The First Fully Polynomial Stabilizing Algorithm for BFS Tree Construction

Alain Cournier, Stephane Rovedakis, Vincent Villain

Version of Record: https://www.sciencedirect.com/science/article/pii/S0890540119300069
Manuscript_9cd09e30122cb7ab65c54415c738a853

The First Fully Polynomial Stabilizing Algorithm for BFS Tree Construction☆,☆☆

Alain Cournier[a], Stéphane Rovedakis[b,*], Vincent Villain[a]

[a]*MIS Lab., Université Picardie Jules Verne, 33 Rue St Leu, 80039 Amiens Cedex 1, France*
[b]*CEDRIC Lab., CNAM, 292 Rue St Martin, 75141 Paris Cedex 03, France*

Abstract

The construction of a spanning tree is a fundamental task in distributed systems which allows to resolve other tasks (i.e., routing, mutual exclusion, network reset). In this paper, we are interested in the problem of constructing a *Breadth First Search* (BFS) tree. *Stabilization* is a versatile technique which ensures that the system recovers a correct behavior from an arbitrary global state resulting from transient faults.

A *fully polynomial* algorithm has a round complexity in $O(d^a)$ and a step complexity in $O(n^b)$ where d and n are the diameter and the number of nodes of the network and a and b are constants. We present the first fully polynomial stabilizing algorithm constructing a BFS tree under a distributed daemon. Moreover, as far as we know, it is also the first fully polynomial stabilizing algorithm for spanning tree construction. Its round complexity is in $\Theta(d^2)$ and its step complexity is in $O(n^6)$.

To our knowledge, since in general the diameter of a network is much smaller than the number of nodes ($\log(n)$ in average instead of n [1]), this algorithm reaches the best compromise of the literature between the complexities in terms of rounds and in terms of steps.

Keywords: Distributed systems, Fault-tolerance, Stabilization, Spanning tree construction.

1. Introduction

The computation of a solution to a given problem in a distributed system requires the collaboration of a part or the whole system. To this end, some amount of information have to be exchanged between the elements of the system involved in the computation. One important aspect in the design of distributed algorithms is to solve a problem in an efficient way. In the context of distributed fault-tolerance, the communication and time complexities to solve a given problem by a distributed algorithm are important measures, expressed respectively in terms of number of exchanged messages and *rounds*. Indeed, every execution of an algorithm is decomposed into rounds, where a *round* contains a single computation *step* of each processor of

☆A preliminary version of this paper has been published in OPODIS [2].

☆☆This work has been partially supported by the ANR project SPADES (08-ANR-SEGI-025).

*Corresponding author
Email addresses: alain.cournier@u-picardie.fr (Alain Cournier), stephane.rovedakis@cnam.fr (Stéphane Rovedakis), vincent.villain@u-picardie.fr (Vincent Villain)

the system. Self-stabilization introduced first by Dijkstra in [3] and later publicized by several books [4, 5] is one of the most versatile techniques to handle transient faults arising in distributed systems. A distributed algorithm is self-stabilizing if starting from any arbitrary global state (due to faults or attacks) the system is able to recover from this catastrophic situation in finite time without external (e.g., human) intervention. To ease the design of fault-tolerant algorithms, Dijkstra introduced in [3] also an abstraction model of the message passing model, the *locally shared memory model*. In this model, each processor can directly read the variables of its neighbors in a computation *step* (or *steps*), instead of exchanging messages. This model was largely used for the design of self-stabilizing algorithms. Since the introduction of the self-stabilization paradigm, a lot of effort has been put to design self-stabilizing algorithms with a low time complexity (or *convergence time*) required to reach a correct system state. This leads to the introduction of several stabilization alternatives, in particular *Snap-stabilization* which have been proposed by Bui *et al.* [6]. This form of stabilization characterizes self-stabilizing algorithms with a convergence time of 0 (see Definition 2). As explained by Cournier *et al.* in [7], the zero convergence time does not guarantee that all components of the system never work in a fuzzy manner. A snap-stabilizing protocol ensures that any request is satisfied despite the arbitrary initial configuration, while a self-stabilizing protocol often needs to be repeated an unbounded number of times before guaranteeing the satisfaction of any request.

