Universal Loop-Free Super-Stabilization

Lélia Blin, Stephane Rovedakis, Maria Potop-Butucaru, Sébastien Tixeuil

Universal Loop-Free Super-Stabilization

Lélia Blin[4,2], Maria Potop-Butucaru[1,2,3], Stephane Rovedakis[4,5], and Sébastien Tixeuil[1,2]

[1] Univ. Pierre & Marie Curie - Paris 6, France
[2] LIP6-CNRS UMR 7606, France
maria.gradinariu@lip6.fr, sebastien.tixeuil@lip6.fr
[3] INRIA REGAL, France
[4] Université d'Evry Val d'Essonne, France
lelia.blin@ibisc.univ-evry.fr, stephane.rovedakis@ibisc.univ-evry.fr
[5] IBISC-EA 4526, France

Abstract. We propose an univesal scheme to design loop-free and super-stabilizing protocols for constructing spanning trees optimizing any tree metrics (not only those that are isomorphic to a shortest path tree). Our scheme combines a novel super-stabilizing loop-free BFS with an existing self-stabilizing spanning tree that optimizes a given metric. The composition result preserves the best properties of both worlds: super-stabilization, loop-freedom, and optimization of the original metric without any stabilization time penalty. As case study we apply our composition mechanism to two well known metric-dependent spanning trees: the maximum-flow tree and the minimum degree spanning tree.

1 Introduction

New distributed emergent networks such as P2P or sensor networks face high churn (nodes and links creation or destruction) and various privacy and security attacks that are not easily encapsulated in the existing distributed models. One of the most versatile techniques to ensure forward recovery of distributed systems is that of *self-stabilization* [1,2,3]. A distributed algorithm is self-stabilizing if after faults and attacks hit the system and place it in some arbitrary global state, the system recovers from this catastrophic situation without external (*e.g.* human) intervention in finite time. A recent trend in self-stabilizing research is to complement the self-stabilizing abilities of a distributed algorithm with some additional *safety* properties that are guaranteed when the permanent and intermittent failures that hit the system satisfy some conditions. In addition to being self-stabilizing, a protocol could thus also tolerate crash faults [4,5], nap faults [6,7], Byzantine faults [8,9,10,11], a limited number of topology changes [12,13,14] and sustained edge cost changes [15,16].

The last two properties are especially relevant when building optimized spanning trees in dynamic networks, since the cost of a particular edge and the network topology are likely to evolve through time. If a spanning tree protocol is *only* self-stabilizing, it may adjust to the new costs or network topology in

such a way that a previously constructed spanning tree evolves into a disconnected or a looping structure (of course, in the absence of network modifications, the self-stabilization property guarantees that *eventually* a new spanning tree is constructed). Now, a packet routing algorithm is *loop free* [17,18] if at any point in time the routing tables are free of loops, despite possible modification of the edge-weights in the graph (*i.e.*, for any two nodes u and v, the actual routing tables determines a simple path from u to v, at any time). The *loop-free* property [15,16] in self-stabilization guarantees that, a spanning tree being constructed (not necessarily a "minimal" spanning tree for some metric), then the self-stabilizing convergence to a "minimal" spanning tree maintains a spanning tree at all times (obviously, this spanning tree is not "minimal" at all times). The consequence of this safety property in addition to that of self-stabilization is that the spanning tree structure can still be used (*e.g.* for routing) while the protocol is adjusting, and makes it suitable for networks that undergo such very frequent dynamic changes. In order to deal with the network churn, *super-stabilization* captures the quality of services a tree stucture can offer during and after a localized topological change. Super-stabilization [19] is an extension of self-stabilization for dynamic settings. The idea is to provide some minimal guarantees (a *passage* predicate) while the system repairs after a topology change. In the case of optimized spanning trees algorithms while converging to a correct configuration (*i.e.* an optimized tree) after some topological change, the system keeps offering the tree service during the stabilization time to all members that have not been affected by this modification.

Related works Relatively few works investigate merging self-stabilization and loop free routing, with the notable exception of [15,16,20]. In [15], Cobb and Gouda propose a self-stabilizing algorithm which constructs spanning trees with loop-free property. This algorithm allows to optimize general tree metrics from a considered root, such as bandwidth, delay, distance, etc ... To this end, each node maintains a value which reflects its cost from the root for the optimized metric, for example the maximum amount of bandwith on its path to reach the root. The basic idea is to allows a node to select a neighbor as its parent if this one offers a better cost. To avoid loop creation, when the cost of its parent or the edge-cost to its parent changed a propagation of information is started to propagate the new value. A node can safely change its parent if its propagation of information is ended. Thus, a node can not select one of its descendant as its parent. This algorithm requires a upper bound on the network diameter known to every participant to detect the presence of a cycle and to reset the states of the nodes. Each node maintains its distance from the root and a cycle is detected when the distance of a node is higher than the diameter upper bound.

Johnen and Tixeuil [16] propose another loop-free self-stabilizing algorithm constructing spanning trees, which makes no assumption on the network. This algorithm follows the same approach used in [15], that is using propagation of information in the tree. As in [15], this second algorithm constructs trees optimizing several metrics from a root, *e.g.*, depth first search tree, breadth first search tree, shortest path tree, etc. Since no upper bound on the network

diameter is used, when a cycle is present in the initial network state the protocol continues the initiate propagation of information to grow the value of the nodes in the cycle. The values of these nodes grow until the value of a node reaches a threshold which is the value of a node out of the cycle. Thus, the node reaching this threshold discover a neighbor which offers a better value and can select it to break the cycle. When no cycle is present in the network, the system converges to a correct state.

