}_{\texttt{Backbone}}(v)$. 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 $\mathrm{N}_{\texttt{Backbone}}(v) = \text{child}(v) \cup \{\mathsf{p}_v\}$. We also define the predicate $\mathrm{Root}(v)$ 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.

C. Token Circulation

The second layer of our algorithm is a slight adaptation of a self-stabilizing token circulation algorithm by Petit and Villain [33]. The (eventually unique) token circulates infinitely often over the `Backbone` in a DFS order. This algorithm operates

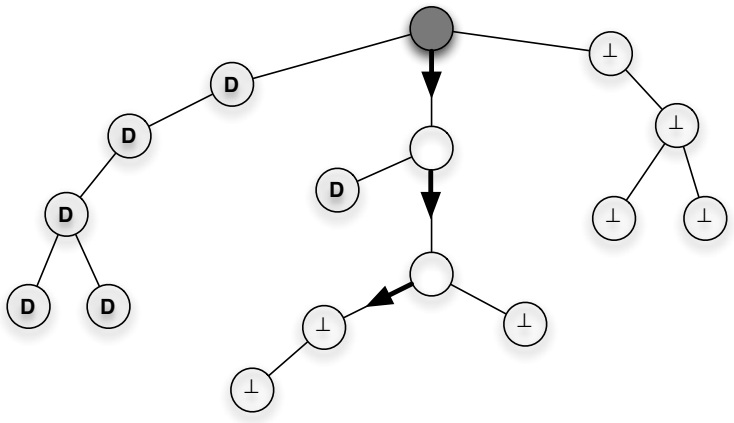

Fig. 1. Illustration of the variable next of the token circulation layer. The gray process is the root of the Backbone. D stands for the value done.

$$
\begin{array}{rcl}
\mathrm{Next}(v) & = & \begin{cases} u & \text{if } \exists u \in \mathsf{child}(v), \mathrm{ID}_u = \min\{\mathrm{ID}_w \mid w \in \mathsf{child}(v) \wedge (\mathrm{ID}_w \succ \mathsf{next}_v)\} \\ \bot & \text{if } \mathrm{Root}(v) \wedge \mathsf{next}_v \in \mathsf{child}(v) \wedge (\forall u \in \mathsf{child}(v), \mathsf{next}_v \succ \mathrm{ID}_u \vee \mathsf{next}_v = \mathrm{ID}_u) \\ \text{done} & \text{otherwise} \end{cases} \\
\mathrm{ErValues}(v) & \equiv & \mathsf{next}_v \notin \mathsf{child}(v) \cup \{\bot, \text{done}\} \\
\mathrm{PermR}(v) & \equiv & (\mathrm{Next}(v) = \bot) \vee (\mathsf{next}_{\mathrm{Next}(v)} = \bot) \\
\mathrm{PermNd}(v) & \equiv & (\mathrm{Next}(v) = \text{done}) \vee (\mathsf{next}_{\mathrm{Next}(v)} = \bot) \\
\mathrm{BackR}(v) & \equiv & \mathrm{Root}(v) \wedge (\mathsf{next}_v = u) \wedge (\mathsf{next}_u = \text{done}) \wedge \mathrm{PermR}(v) \\
\mathrm{BackNd}(v) & \equiv & \neg\mathrm{Root}(v) \wedge (\mathsf{next}_{\mathsf{p}_v} = v) \wedge (\mathsf{next}_v \in \mathsf{child}(v)) \wedge (\mathsf{next}_{\mathrm{Next}(v)} = \text{done}) \wedge \mathrm{PermNd}(v) \\
\mathrm{TokenD}(v) & \equiv & (\mathsf{next}_v = \bot) \wedge \big((\mathrm{Root}(v) \wedge \mathrm{PermR}(v)) \vee (\neg\mathrm{Root}(v) \wedge (\mathsf{next}_{\mathsf{p}_v} = v) \wedge \mathrm{PermNd}(v)\big)
\end{array}
$$

Fig. 2. Predicates and fonction dedicated to the token circulation.

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: $\mathsf{next}_v \in \{\bot, \text{done}, N_v\}$. This variable stores the state of the process with respect to the current token circulation. The value $\bot$ 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. 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 $\text{done} \succ \mathrm{ID}_v \succ \bot$.

The formal presentation of this layer is provided by Algorithm 1 and Figure 2 sums up the function and predicates used by this algorithm. This algorithm consists of two rules. The first one, $\mathbb{R}_{\texttt{ErToken}}$, is used to perform the convergence towards a unique token 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).