Although the time complexity given in (asynchronous) rounds allows to capture the execution rate of the slowest process in any execution, another crucial measure is the number of steps needed by an algorithm to compute the desired solution. In fact, the step complexity reflect more accurately the workload generated by a distributed algorithm. Moreover, one can observe that the communication complexity is deeply related to the number of steps. Indeed, this measure reflects the amount of information exchanged to compute the solution, especially in the context of stabilizing algorithms in which in a step a processor sends at least one message for incorrect local state detection in the neighborhood. Recently, reducing the communication complexity becomes a new challenge to increase the practical use of self-stabilizing algorithms. Contrary to non fault-tolerant distributed algorithms, self-stabilizing distributed algorithms may introduce an overhead on the communication complexity before and after reaching a correct global state. Indeed, this is due to two facts: (i) self-stabilizing algorithms can start from an arbitrary state (not only from a defined initial state) of the system and converge to a correct global state, and (ii) they have to perpetually exchange information to always allow the correction of the nodes' state (the latter fact is inherent to any adaptive approach). Devismes *et al.* [8] and Masuzawa *et al.* [9, 10] study the communication efficiency of self-stabilizing algorithms after the convergence to a correct state of the system. The goal is to determine if it is possible to design self-stabilizing algorithms which work correctly while avoiding the communication between pairs of neighbors in the system. To this end, they introduced several complexity measures to analyze the communication efficiency of a protocol, among them k-*efficiency* and $\diamond$-k-*stable*. The first measure establishes an upper bound k on the number of neighbors with which each node communicates at each step. Note that every distributed self-stabilizing algorithm is obviously at most Δ-*efficient* since each processor communicates with all its neighbors [8]. The second complexity measures is the same as the former but focusing only on computation steps after a correct system state is reached. The authors consider the *maximal independent set*, the *maximal matching*, the *spanning tree construction* and the *minimal connected dominating set* problem. They present self-stabilizing distributed algorithms achieving better communication complexities than existing algorithms, otherwise impossibility results are given. Kutten *et al.* [11] addressed also the problem of reducing the communication complexity of distributed self-stabilizing algorithms. Self-stabilizing

algorithms with low communication are proposed for several tasks: *spanning tree construction*, *distributed reset* and *unison*. The given algorithms have a low communication complexity during the whole algorithm's execution, i.e., before and after reaching a correct system state. To achieve low communication, randomized algorithms are designed (instead of deterministic ones) allowing to by-pass the constraint on the number of communicating neighbors. In fact, a reduction on the number of communicating neighbors in the case of deterministic algorithms induces an increase of the convergence time.

Contrary to the above cited works, we adopt another approach to achieve communication efficiency. In this paper, we address this problem by bounding and reducing the number of steps needed to solve a distributed problem, and we introduce a particular class of distributed algorithms, called *fully polynomial* algorithms, in the context of the locally shared memory model. These algorithms are characterized by efficient complexities to solve a distributed task, i.e., a round complexity polynomial in the network diameter and step complexity polynomial in the network size.

Definition 1 (Fully polynomial algorithms). *A distributed algorithm is* fully polynomial *if it has a round complexity in $O(d^a)$ and a step complexity in $O(n^b)$ where d and n are the diameter and the number of nodes of the network and a and b are constants.*

This class of distributed algorithms is particularly suitable in the context of large scale networks. Indeed, these systems are characterized by an ever growing size and a low diameter in average. In this context, it is of primary importance to design algorithms with a round complexity independent of the network size and a low communication complexity (e.g., achieved via a polynomial step complexity). Distributed fully polynomial algorithms allow to have an execution with a high degree of parallelism (round complexity defined by a polynomial function of the network diameter, and not dependent of the network size), while the communication complexity is bounded (step complexity defined by a polynomial function of the network size, since each processor executes at least a step). Having fault-tolerant distributed algorithms which belong to this class of algorithms is of theoretical and practical importance, since as large scale systems have a high number of processor they are highly subject to crash faults or topological changes.

So, an interesting question addressed in this paper can be the following: Do there exist fully polynomial self-stabilizing algorithms solving global distributed tasks?

Contributions. To our knowledge there exists no fully polynomial stabilizing algorithm solving global distributed tasks in the literature[1]. In this paper, we show that this class of algorithms related to global distributed tasks is not empty. To this end, we consider the global distributed task of constructing a spanning tree rooted at a designated node, due to its importance in distributed systems. As presented in the Related work section below, to our knowledge the existing self-stabilizing spanning tree constructions cannot belong to the class of algorithms defined in Definition 1. Indeed, the existing stabilizing algorithms have either a round and step complexity both polynomial in n, or a round complexity polynomial in d but an exponential or unbounded step complexity, where d and n are respectively the diameter and the size of the network. Therefore, we present the first fully polynomial stabilizing algorithm for the construction of a spanning tree with a round complexity lower than $\Theta(\max(d^2, n))$. Our algorithm computes a BFS tree in $\Theta(d^2)$ rounds with a polynomial number of steps in $O(n^6)$ (the step complexity is $O(mn^4)$ and

[1]Except the abstract version of this paper [2].