Also, both protocols use only a reasonable amount of memory ($O(\log n)$ bits per node) and consider networks with static topology and dynamic edge costs. However, the metrics that are considered in [15,16] are derivative of the shortest path (distance graph) metric, that is considered a much easier task in a distributed setting than that of tree metrics not based on distances, *e.g.*, minimum spanning tree, minimum degree spanning tree, maximum leaf spanning tree, etc. Indeed, the associated metric is *locally optimizable* [21], allowing essentially locally greedy approaches to perform well. By contrast, some sort of *global optimization* is needed for tree metrics not based on distances, which often drives higher complexity costs and thus less flexibility in dynamic networks.

Recently, [20] proposed a loop-free self-stabilizing algorithm to solve the minimum spanning tree problem for networks, assuming a static topology but dynamic edge costs. None of the previously mentioned works can cope with both dynamic edge changes (loop-freedom) and dynamic local topology changes (super-stabilization). Also, previous works are generic only for local tree metrics, while global tree metrics require *ad hoc* solutions.

Our contributions We propose a distributed generic scheme to transform existing self-stabilizing protocols that construct spanning tree optimizing an arbitrary tree metric (local or global), adding loop-free and super-stabilizing properties to the input protocol. Contrary to existing generic protocols [15,16], our approach provides the loop-free property for *any* tree metric (global or local, rather than only local). Our technique also adds super-stabilization, which the previous works do not guarantee. Our scheme consists in composing a distributed self-stabilizing spanning tree algorithm (established and proved to be correct for a given metric) with a novel BFS construction protocol that is both loop-free and super-stabilizing. The output of our scheme is a loop-free super-stabilizing spanning tree optimizing the tree metric of the input protocol. Moreover, we provide complexity analysis for the BFS construction in both static and dynamic settings. We examplify our scheme with two case study: the maximum flow tree and the minimum degree spanning tree. In both cases, the existing self-stabilizing algorithms can be enhanced via our method with both loop-free and super-stabilizing properties. Interestingly enough, the stabilization time complexity of the original protocols is not worsen by the transformation.

2 Model and notations

We consider an undirected weighted connected network $G = (V, E, w)$ where V is the set of nodes, E is the set of edges and $w : E \to \mathbb{R}^+$ is a positive cost

function. Nodes represent processors and edges represent bidirectional communication links. Additionally, we consider that $G = (V, E, w)$ is a dynamic network in which the weight of the communication links and the sets of nodes and edges may change. We consider anonymous networks (i.e., processors have no IDs), with one distinguished node, called the *root*[6]. Throughout the paper, the root is denoted r. We denote by $\deg(v)$ the number of v's neighbors in G. The $\deg(v)$ edges incident to any node v are labeled from 1 to $\deg(v)$, so that a processor can distinguish the different edges incident to a node.

The processors asynchronously execute their programs consisting of a set of variables and a finite set of rules. The variables are part of the shared register which is used to communicate with the neighbors. A processor can read and write its own registers and can only read the shared registers of its neighbors. Each processor executes a program consisting of a sequence of guarded rules. Each *rule* contains a *guard* (boolean expression over the variables of a node and its neighborhood) and an *action* (update of the node variables only). Any rule whose guard is *true* is said to be *enabled*. A node with one or more enabled rules is said to be *privileged* and may make a *move* executing the action corresponding to the chosen enabled rule.

A *local state* of a node is the value of the local variables of the node and the state of its program counter. A *configuration* of the system $G = (V, E)$ is the cross product of the local states of all nodes in the system. The transition from a configuration to the next one is produced by the execution of an action of at least one node. A *computation* of the system is defined as a *weakly fair, maximal* sequence of configurations, $e = (c_0, c_1, \ldots c_i, \ldots)$, where each configuration c_{i+1} follows from c_i by the execution of a single action of at least one node. During an execution step, one or more processors execute an action and a processor may take at most one action. *Weak fairness* of the sequence means that if any action in G is continuously enabled along the sequence, it is eventually chosen for execution. *Maximality* means that the sequence is either infinite, or it is finite and no action of G is enabled in the final global state.

In the sequel we consider the system can start in any configuration. That is, the local state of a node can be corrupted. Note that we don't make any assumption on the bound of corrupted nodes. In the worst case all the nodes in the system may start in a corrupted configuration. In order to tackle these faults we use self-stabilization techniques.

[6] Observe that the two self-stabilizing MST algorithms mentioned in the Previous Work section assume that the nodes have distinct IDs with no distinguished nodes. Nevertheless, if the nodes have distinct IDs then it is possible to elect one node as a leader in a self-stabilizing manner. Conversely, if there exists one distinguished node in an anonymous network, then it is possible to assign distinct IDs to the nodes in a self-stabilizing manner [2]. Note that it is not possible to compute deterministically a MST in a fully anonymous network (i.e., without any distinguished node), as proved in [22].

Definition 1 (self-stabilization). *Let $\mathcal{L}_{\mathcal{A}}$ be a non-empty* legitimacy predicate[7] *of an algorithm $\mathcal{A}$ with respect to a specification predicate Spec such that every configuration satisfying $\mathcal{L}_{\mathcal{A}}$ satisfies Spec. Algorithm $\mathcal{A}$ is* self-stabilizing *with respect to Spec iff the following two conditions hold:*
*(i) Every computation of $\mathcal{A}$ starting from a configuration satisfying $\mathcal{L}_{\mathcal{A}}$ preserves $\mathcal{L}_{\mathcal{A}}$ (*closure*).*
*(ii) Every computation of $\mathcal{A}$ starting from an arbitrary configuration contains a configuration that satisfies $\mathcal{L}_{\mathcal{A}}$ (*convergence*).*

We define bellow a *loop-free* configuration of a system as a configuration which contains paths with no cycle between any couple of nodes in the system.

Definition 2 (Loop-Free Configuration). *Let $Cycle(u, v)$ be the following predicate defined for two nodes u, v on configuration C, with $P(u, v)$ a path from u to v described by C:*

$$Cycle(u, v) \equiv \exists P(u, v), P(v, u) : P(u, v) \cap P(v, u) = \emptyset.$$

A loop-free configuration is a configuration of the system which satisfies $\forall u, v : Cycle(u, v) = false$.