We modify the original algorithm of [33] 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 III-D 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 $\mathrm{TokenD}(v)$. 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 performs the forward token circulation (exactly as in the original algorithm of [33]).

We also slightly modify the backward token circulation in

Algorithm 1 Token circulation for process v

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

$\mathbb{R}_{\texttt{Backward}}$: $\neg\mathrm{ErValues}(v) \wedge (\mathrm{BackNd}(v) \vee \mathrm{BackR}(v)) \wedge (\mathsf{Rbfs}_{\mathrm{Next}(v)} \neq \mathrm{ID}(\mathrm{Next}(v))) \rightarrow \mathsf{next}_v := \mathrm{Next}(v);$

order to ensure the converge 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 variable Rbfs_u dedicated to the BFS construction (see Section III-D 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 Rbfs_u is equal to u. If u wrongly believes to have the token, it is able to detect it locally and to correct this error within a finite time (see Section III-D), that ensures us the token to be 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, that 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.

D. 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 $\texttt{BFS}(r)$ (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 $\texttt{Backbone}$ (we perform this broadcast by re-orienting temporarily the $\texttt{Backbone}$ towards r) Then, we use a classical BFS spanning tree construction borrowed from [11] 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 $\texttt{BFS}(r)$ (and not earlier). As we already said, this gathering is made by a wave from the leaves to the root of $\texttt{BFS}(r)$. Each leave of $\texttt{BFS}(r)$ 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 $\texttt{BFS}(r)$. 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 $\texttt{BFS}(r)$ 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 $\texttt{BFS}(r)$, this process "cleans" its eccentricity variable (that is, it erases the current value of this variable and replaces it by a specific value). Then, all processes on the path of $\texttt{BFS}(r)$ between r and v clean their eccentricity variables in a upward process. In other words, we maintain at least one path in $\texttt{BFS}(r)$ in which all the eccentricity variables are cleaned until $\texttt{BFS}(r)$ 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 $\texttt{Backbone}$ to root it at the process r with TokenD(r)=true. We call this oriented tree $\texttt{Backbone}(r)$. For this purpose, we introduce the function $\mathrm{p\_next}(v)$ for each process v. This function returns the identity of the neighbor of v that belongs to the path of $\texttt{Backbone}$ from v to r. More precisely, $\mathrm{p\_next}$

$$
\begin{array}{rcl}
\mathrm{RootBFS}(v) & \equiv & (\mathsf{Rbfs}_v, \mathsf{Pbfs}_v, \mathsf{dbfs}_v) = (\mathrm{ID}(v), \bot, 0) \\
\mathrm{On}(v) & \equiv & \{u \mid u \in N(v) \wedge (\mathsf{Rbfs}_u = \mathsf{Rbfs}_v)\} = N(v) \\
\mathrm{ChBFS}(v) & = & \{u \mid u \in N(v) \wedge (\mathsf{Pbfs}_u, \mathsf{Rbfs}_u, \mathsf{dbfs}_u) = (v, \mathsf{Rbfs}_v, \mathsf{dbfs}_v + 1)\} \\
\mathrm{Cand}(v) & = & \{u \mid u \in N(v) \setminus \mathrm{ChBFS}(v)\} \\
\mathrm{Best}(v) & = & \begin{cases} \min\{\mathrm{ID}(u) \mid \mathsf{dbfs}_u = \min\{\mathsf{dbfs}_w \mid w \in \mathrm{Cand}(v)\}\} & \text{if } \mathsf{dbfs}_u < \mathsf{dbfs}_v + 1 \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 3. Predicates and functions dedicated to the BFS construction