$m \ll n^2$) under a distributed daemon without any fairness assumptions (which is the weakest daemon), with m the number of edges in the network. To our knowledge, since in general the diameter of a network is much smaller than the number of nodes ($\log(n)$ in average instead of n [1]), this algorithm reaches the best compromise of the literature between the complexities in terms of rounds and in terms of steps. Moreover, the algorithm proposed in this paper is fully polynomial for any topology, i.e., we do not consider a particular network topology. Finally, this BFS tree construction is based on a snap-stabilizing algorithm given in this paper resolving a sub-problem, called the *Question-Answer* problem, in which each node requests a permission (delivered by a subset of network nodes) in order to perform a defined computation, which is of independent interest.

Related work. One basic task performed in every network is the transmission of information, which can be done by resolving the construction of a spanning tree (acyclic virtual structure with no cycle and interconnecting all the nodes of a network). In distributed systems, the construction of a spanning tree is commonly used to design algorithms resolving other distributed tasks, like routing, token circulation or message broadcasting in a network. Hence, there are a lot of works which study this task. There are many different spanning tree construction problems guaranteeing various properties, e.g., a spanning tree of minimum weight or a spanning tree of minimum diameter. A survey on several self-stabilizing tree constructions can be found in [12].

Arora and Gouda [13] are interested in designing an algorithm which allows to reset a network by resetting the state of the nodes when a fault is detected in a dynamic network. To this end, the authors present a self-stabilizing reset algorithm which constructs a BFS tree in $O(N^2)$ rounds, with N an upper bound on the number of nodes in the network. Dolev, Israeli and Moran [14] give one of the first self-stabilizing algorithms for the construction of a spanning tree. In their work, a BFS tree is used to resolve the mutual exclusion problem. Afek, Kutten, and Yung [15] have proposed independently from [14] a self-stabilizing algorithm constructing a BFS tree. This algorithm uses the node identifiers to construct a BFS tree rooted at the node of highest identifier in the network in $O(n^2)$ rounds. Moreover, it incorporates a mechanism to transmit requests and acknowledgments for the addition of new nodes in a tree. The root of a tree allows the connection of new nodes if no higher identifier is detected in the network. Chen *et al.* proposed a self-stabilizing spanning tree construction algorithm [16], which was improved later to construct a BFS tree [17]. The time complexity of these two algorithms is $\Theta(n)$ rounds. Ducourthial et al. proposed a self-stabilizing algorithm for the construction of a BFS tree which is a modification of the algorithm proposed in [17]. As analyzed in [18], this algorithm has a time complexity of $\Theta(d)$ rounds, where d is the diameter of the network. More recently, Burman and Kutten [19] give a solution to construct a Shortest Path tree in $O(d)$ rounds, extending to the message passing model a solution proposed by Awerbuch *et al.* [20]. Datta, Larmore, and Vemula [21] resolves the election problem by constructing a silent self-stabilizing BFS tree in $O(n)$ rounds. The *silent* property is to guarantee that when a *legitimate* configuration[2] is reached the values stored in the registers do not change anymore. $O(d)$ additional rounds are needed to the algorithm to become silent. There are many other works on the self-stabilizing construction of a spanning tree with additional properties, e.g., DFS tree [22, 23]. There are also works which study the construction of a spanning tree with a low memory complexity. For example, Johnen and Beauquier give a self-stabilizing token circulation allowing to construct a DFS tree using

[2]A configuration from which every execution suffix satisfies the specification of the problem to solve.

$O(\log \Delta)$ bits [24], whereas Johnen proposes a self-stabilizing algorithm for the construction of a BFS tree using $O(\Delta)$ bits [25], with Δ the maximum node degree in the network.

Table 1: Distributed stabilizing algorithms for the construction of spanning trees. n, d and Δ are respectively the number of nodes, the diameter and the maximum degree in the network, while N is an upper bound of n, D is an upper bound of d, and Max is the maximum height value in the tree of a node in the initial configuration. The *silent* property for a self-stabilizing algorithm is to guarantee that when a legitimate configuration is reached the values stored in the registers do not change anymore.