We use the definition of a loop-free configuration to define a *loop-free stabilizing* system.

Definition 3 (Loop-Free Stabilization). *A distributed system is called loop-free stabilizing if and only if it is self-stabilizing and there exists a non-empty set of configurations such that the following conditions hold: (i) Every execution starting from a loop-free configuration reaches a loop-free configuration (closure). (ii) Every execution starting from an arbitrary configuration contains a loop-free configuration (convergence).*

Definition 4 (Super-stabilization [19]). *A protocol P is super-stabilizing with respect to a class of topology change event Λ iff the following two conditions hold:*
(i) P is self-stabilizing and (ii) for every computation beginning at a legitimate configuration and containing a single topology change events of type Λ, a passage predicate holds.

In the sequel we study the problem of constructing a spanning tree optimizing a desired metric in self-stabilizing manner, while guaranteeing the loop-free and super-stabilizing properties.

3 Super-stabilizing Loop-Free BFS

In this section, we describe the extension of the self-stabilizing loop-free algorithm proposed in [16] to dynamic networks. Furthermore, we disscuss the

[7] A legitimacy predicate is defined over the configurations of a system and is an indicator of its correct behavior.

super-stabilization of new algorithm. Interestingly, our algorithm preserves the loop-free property without any degradation of the time complexity of the original solution.

3.1 Algorithm description

Algorithm **Dynamic-LoopFree-BFS** constructs a BFS tree and guarantees the loop-free property for dynamic networks. That is, when topological changes arise in the network (add or deletion of nodes or edges) the algorithm maintains a BFS tree without creating a cycle in the spanning tree. To this end, each node has two states: *Neutral*, noted N, and *Propagate*, noted P. A node in state N can safely select as parent its neighbor with the smallest distance (in hops) from the root without creating a cycle. A node in state P has an incoherent state according to its parent in the spanning tree. In this case, the node must not select a new parent otherwise a cycle can be created. So, this node has to inform first its descendants in the tree that an incoherency in the BFS tree was detected. Then, it corrects when all its subtrees have recovered a coherent state. Therefore, a node v in state P initiates a propagation of information with feedback in its subtree. When the propagation is finished the nodes in the subtree v (including v) recovers a correct distance and the state N.

We consider a particular node r which acts as the root of the BFS tree in the network. Every node executes the same algorithm, except the root which uses only Rule $\mathsf{R}_{\mathsf{InitRoot}}$ to correct its state. In a correct state, the root r of the BFS tree has no parent, a zero level and the state N. Otherwise, Rule $\mathsf{R}_{\mathsf{InitRoot}}$ is executed by r to correct its state.

The other five rules are executed by the other nodes of the network.

Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ is used by a node v with the state N if it detects a better parent, i.e., a neighbor node with a lower level than the level of its actual parent. In this case, v can execute this rule to update its state in order to select a new parent without creating a cycle in the tree.

If a node v has the best parent in its neighborhood but an incoherent level according to its parent, then v executes Rule $\mathsf{R}_{\mathsf{Level}++}$ to change its status to P and to initiate a propagation of information with feedback which aims to inform its descendants of its new correct level. A descendant x of node v with state N with a parent in state P executes Rule $\mathsf{R}_{\mathsf{Level}++}$ to continue the propagation and to take into account its new level.

When a leaf node x, descendant of v in Status P is reached, x stops the propagation by executing Rule $\mathsf{R}_{\mathsf{EndPropag}}$ to change its state to N and to obtain its correct level. The end of propagation is pull up in the tree using Rule $\mathsf{R}_{\mathsf{EndPropag}}$.

Rule $\mathsf{R}_{\mathsf{LevelCorrect}}$ corrects at node v the variable used to propagate the new level in the tree (variable $\mathsf{NewLevel}_v$) if this variable is lower than the actual level of v.

Rule $\mathsf{R}_{\mathsf{Dynamic}}$ deals with the dynamism of the network. This rule is executed by a node v when it detects that its parent is no more in the network and it cannot select with Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ a new parent because of its level (otherwise it may create a cycle). The aim of this rule is to increase the level of node v

using propagations of information as with Rule $\mathsf{R}_{\mathsf{Level++}}$, until v's level allows v to select a neighbor as its new parent without creating a cycle.

Figure 2 illustrates the mechanic of Rule $\mathsf{R}_{\mathsf{Dynamic}}$. In Figure 2(a) is depicted a part of the constructed BFS tree before the deletion of the node of level 2. After the deletion of this node, the node v with level 3 executes Rule $\mathsf{R}_{\mathsf{Dynamic}}$ to increase its level (equal to the lowest neighbor level plus one) in order to recover a new parent. Figure 2(b) shows the new level of v and the new levels v's descendants when the first propagation is ended. However, a level of 5 is not sufficient to allow v to select a new parent, so a second propagation is started by v which affects the levels given by Figure 2(c). Note that a descendant of v can leave v's subtree to obtain a better level if possible, this can be observed in Figure 2(c). Finally, v reaches a state with a level which allows v to execute Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ to select its new parent, and v's descendants execute Rule $\mathsf{R}_{\mathsf{Level++}}$ to correct their levels according to v's level. Figure 2(d) shows the new levels computed by the nodes.

Detailed level description. In the following, we describe the variables, the predicates and the rules used by Algorithm **Dynamic-LoopFree-BFS**.