$$
\begin{array}{rcl}
\mathsf{MaxE}(v) & = & \max\{\mathsf{dbfs}_v, \max\{\mathsf{Dbfs}_u \mid u \in \mathrm{ChBFS}(v)\}\} \\
\mathsf{GoodEN}(v) & \equiv & (\forall u \in \mathrm{ChBFS}(v), \mathsf{Dbfs}_u \in \mathbb{N}) \wedge (\mathsf{Dbfs}_v = \mathsf{MaxE}(v)) \wedge (\mathsf{Dbfs}_{\mathsf{Pbfs}_v} \in \{\downarrow, \mathbb{N}\}) \\
\mathsf{GoodC}(v) & \equiv & (\mathsf{Dbfs}_v \in \{\uparrow, \downarrow\}) \wedge (\forall u \in \mathrm{ChBFS}(v), \mathsf{Dbfs}_u \neq \bot) \\
\mathsf{GoodE}(v) & \equiv & [(\mathsf{Dbfs}_v, \mathsf{Dbfs}_{\mathsf{Pbfs}_v}) \in \{(\bot,\bot), (\uparrow,\bot)(\uparrow,\uparrow), (\uparrow,\downarrow), (\downarrow,\downarrow)\} \wedge \mathsf{GoodC}(v)] \vee \mathsf{GoodEN}(v) \\
\mathrm{Ecc}(v) & = & \begin{cases} \bot & \text{if } \neg\mathsf{GoodE}(v) \\ \uparrow & \text{if } \mathsf{GoodE}(v) \wedge (\mathsf{Dbfs}_v = \bot) \wedge (\forall u \in \mathrm{ChBFS}(v), \mathsf{Dbfs}_u = \uparrow) \\ \downarrow & \text{if } \mathsf{GoodE}(v) \wedge (\mathsf{Dbfs}_v = \uparrow) \wedge (\mathsf{Dbfs}_{\mathsf{Pbfs}_v} = \downarrow) \\ \mathsf{MaxE}(v) & \text{if } \mathsf{GoodE}(v) \wedge (\mathsf{Dbfs}_v = \downarrow) \wedge (\forall u \in \mathrm{ChBFS}(v), \mathsf{Dbfs}_u \in \mathbb{N}) \end{cases}
\end{array}
$$

Fig. 4. Predicates and function related to eccentricity computation

is defined as follow:

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

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

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

- $\mathsf{Ecc}_v \in \mathbb{N} \cup \{\bot\}$ is the eccentricity of process v;
- $\mathsf{Rbfs}_v \in \mathbb{N}$ is the identity of the root in $\mathtt{BFS}(r)$;
- $\mathsf{Pbfs}_v \in \mathbb{N} \cup \{\bot\}$ is the identity of the parent of v in $\mathtt{BFS}(r)$;
- $\mathsf{dbfs}_v \in \mathbb{N} \cup \{\bot, \infty\}$ is the distance between v and the root in $\mathtt{BFS}(r)$;
- $\mathsf{Dbfs}_v \in \mathbb{N} \cup \{\bot, \downarrow, \uparrow\}$ is the maximum distance between the root r and the farthest leaf in the sub-tree of v in $\mathtt{BFS}(r)$;

Now, we can present rules of the third layer of our algorithm. These rules make use of some predicates and functions that are defined in Figures 3 (for those regarding BFS spanning tree construction) and 4 (for those that are related to eccentricity computation). 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 $\mathrm{TokenD}(v) = true$) 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 III-C). Once the process r received the token, its predicate $\mathrm{TokenD}(r)$ becomes true. In this state, the process r can applied only three rules.

The rule $\mathbb{R}_{\mathtt{StartBFS}}$ starts the computation of $\mathtt{BFS}(r)$ since r takes a state indicating that it is the the root of the current BFS spanning tree. The rule $\mathbb{R}_{\mathtt{StartEcc}}$ cleans the eccentricity variable of r when needed (that is, after a fake computation of eccentricity due to the asynchrony of the system). Finally, the rule $\mathbb{R}_{\mathtt{EndBFS}}$ 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 III-C), and updates one variable for communicating the new eccentricity to the centers determination layer (see Section III-E 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 $\mathbb{R}_{\mathtt{Tree}}$ 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 $\mathbb{R}_{\mathtt{Tree}}$ also detects some local errors. As an example, the variable Dbfs_v of a process (used to collect the eccentricity of r) must not have an integer value if one of the children (in $\mathtt{BFS}(r)$) of this process is not in the same case. The rule $\mathbb{R}_{\mathtt{BFS}}$ performs the BFS construction in itself. Details of this construction are managed throughout predicates and functions defined in Figure 3. Finally, the rule $\mathbb{R}_{\mathtt{Ecc}}$ deals with the tricky

