J. Parallel Distrib. Comput. 117 (2018) 50–62

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

A self-stabilizing memory efficient algorithm for the minimum diameter spanning tree under an omnipotent daemon

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

[a] Sorbonne Université, Université d'Evry-Val-d'Essonne, CNRS, LIP6 UMR 7606, 4 place Jussieu, 75005 Paris, France
[b] DAVID, UVSQ, Université Paris-Saclay, 45 Avenue des Etats-Unis, 78035 Versailles, France
[c] Sorbonne Université, UPMC Univ Paris 06, CNRS, INRIA, LIP6 UMR 7606, 4 place Jussieu, 75005 Paris, France

HIGHLIGHTS

- A self-stabilizing algorithm for the minimum diameter spanning tree construction problem is proposed.
- It is the first one that operates under the unfair and distributed scheduler.
- It needs only $O(\log n)$ bits of memory per process (where n is the number of processes).
- These features are not achieved to the detriment of the convergence time, that stays polynomial.

ARTICLE INFO

Article history:
Received 22 July 2016
Received in revised form 18 October 2017
Accepted 17 February 2018
Available online 27 February 2018

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

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.

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

* Correspondence to: Sorbonne Université, LIP6 - case 26-00/207, 4 place Jussieu, 75005 Paris Cedex 5, France.
E-mail addresses: lelia.blin@lip6.fr (L. Blin), fadwa.boubekeur@lip6.fr (F. Boubekeur), swan.dubois@lip6.fr (S. Dubois).

https://doi.org/10.1016/j.jpdc.2018.02.007
0743-7315/

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. A preliminary version of this work appears in [5].

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 [1,12,16,25,35], depth-first search (DFS) trees [14,34], 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 afterward. 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 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 *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 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 [24]. 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 $\mathcal{S}_\mathcal{P}$.