Variables: For any node $v \in V(G)$, we denote by $N(v)$ the set of all neighbors of v in G and by $\mathcal{D}_v$ the set of sons of v in the tree. We use the following notations:

- p_v: the parent of node v in the current spanning tree;
- status_v: the status of node v, P when v is in a propagation phase, N otherwise;
- level_v: the number of edges from v to the root r in the current spanning tree;
- $\mathsf{NewLevel}_v$: the new level in the current spanning tree (used to propagate the new level).

$$\widehat{level_v} \equiv \begin{cases} \min\{\mathsf{level}_u + 1 : u \in N(v)\} & \text{if } v \neq r \\ 0 & \text{otherwise} \end{cases}$$

$$Min_v \equiv \min\{u : u \in N(v) \wedge \mathsf{level}_u = \widehat{level_v} - 1 \wedge \mathsf{status}_u = N\}$$

$$\widehat{parent_v} \equiv \begin{cases} Min_u & \text{if } \exists u \in N(v), \mathsf{level}_u = \widehat{level_v} - 1 \wedge \mathsf{status}_u = N \\ \perp & \text{otherwise} \end{cases}$$

$$\mathcal{D}_v \equiv \{u : u \in N(v) \wedge \mathsf{p}_u = v \wedge \mathsf{level}_u > \mathsf{level}_v\}$$

$$ubl_v \equiv \begin{cases} \min\{\mathsf{level}_u - 1 : u \in \mathcal{D}_v\} & \text{if } \mathcal{D}_v \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

$$\mathbf{Propag_{End}}(v) \equiv (\forall u \in \mathcal{D}_v, \mathsf{status}_u = N)$$

$$\mathbf{P_{Change}}(v) \equiv (\widehat{level_v} < \mathsf{level}_v \vee (\mathsf{level}_v = \widehat{level_v} \wedge \mathsf{p}_v \neq \widehat{parent_v})) \wedge \widehat{parent_v} \neq \perp$$

$$\mathbf{Level_{up}}(v) \equiv \mathsf{level}_v \neq \mathsf{level}_{\mathsf{p}_v} + 1 \vee (\mathsf{status}_{\mathsf{p}_v} = P \wedge \mathsf{level}_v \neq \mathsf{NewLevel}_{\mathsf{p}_v} + 1)$$

Fig. 1. Predicates used by the algorithm.

The root of the tree executes only the first rule, named $\mathsf{R}_{\mathsf{InitRoot}}$, while the other nodes execute the five last rules.

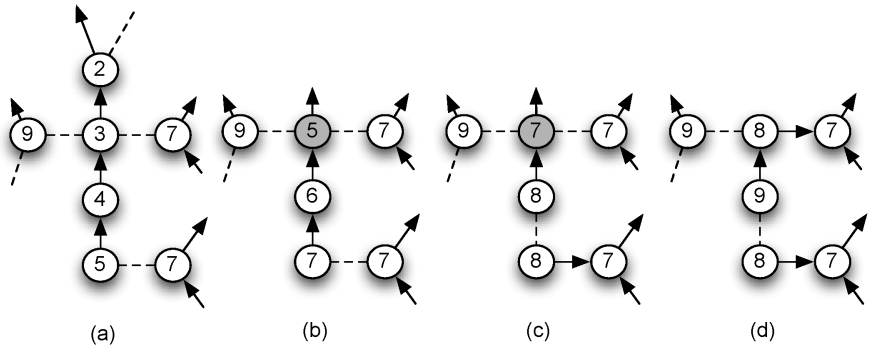

Fig. 2. Correction of the BFS tree after a node deletion.

$\mathsf{R}_{\mathsf{InitRoot}}$: **(Root Rule)**
if $v = r \wedge (\mathsf{p}_v \neq \perp \vee \mathsf{level}_v \neq 0 \vee \mathsf{NewLevel}_v \neq 0 \vee \mathsf{status}_v \neq N)$
then $\mathsf{p}_v := \perp; \mathsf{level}_v := 0; \mathsf{NewLevel}_v := 0; \mathsf{status}_v := N;$

$\mathsf{R}_{\mathsf{SafeChangeP}}$: **(Safe parent change Rule)**
if $v \neq r \wedge \mathsf{status}_v = N \wedge \mathbf{P_{Change}}(v)$
then $\mathsf{level}_v := \widehat{level_v}; \mathsf{NewLevel}_v := \mathsf{level}_v; \mathsf{p}_v := \widehat{parent_v};$

$\mathsf{R}_{\mathsf{Level++}}$: **(Increment level Rule)**
if $v \neq r \wedge \mathsf{status}_v = N \wedge \mathsf{p}_v \in N(v) \wedge \neg\mathbf{P_{Change}}(v) \wedge \mathbf{Level_{up}}(v)$
then $\mathsf{status}_v := P; \mathsf{NewLevel}_v := \mathsf{NewLevel}_{\mathsf{p}_v} + 1;$

$\mathsf{R}_{\mathsf{EndPropag}}$: **(End of propagation Rule)**
if $v \neq r \wedge \mathsf{status}_v = P \wedge \mathbf{Propag_{End}}(v) \wedge ubl_v \geq \mathsf{NewLevel}_v$
then $\mathsf{status}_v := N; \mathsf{level}_v := \mathsf{NewLevel}_v;$

$\mathsf{R}_{\mathsf{LevelCorrect}}$: **(Level correction Rule)**
if $v \neq r \wedge \mathsf{NewLevel}_v < \mathsf{level}_v$ **then** $\mathsf{NewLevel}_v := \mathsf{level}_v;$

$\mathsf{R}_{\mathsf{Dynamic}}$: **(Increment level Rule for dynamic networks)**
if $v \neq r \wedge \mathsf{status}_v = N \wedge \mathsf{p}_v \notin N(v) \wedge \neg\mathbf{P_{Change}}(v)$
then $\mathsf{status}_v := P; \mathsf{NewLevel}_v := \widehat{level_v};$

3.2 Correctness proof

The algorithm proposed in the precedent subsection extends the algorithm of [16] to dynamic network topologies. When the system is static the correctness of the algorithm directly follows from the results proven in [23]. In the following, we focus only the case of dynamic topologies, i.e., when nodes/edges of the tree fails or nodes/edges are added in the network. Note that in the following, we only

study the case of an edge failure. A node failure produces the same consequences, i.e., the spanning tree is splitted and some nodes have no parent. Moreover, we do not consider edges out of the tree because this does not lead the system in an illegitimate configuration. After each fail of node or edge in the tree, we assume the uderlying network is always connected.