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

$$\mathbb{R}_{\texttt{StartBFS}}: \neg\mathrm{RootBFS}(v) \longrightarrow (\mathsf{Rbfs}_v, \mathsf{dbfs}_v, \mathsf{Dbfs}_v) := (\mathrm{ID}_v, 0, \perp)$$

$$\mathbb{R}_{\texttt{StartEcc}}: \mathrm{RootBFS}(v) \wedge \mathrm{On}(v) \wedge (\forall u \in \mathrm{ChBFS}(v), \mathsf{Dbfs}_u = \uparrow) \longrightarrow \mathsf{Dbfs} = \downarrow$$

$$\mathbb{R}_{\texttt{EndBFS}}: \mathrm{RootBFS}(v) \wedge \mathrm{On}(v) \wedge (\forall u \in \mathrm{ChBFS}(v), \mathsf{Dbfs}_u \in \mathbb{N}) \longrightarrow \mathsf{Ecc}_v := \mathrm{Ecc}(v), \mathsf{next}_v := \mathrm{Next}(v); \ \mathsf{MinEUP}_v := \mathsf{MinEcc}(v);$$

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

$$\mathbb{R}_{\texttt{Tree}}: (\mathsf{Rbfs}_v \neq \mathsf{Rbfs}_{\mathrm{p\_next}(v)}) \longrightarrow (\mathsf{Rbfs}_v, \mathsf{Pbfs}_v, \mathsf{dbfs}_v, \mathsf{Dbfs}_v) := (\mathsf{Rbfs}_{\mathrm{p\_next}(v)}, \perp, \infty, \perp)$$

$$\mathbb{R}_{\texttt{BFS}}: (\mathsf{Rbfs}_v = \mathsf{Rbfs}_{\mathrm{p\_next}_v}) \wedge \mathrm{On}(v) \wedge (\mathrm{Best}(v) \neq \perp) \longrightarrow (\mathsf{Pbfs}_v, \mathsf{dbfs}_v, \mathsf{Dbfs}_v) := (\mathrm{Best}(v), \mathsf{dbfs}_{\mathrm{Best}(v)} + 1, \perp);$$

$$\mathbb{R}_{\texttt{Ecc}}: (\mathsf{Rbfs}_v = \mathsf{Rbfs}_{\mathrm{p\_next}_v}) \wedge \mathrm{On}(v) \wedge (\mathrm{Best}(v) = \perp) \wedge (\mathsf{Dbfs}_v \neq \mathrm{Ecc}(v)) \longrightarrow \mathsf{Dbfs}_v := \mathrm{Ecc}(v)$$

phase of eccentricity leaves-to-root gathering with the cleaning mechanism explained above. This task is implemented with the help of predicates and functions defined in Figure 4.

E. 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 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 III-D) 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. We define the following predicate: $\mathsf{MinEcc}(v) \equiv \min\{\mathsf{Ecc}_v, \min\{\mathsf{MinE}_u \mid u \in \mathrm{child}(v)\}$.

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 the system is identified, 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.

IV. Sketch of Proof of the Algorithm

Due to space limitation, we present only a sketch of the formal proof of our self-stabilizing algorithm that aims to provide all main arguments of the formal proof in a nutshell[1].

Our objective is to prove the following result.

Theorem 1: The algorithm $\mathcal{SSCC}$ presented in Section III is a self-stabilizing algorithm that computes the center(s) 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.

The proof of this theorem is done thourough four main steps that are described by the following lemmas. Note that the $O(\log n)$ bits of memory requirement follows directly from the definition of the variables of $\mathcal{SSCC}$. Before presenting these results and their proofs, we need to introduce some definitions and notations.

A classical way to prove the self-stabilization of an algorithm is to prove that 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 $\mathcal{A}$.

Let us now define some configurations sets used in our proof. $\Gamma_{1-\texttt{Token}} \subseteq \Gamma$ is the set of configurations in which there exists exactly one token (*i.e.* no process is enabled by rule $\mathbb{R}_{\texttt{ErrToken}}$). $\Gamma_{\texttt{Ecc}} \subseteq \Gamma_{1-\texttt{Token}}$ is the set of configurations of $\Gamma_{1-\texttt{Token}}$ where each process has correctly computed its own eccentricity (that is each process $v \in V$ satisfies $\mathsf{Ecc}_v =$

[1] Note that a full version of this proof is available at: http://pagesperso-systeme.lip6.fr/Swan.Dubois/proofIPDPS2015.pdf.