In [23], a legitimate configuration for the algorithm is defined by the following predicate satisfied by every node $v \in V$: $\Pr_v^{\mathcal{LP}} \equiv [(v = r) \wedge (\mathsf{level}_v = 0) \wedge (\mathsf{status}_v = N)] \vee [(v \neq r) \wedge (\mathsf{level}_v = \dot{\mathsf{level}}_v) \wedge (\mathsf{status}_v = N) \wedge (\mathsf{level}_v = \mathsf{level}_{\mathsf{p}_v} + 1)]$, with $\forall v \neq r, \dot{\mathsf{level}}_v = \min\{\dot{\mathsf{level}}_u + 1 : u \in N(v)\}$ defines the optimal level of node v.

Note that after a fail of an edge of the tree T, Predicate $\Pr_v^{\mathcal{LP}}$ is not satisfied anymore. The tree T splits in a forest F which contains the subtrees of T. Let $Orph$ be the set of nodes v such that $\mathsf{p}_v \notin N(v)$, note that $r \notin Orph$. The following predicate is satisfied by every node $v \in V, v \notin Orph$

$$\Pr_v^{\mathcal{LC}} \equiv \begin{cases} \mathsf{level}_{\mathsf{p}_v} + 1 \leq \mathsf{level}_v \wedge \mathsf{NewLevel}_v \geq \mathsf{level}_v & \text{if } v \neq r \\ \mathsf{level}_r = 0 \wedge \mathsf{status}_r = N & \text{otherwise} \end{cases}$$

We show below that each node with no parent in F starts a propagation of information in its subtree.

Lemma 1. *Let a node $v \in V, v \in Orph$. If $\mathsf{status}_v = N$ and $\mathbf{P_{Change}}(v) = false$ then status v eventually moves to P.*

Proof. Let $v \in V, v \in Orph$ be a node such that $\mathsf{status}_v = N$ and $\mathbf{P_{Change}}(v) = false$. v can only execute Rules $\mathsf{R_{LevelCorrect}}$ or $\mathsf{R_{Dynamic}}$, because v can not execute Rules $\mathsf{R_{SafeChangeP}}$, $\mathsf{R_{Level++}}$ and $\mathsf{R_{EndPropag}}$ since $\mathsf{status}_v = N, \mathbf{P_{Change}}(v) = false$ and $\mathsf{p}_v \notin N(v)$. To change its status from N to P, a node $v \in Orph$ must execute Rule $\mathsf{R_{Dynamic}}$. Suppose that v does not execute Rule $\mathsf{R_{Dynamic}}$. So v can only execute Rule $\mathsf{R_{LevelCorrect}}$. However, after execution of Rule $\mathsf{R_{LevelCorrect}}$ we have $\mathsf{NewLevel}_v := \mathsf{level}_v$ and the guard of Rule $\mathsf{R_{LevelCorrect}}$ is no more satisfied. Thus, only the guard of Rule $\mathsf{R_{Dynamic}}$ is satisfied and v remains enabled until it performs Rule $\mathsf{R_{Dynamic}}$. Therefore, the scheduler eventually selects v to perform Rule $\mathsf{R_{Dynamic}}$. □

According to Lemma 9 in [23], a node v such that $\mathsf{status}_v = P$ eventually performs Rule $\mathsf{R_{EndPropag}}$ to change its status to N. In the following, we show that a node in $Orph$ (i.e., without a parent in its neighborhood) eventually leaves the set $Orph$.

Lemma 2. *Let $v \in V, v \in Orph$. Eventually, v is not anymore in the set Orph and selects a parent without creating a cycle.*

Proof. We show the lemma by induction on the height of the subtree of v. Consider the case where a node $v \in Orph$ has a neighbor $u \in N(v)$ such that $\mathsf{level}_u < \mathsf{level}_v$. We assume that for every node x in F, $x \notin Orph$, we have $\mathsf{level}_{p_x} + 1 \leq \mathsf{level}_x$. So, u can not be a descendant of v. Thus, v performs Rule $\mathsf{R_{SafeChangeP}}$ to choose u as its parent without creating any cycle in F. Otherwise,

every node $u \in N(v)$ is a child of v. According to Lemma 9 in [23] and Lemma 1 (above), the level of every node in the subtree of v increases. Since we assume the network is always connected, there exists a leaf node x in the subtree of v such that $\mathsf{level}_x > \dot{\mathsf{level}}_x = \mathsf{level}_y$, with $y \in N(x)$. Thus, x can execute Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ to choose y as its parent and x leaves the subtree of v. Since the height of the subtree of v is finite, eventually v can choose a neighbor u as its parent because u is no more in the subtree of v. Therefore, in a finite time a node $v \in \mathit{Orph}$ leaves the set Orph by selecting a parent in its neighborhood without creating a cycle. □

According to Lemma 2, each node has a parent and no cycle is created. Thus, the system reaches a configuration where a spanning tree is constructed. So the analysis given in [23] can be used to show that the system reaches a configuration in which for each node $v \in V$ we have $\mathsf{level}_v = \dot{\mathsf{level}}_v$. Since the initial configuration contains a spanning tree, the algorithm stabilizes to a breadth first search tree and during the stabilization of the algorithm the loop-free property is maintained, as showed in [23].