Algorithm 4 Computation of the minimum eccentricity for process v

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

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

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

$e^*(v)$ where $e^*(v)$ is the eccentricity of process v). Finally, $\Gamma_{\texttt{Centers}} \subseteq \Gamma_{\texttt{Ecc}}$ is the set of configurations of $\Gamma_{\texttt{Ecc}}$ such that each process knows the minimal eccentricity of the system (*i.e.* $\mathsf{MinE}_v = E^*$ for all v in V where E^* denotes the minimal eccentricity of a process in the system) and knows hence if it is a center or not.

Now, we are able to present the four milestones of our proof and the key concepts of their proofs.

Lemma 1: $\Gamma \rhd \Gamma_{1-\texttt{Token}}$ in $O(n^2)$ rounds for $\mathcal{SSCC}$ under the distributed unfair daemon.

Intuitively, this lemma means that, starting from any configuration, any execution of $\mathcal{SSCC}$ reaches in $O(n^2)$ rounds a configuration in which exactly one token is present and that this property remains forever true afterwards.

The proof of this lemma is based on the ones of [32] (for the leader election and `Backbone` construction) and of [33] (for the token circulation). Indeed, note that we do not modifiy at all the first algorithm. Although we slightly modify the second one, these modifications have no effect on this part of the proof since we do not modify the correction rule that ensures the "cleaning" of overnumerous tokens. Indeed, note that the token circulation algorithm from [33] ensures this cleaning phase completes even if tokens do not move (that may be possible in our case since we modify the release condition to wait after the completion of the second layer). The closure property follows directly from the one of [33].

Lemma 2: Any execution of $\mathcal{SSCC}$ under the distributed unfair daemon starting from a configuration of $\Gamma_{1-\texttt{Token}}$ contains an execution of $\mathbb{R}_{\texttt{EndBFS}}$ by some process.

In other words, this lemma says that no process may hold the token during an infinite time without releasing it infinitely often. This part of the proof is the more tricky since it consists to prove the correctness of the interaction between two layers of our algorithm. Indeed, we delegate the release of the token to the eccentricity computation layer (refer to Section III-D). Then, we have to prove, for any process that holds the token, that the construction of its BFS spanning tree and the gathering of its eccentricity end in a finite time (we do not care about the correctness of the result here).

This proof is performed by exhibiting a *potential function*. A potential function is a bounded function that associates to each configuration of the system a value that strictly decreases at each application of a rule by the algorithm. Hence, if the minimal value of the function is associated to 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, [11] 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 this lemma.

Lemma 3: If a process v executes $\mathbb{R}_{\texttt{StartBFS}}$ in any execution of $\mathcal{SSCC}$ under the distributed unfair daemon starting from a configuration of $\Gamma_{1-\texttt{Token}}$, then $\mathsf{Ecc}_v = e^*(v)$ within a finite time.

Roughly speaking, this lemma ensures that, when a process obtains the token in a forward circulation, the result returned by the eccentricity computation layer (that completes within a finite time by the previous lemma) is correct.

The proof of this lemma is the second difficult one. Indeed, we have to prove that, if the eccentricity computation has been effectively started by a process (contrarily to the previous lemma, we do not care about fake computations due to memory corruption in the initial configuration), then this computation gives the correct eccentricity. As the eccentricity computation is a quite simple process, we have only to prove that the eccentricity computation cannot terminate precociously (that is, before the stabilization of the BFS spanning tree construction). For that, we prove that our algorithm maintains a particular structure, that we call a *blocking path*, in the BFS under construction until this one is not stabilized. A blocking path is one of the longest path from the root to another process in the tree such that each process on this path has a non-numerical value for its Dbfs variable. We prove that the existence of such a blocking path prevents the root of the BFS spanning tree to release the token. Then, as our algorithm ensures the existence of a blocking path until the stabilization of the BFS spanning tree construction, we obtain the lemma.

Lemma 4: $\Gamma_{1-\texttt{Token}} \rhd \Gamma_{\texttt{Ecc}}$ and $\Gamma_{\texttt{Ecc}} \rhd \Gamma_{\texttt{Centers}}$ in $O(n^2)$ rounds for $\mathcal{SSCC}$ under the distributed unfair daemon.

Intuitively, this lemma states that, starting from a configuration where only one token exists, the algorithm $\mathcal{SSCC}$ computes correctly the eccentricity of each process and then determines the correct minimal eccentricity within $O(n^2)$ rounds.