Above we consider only the fail of nodes/edges of the tree, now we discuss the add of nodes and edges in the network. In a legitimate configuration, after the add of an edge every node $v \in V$ always satisfies $\mathsf{level}_v \geq \dot{\mathsf{level}}_v$. According to Lemma 12 and Corollary 1 in [23], in a finite time eventually for every node $v \in V$ we have $\mathsf{level}_v = \dot{\mathsf{level}}_v$. In a legitimate configuration, after the add of a node v Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ is executed by v to select a neighbor $u \in N(v)$ as its parent, there exists such a node u because we assume that the network is always connected. Therefore, the system is in an arbitrary configuration where a spanning tree is constructed. Therefore, the analysis given in [23] can be used to show that in a finite time for every node $v \in V$ we have $\mathsf{level}_v = \dot{\mathsf{level}}_v$. Moreover, in the case of node/edge adds the initial configuration contains a spanning tree, thus the loop-free property is maintained by the algorithm.

In the following, we prove that the presented algorithm has a superstabilizing property for a particular class of topology change events. We show that a passage predicate is satisfied during the restabilizing execution of our algorithm. We define the considered topology change events, noted ε:

- an add (resp. a removal) of an edge (u, v) in the network noted $\mathtt{recov}_{uv}$ (resp. $\mathtt{crash}_{uv}$);
- an add (resp. a removal) of a neighbor node u of v in the network noted $\mathtt{recov}_u$ (resp. $\mathtt{crash}_u$).

In the sequel, we suppose that after every topology change event the network remains connected. We provide below definitions of the topology change events class Λ and passage predicate.

Definition 5 (Class Λ of topology change events). $\mathtt{crash}_{uv}$ *and* $\mathtt{crash}_v$ *compose the class* Λ *of topology change events.*

Definition 6 (Passage predicate). *The parent of a node v can be modified if v is in the subtree connected by the removed edge or node, and the parent is not changed for any other node in the tree.*

Lemma 3. *The proposed protocol is superstabilizing for the class Λ of topology change events, and the passage predicate (Definition 6) continues to be satisfied while a legitimate configuration is reached.*

Proof. Consider a legitimate configuration Δ. Suppose ε is a removal of edge (u, v) from the network. If (u, v) is not a tree edge then the levels of u and v are not modified and neither u nor v changes its parent, thus no parent variable is modified. Otherwise, let $\mathsf{p}_v = u$, u's level and u's parent are not modified, it is true for any other node x not contained in the subtree of v since the distance between x and the root r in the graph is not modified (i.e., Predicate $\mathbf{P_{Change}}(x)$ is not satisfied). However, u is no more a neighbor of v so according to Lemma 1 v executes Rule $\mathsf{R_{Dynamic}}$ and starts a propagation phase. Moreover, according to Lemma 2 v selects a new parent without creating a cycle. Therefore, only a node in the subtree connected by the edge (u, v) may change its parent.

Suppose ε is a removal of node u from the network. Any node x not contained in the subtree of u do not change its parent relation because the distance between x and the root node r is not modified (i.e., Predicate $\mathbf{P_{Change}}(x)$ is not satisfied). Consider each edge (u, v) between u and its child v, we can apply the same argument described above for an edge removal. So only any node contained in the subtree connected by u may change its parent. □

3.3 Complexity analysis

In the following we focus the complexity analysis of our algorithm in both static and dynamic networks. Note that the original algorithm proposed in [16] had no complexity analysis. Interestingly, we prove that our extension has a zero time extra-cost with respect to the original solution.

Lemma 4. *Starting from an arbitrary configuration, in at most $O(n^2)$ rounds a breadth first search tree is constructed by the algorithm in a static network.*

Proof. To construct a spanning tree, the algorithm must remove all the cycles present in the starting configuration. So, we first analyze the number of rounds needed to remove a cycle.

To remove a cycle, a node of the cycle must change its parent to select a node out of the cycle, such a node is named a *break node*. A node can change its parent using Rule $\mathsf{R_{SafeChangeP}}$, but a break node executes Rule $\mathsf{R_{SafeChangeP}}$ if the level of the new parent (out of the cycle) is lower than the level of the break node. Consider a break node x and the neighbor y of x which must be selected as the new parent of x. We note L_x and L_y the level of x and y respectively. To select y as its new parent and to break the cycle, x must have its level L_x such that $L_y < L_x$. In the cycle, a node corrects its level according to its parent by initiating a propagation of information with Rules $\mathsf{R_{Level++}}$ and $\mathsf{R_{EndPropag}}$. Thus

the number of increments until we have $L_y < L_x$ is equal to $\lceil \frac{(L_x+1)-L_y}{|C|} \rceil$, with $|C|$ the size of the cycle C to break. The propagation of information is in order of the size of C. Thus, $O((L_x + 1) - L_y)$ rounds are needed to have $L_y < L_x$. Since we want to construct a breadth first search tree the level of a node cannot exceed n, with n the size of the network. Thus, we consider that the level of a node is encoded using $\log n$ bits. The biggest value for $(L_x + 1) - L_y$ is obtained when $L_y = 1$ and therefore we have $(L_x + 1) - L_y \leq n$.

Since the maximum number of possible cycles of a network is no more than $n/2$, obtained with cycles of size 2, we have that in $O(n^2)$ all cycles are removed in the network and a spanning tree is constructed. In at most $O(D)$ additional rounds a breadth first tree is constructed, with D the diameter of the network. Indeed, no cycle is created by the algorithm until reaching a legitimate configuration, since the algorithm guarantee the loop-free property. □

Lemma 5. *Starting from an arbitrary configuration, in at most $O(n^2)$ rounds a breadth first search tree is constructed by the algorithm in a dynamic network.*

Proof. In a dynamic network, for a node we can have the case where the edge leading to its parent or its parent is deleted from the network. When a node x detects this case, x executes Rule $\mathsf{R}_{\mathsf{Dynamic}}$ to find a new parent in the network. To accomplish this task, x starts a propagation of information to increment its level since it has an incorrect level according to its parent (which is no more in the network).