The proof of this lemma is quite simple. Indeed, it is sufficient to apply inductively the results of the two previous lemmas to ensures the correct process eccentricity computation (the token visits infinitely often each node and the eccentricity computation is correct after the first release of the token). The last part of the proof does not contains any particular difficulty since it consists to prove the correctness of the gathering of

the minimal eccentricity by the root and its diffusion along a stabilized tree.

Once these four lemmas proved, it is straightforward to deduce the main theorem and hence, to conclude the proof.

V. 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 [34], that 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 [35]. 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.

References

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

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

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

[4] J.-M. Ho, D. Lee, C.-H. Chang, and C. Wong, "Minimum diameter spanning trees and related problems," *SIAM Journal of Computing*, vol. 20, no. 5, pp. 987–997, 1991.

[5] A. Cournier, "A new polynomial silent stabilizing spanning-tree construction algorithm," in *SIROCCO'09*, 2009, pp. 141–153.

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

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

[8] J. Burman and S. Kutten, "Time optimal asynchronous self-stabilizing spanning tree," in *DISC'07*, 2007, pp. 92–107.

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

[10] S. Dolev, A. Israeli, and S. Moran, "Uniform dynamic self-stabilizing leader election," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 4, pp. 424–440, 1997.

[11] S.-T. Huang and N.-S. Chen, "A self-stabilizing algorithm for constructing breadth-first trees," *Information Processing Letters*, vol. 41, no. 2, pp. 109–117, 1992.

[12] Z. Collin and S. Dolev, "Self-stabilizing depth-first search," *Information Processing Letters*, vol. 49, no. 6, pp. 297–301, 1994.

[13] S. Huang and N. Chen, "Self-stabilizing depth-first token circulation on networks," *Distributed Computing*, vol. 7, no. 1, pp. 61–66, 1993.

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

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

[16] 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*, 2000, pp. 600–604.

[17] T. Huang, "A self-stabilizing algorithm for the shortest path problem assuming read/write atomicity," *J. Comput. Syst. Sci.*, vol. 71, no. 1, pp. 70–85, 2005.

[18] 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*, vol. 71, no. 3, pp. 438–449, 2011.

[19] ——, "A superstabilizing log (n)-approximation algorithm for dynamic steiner trees," in *SSS'09*, 2009, pp. 133–148.

[20] F. Gärtner, "A survey of self-stabilizing spanning-tree construction algorithms," EPFL, Technical Report IC/2003/38, 2003.

[21] R. Hassin and A. Tamir, "On the minimum diameter spanning tree problem," *Information Processing Letters*, vol. 53, no. 2, pp. 109–111, 1995.

[22] S. Holzer and R. Wattenhofer, "Optimal distributed all pairs shortest paths and applications," in *PODC'12*, 2012, pp. 355–364.

[23] D. Peleg, L. Roditty, and E. Tal, "Distributed algorithms for network diameter and girth," in *ICALP'12*, 2012, pp. 660–672.

[24] L. Roditty and V. Williams, "Fast approximation algorithms for the diameter and radius of sparse graphs," in *STOC'13*, 2013, pp. 515–524.

[25] E. Korach, D. Rotem, and N. Santoro, "Distributed algorithms for finding centers and medians in networks," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 3, pp. 380–401, 1984.

[26] G. Antonoiu and P. Srimani, "A self-stabilizing distributed algorithm to find the center of a tree graph," *Parallel Algorithms and Applications*, vol. 10, no. 3-4, pp. 237–248, 1997.

[27] S. Bruell, S. Ghosh, M. Karaata, and S. Pemmaraju, "Self-stabilizing algorithms for finding centers and medians of trees," *SIAM Journal of Computing*, vol. 29, no. 2, pp. 600–614, 1999.

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

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

[30] S. Dubois and S. Tixeuil, "A taxonomy of daemons in self-stabilization," ArXiv eprint, Tech. Rep. 1110.0334, October 2011.

[31] S. Dolev, A. Israeli, and S. Moran, "Resource bounds for self-stabilizing message-driven protocols," *SIAM J. Comput.*, vol. 26, no. 1, pp. 273–290, 1997.

[32] A. Datta, L. Larmore, and P. Vemula, "An o(n)-time self-stabilizing leader election algorithm," *J. Parallel Distrib. Comput.*, vol. 71, no. 11, pp. 1532–1544, 2011.

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

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

[35] L. Blin, P. Fraigniaud, and B. Patt-Shamir, "On proof-labeling schemes versus silent self-stabilizing algorithms," in *SSS'14*, 2014, pp. 18–32.