We have two cases for the new parent selected by x. The first case is that the new parent of x is a neighbor y with level L_y bigger than x's level L_x. In this case, x must increment its level to have the condition $L_y < L_x$. To obtain this condition, at most $L_y - L_x$ increments are needed, that is at most n increments since we want to construct a breadth first tree and the level of a node is encoded using $\log n$ bits. The second case is that x selects one of its children u as its new parent, but to preserve the loop-free property x can do this only when u is no more a child of x. The worst case for x is to wait that it has no more children if u is its only child, that is the subtree of x has disappeared. At most n increments are needed to have that x has an empty subtree.

In all cases, at most n increments are needed and the number of rounds for a propagation of information is in the order of the size of the subtree of x, that is at most n. Thus, in at most $O(n^2)$ rounds x finds a new parent in the network, then we can consider we are in the case of a static network and Lemma 4 can be applied. Therefore, in at most $O(n^2)$ rounds a legitimate configuration is reached by the algorithm. □

4 Super-stabilizing Loop-Free transformation scheme

Our objective is to design a generic scheme for the construction of spanning trees considering any metric (not only metrics based on distances in the graph) with loop-free and super-stabilizing properties. The idea is to extend an existing self-stabilizing spanning tree optimized for a given metric (e.g. MST, maximum

degree spanning tree, max-flow tree etc) with super-stabilizing and loop-free properties via the composition with a spanning tree construction that already satisfies these properties. Assume $\mathcal{M}$ be the predicate that captures the properties of the metric to be optimized. Consider $\mathcal{A}$ the algorithm that outputs a self-stabilizing spanning tree and verifies $\mathcal{M}$. That is, given a graph, $\mathcal{A}$ computes the set of edges $\mathcal{S}_{\mathcal{A}}$ that satisfies $\mathcal{M}$ and is a spanning tree. Consider Algorithm $\mathcal{B}$ an algorithm that outputs a super-stabilizing and loop-free spanning tree $\mathcal{S}_{\mathcal{B}}$. Ideally, if all edges in $\mathcal{S}_{\mathcal{A}}$ are included in $\mathcal{S}_{\mathcal{B}}$ then there is no need for further transformations. However, in most of the cases the two trees are not identical. Therefore, the idea of our methodology is very simple. Algorithms $\mathcal{A}$ and $\mathcal{B}$ run such that the output of $\mathcal{A}$ defines the graph input for $\mathcal{B}$. That is, the neighborhood relation used by $\mathcal{B}$ is the initial graph filtered by $\mathcal{A}$ to satisfy the predicate $\mathcal{M}$. The principal of this composition is already known in the literature as fair composition [24]. In our case the "slave" protocol is protocol $\mathcal{A}$ that outputs the set of edges input for the "master" protocol $\mathcal{B}$.

The following lemma direct consequence of the results proven in [24] guaranties the correctness of the composition.

Lemma 6. *Let $\mathcal{M}$ be the predicate that captures the properties of the metric to be optimized. Let $\mathcal{A}$ be an algorithm that outputs a self-stabilizing spanning tree that satisfies $\mathcal{M}$, $\mathcal{S}_{\mathcal{A}}$. Let $\mathcal{B}$ be a loop-free protocol that computes a spanning tree on the topology defined by $\mathcal{S}_{\mathcal{A}}$ and super-stabilizing for a class of topology changes Λ. The fair composition of $\mathcal{A}$ and $\mathcal{B}$ is a protocol that outputs a loop-free spanning tree that satisfies $\mathcal{M}$ and is super-stabilizing for Λ.*

Note that our super-stabilizing loop-free BFS can be used as protocol $\mathcal{B}$ in the above composition. The interesting property of the composition is that the time complexity will be maximum between $O(n^2)$ and the complexity time of the candidate to be transformed. Note that so far, the best time complexity of a spanning tree optimized for a given metric is $O(n^2)$ which leads to the conclusion that the composition does not alterate the time complexity of the candidate.

In the following, we specify the predicate $\mathcal{M}$ for two well known problems: max-flow trees and minimum degree spanning trees.

Case study 1: Maximum-flow tree The problem of constructing a maximum-flow tree from a given root node r can be stated as follows. Given a weighted undirected graph $G = (V, E, w)$, the goal is to construct a spanning tree $T = (V, E_T)$ rooted at r, such that for every node $v \in V$ the path between r and v has the maximum flow. Formally, let $fw(v) = \min(fw(\mathsf{p}_v), w(\mathsf{p}_v, v))$ the flow for every node $v \in V$ in tree $\mathcal{T}$ and mfw_v the maximum flow value of v among all spanning trees of G rooted at r. The maximum-flow tree problem is to compute a spanning tree T, such that $\forall v \in V, fw(v) = mfw_v$. The max flow tree problem has been studied *e.g.* in [21]. In this case, the graph $G_{\mathcal{S}_{\mathcal{A}}} = (V_{\mathcal{S}_{\mathcal{A}}}, \mathcal{S}_{\mathcal{A}})$ for the maximum-flow tree problem must satisfies the following predicate:

$$\mathcal{M} \equiv (|\mathcal{S}_{\mathcal{A}}| = n-1) \wedge (V = V_{\mathcal{S}_{\mathcal{A}}}) \wedge (\forall v \in V, fw(v) = \max\{\min(fw(u), w(u,v)) : u \in N(v)\}).$$

Case study 2: Minimum degree spanning tree Given an undirected graph $G = (V, E)$ with $|V| = n$, the minimum degree spanning tree problem is to construct a spanning tree $T = (V, E_T)$, such that the maximum degree of T is minimum among all spanning trees of G. Formally, let $deg_{\mathcal{T}}(v)$ the degree of node $v \in V$ in the subgraph $\mathcal{T}$ and $deg(\mathcal{T})$ the maximum degree of subgraph $\mathcal{T}$ (i.e., $deg(\mathcal{T}) = \max\{deg_{\mathcal{T}}(v) : v \in V\}$). The minimum spanning tree problem is to compute a spanning tree T, such that $deg(T) = \min\{deg(T') : T' \text{ is a spanning tree of } G\}$. A self-stabilizing solution for the minimum degree spanning tree algorithm has been proposed in [25]. If this solution plays the slave master in our transformation scheme then the graph $G_{\mathcal{S}_\mathcal{A}} = (V_{\mathcal{S}_\mathcal{A}}, \mathcal{S}_\mathcal{A})$ input for the BFS algorithm satisfy the following predicate:

$$\mathcal{M} \equiv (|\mathcal{S}_\mathcal{A}| = n-1) \wedge (V = V_{\mathcal{S}_\mathcal{A}}) \wedge deg(G_{\mathcal{S}_\mathcal{A}}) = \min\{deg(T') : T' \text{ a spanning tree of } G\}.$$

5 Concluding remarks

We presented a scheme for constructing loop-free and super-stabilizing protocol for universal tree metrics, without significant impact on the performance. There are several open questions raised by our work:

1. Decoupling various added properties (such as loop-freedom or super-stabilization) seems desirable. As a particular network setting may not need both properties and/or temporarily run in conditions where the network is essentially static, some complexity cost could be saved by removing uneeded properties. Of course, stripping our scheme can trivially result in a generic loop-free transformer or to a generic super-stabilizing transformer. Yet, modular design of features, as well as further enhancements (such as safe convergence [26,27]), seems an interesting path for future research.
2. The implementation of self-stabilizing protocols recently was helped by compilers that take as input guarded commands and provide as output actual source code for existing devices [28]. Transformers such as this one would typically benefit programmers' toolboxes as they ease the reasoning by keeping the source code intricacies at a very high level. Actual implementation of our transformer into a programmer's toolbox is a challenging ingeneering task.

References

1. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM **17**(11) (1974) 643–644
2. Dolev, S.: Self-Stabilization. MIT Press (2000)
3. Tixeuil, S.: Self-stabilizing Algorithms. Chapman & Hall/CRC Applied Algorithms and Data Structures. In: Algorithms and Theory of Computation Handbook, Second Edition. CRC Press, Taylor & Francis Group (November 2009) 26.1–26.45
4. Gopal, A.S., Perry, K.J.: Unifying self-stabilization and fault-tolerance (preliminary version). In: PODC. (1993) 195–206

5. Anagnostou, E., Hadzilacos, V.: Tolerating transient and permanent failures (extended abstract). In: WDAG. (1993) 174–188
6. Dolev, S., Welch, J.L.: Wait-free clock synchronization. Algorithmica **18**(4) (1997) 486–511
7. Papatriantafilou, M., Tsigas, P.: On self-stabilizing wait-free clock synchronization. Parallel Processing Letters **7**(3) (1997) 321–328
8. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. J. ACM **51**(5) (2004) 780–799
9. Ben-Or, M., Dolev, D., Hoch, E.N.: Fast self-stabilizing byzantine tolerant digital clock synchronization. In: PODC. (2008) 385–394
10. Masuzawa, T., Tixeuil, S.: Bounding the impact of unbounded attacks in stabilization. In: SSS. (2006) 440–453
11. Masuzawa, T., Tixeuil, S.: Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. International Journal of Principles and Applications of Information Science and Technology (PAIST) **1**(1) (December 2007) 1–13
12. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci. **1997** (1997)
13. Herman, T.: Superstabilizing mutual exclusion. Distributed Computing **13**(1) (2000) 1–17
14. Katayama, Y., Ueda, E., Fujiwara, H., Masuzawa, T.: A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. J. Parallel Distrib. Comput. **62**(5) (2002) 865–884
15. Cobb, J.A., Gouda, M.G.: Stabilization of general loop-free routing. J. Parallel Distrib. Comput. **62**(5) (2002) 922–944
16. Johnen, C., Tixeuil, S.: Route preserving stabilization. In: Self-Stabilizing Systems. (2003) 184–198
17. Garcia-Luna-Aceves, J.J.: Loop-free routing using diffusing computations. IEEE/ACM Trans. Netw. **1**(1) (1993) 130–141
18. Gafni, E.M., Bertsekas, P.: Distributed algorithms for generating loop-free routes in networks with frequently changing topology. IEEE Transactions on Communications **29** (1981) 11–18
19. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci. **1997** (1997)
20. Blin, L., Potop-Butucaru, M., Rovedakis, S., Tixeuil, S.: A new self-stabilizing minimum spanning tree construction with loop-free property. In: DISC. (2009) 407–422
21. Gouda, M.G., Schneider, M.: Stabilization of maximal metric trees. In: WSS. (1999) 10–17
22. Gupta, S.K.S., Srimani, P.K.: Self-stabilizing multicast protocols for ad hoc networks. J. Parallel Distrib. Comput. **63**(1) (2003) 87–96
23. Johnen, C., Tixeuil, S.: Route preserving stabilization. Technical Report 1353, LRI, Université Paris-Sud XI (2003)
24. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. Distributed Computing **7**(1) (1993) 3–16
25. Blin, L., Potop-Butucaru, M.G., Rovedakis, S.: Self-stabilizing minimum-degree spanning tree within one from the optimal degree. In: IPDPS. (2009) 1–11
26. Kakugawa, H., Masuzawa, T.: A self-stabilizing minimal dominating set algorithm with safe convergence. In: IPDPS. (2006)
27. Kamei, S., Kakugawa, H.: A self-stabilizing approximation for the minimum connected dominating set with safe convergence. In: OPODIS. (2008) 496–511

28. Dalton, A.R., McCartney, W.P., Dastidar, K.G., Hallstrom, J.O., Sridhar, N., Herman, T., Leal, W., Arora, A., Gouda, M.G.: Desal alpha: An implementation of the dynamic embedded sensor-actuator language. In: ICCCN, IEEE (2008) 541–547