	References	Round complexity	Step complexity	Memory complexity	Silent property
BFS	[13]	$O(N^2)$	Undetermined	$O(\log(n))$	Yes
	[14]	$O(d)$	Undetermined	$O(\Delta \log(n))$	Yes
	[15]	$O(n^2)$	Undetermined	$O(\log(n))$	Yes
	[17]	$\Theta(n)$	$\Omega(2^n)^\dagger$	$O(\log(n))$	Yes
	[26]	$\Theta(d)$	$O(n(Max+d)^n)^\dagger$	$O(\log(n))$	Yes
	[20]	$O(D)$	$\Omega(2^{\frac{D}{2}})^\ddagger$	$O(\log^2(n))$	Yes
	[25]	$\Omega(d^2)$	Undetermined	$O(\log(\Delta))$	No
	[19]	$O(d)$	Undetermined	$O(\log^2(n))$	Yes
	[21, 27]	$O(n)$	$\Omega(n^{\log_2(n)})^\mp$	$O(\log(n))$	Yes
	[28]	$O(n)$	$\Omega(2^{\frac{n}{4}})^\mp$	$O(\log(n))$	Yes
	[7]	$\Theta(d^2+n)$	$O(\Delta n^3)$	$O(\log(n))$	No
	This paper	$\Theta(d^2)$	$O(n^6)$	$O(\log(n))$	Yes
Any	[16]	$O(n)$	$\Omega(2^n)^\dagger$	$O(\log(n))$	Yes
	[29]	$O(n)$	$\Theta(n^2 d)$	$O(\log(n))$	Yes
	[30]	$\Theta(n)$	$\Theta(n^2)$	$O(\log(n))$	Yes
DFS	[22]	$O(dn\Delta)$	Undetermined	$O(n \log(\Delta))$	Yes
	[23]	$O(n^2)$	$O(n^3)$	$O(\log(n))$	Yes
	[31]	$O(n)$	$O(n^2)$	$O(n \log(n))$	Yes
	[7]	$O(n)$	$O(\Delta n^3)$	$O(\log(\Delta+n))$	No

† This is detailed in the analysis given in [18].
‡ This is detailed in the analysis given in [32].
$^\mp$ This is detailed in the analysis given in [33].

Some of the algorithms cited above are optimal in terms of rounds for the construction of an arbitrary spanning tree or a BFS tree. As discussed in the Introduction section, the number of steps required to compute a solution is an important criterion since it is interesting for the communication complexity. As demonstrated in the analysis given in [18, 32], the algorithms presented in [16, 17] have an exponential number of steps, whereas the one given in [26] has a finite number of steps (with a lower and upper bounds respectively of $\Omega(n^2 Max)$ and $O(n(Max+d)^n)$ steps, where Max is the maximum height value of a node in the initial global state). Moreover, Awerbuch *et al.* [20] proposed a solution to construct a Shortest Path tree with an optimal time complexity of $O(D)$ rounds, however as demonstrated recently by Devismes *et al.* [32] it stabilizes in at least $\Omega(2^{D/2})$ steps (with D an upper bound of the network diameter d). Datta, Larmore, and Vemula [21, 27] resolves the election problem by constructing a BFS tree which stabilizes in $O(n)$ rounds. Later, the same authors proposed in [28] a second self-stabilizing algorithm for the election problem which is very similar to the first one proposed in [21, 27]. To elect a leader in the network, a breadth-first search spanning tree rooted at the elected leader is

also build. Notice that there is no explicit pointer to the parent but it can easily be computed. The main difference between these two algorithms is the way to manage fake leader identifiers, since the system can be placed in incorrect configurations. The algorithm proposed in [28] exploits a special value 0, smaller than any node identifier, which is propagated in the network to clean fake identifiers. However, Altisen *et al.* [33] have shown recently exponential step complexities for these two algorithms. Indeed, the first algorithm proposed in [21, 27] has a step complexity of at least $\Omega(n^{\log_2(n)})$ steps, while the second algorithm proposed in [28] has a step complexity of at least $\Omega(2^{\frac{n}{4}})$. Kosowski and Kuszner give a self-stabilizing algorithm to construct a spanning tree with a bounded number of steps ($\Theta(n^2 d)$ steps are needed) [29]. Recently, in [30] Cournier presented a new stabilizing solution for the construction of an arbitrary spanning tree improving the bound on the number of steps of [29]. This algorithm runs in $\Theta(n)$ rounds and $\Theta(n^2)$ steps. Cournier, Devismes, and Villain proposed a snap-stabilizing solution for the problem of Propagation of Information with Feedback (PIF) [34]. A spanning tree rooted at the source node with the information to propagate is constructed. This algorithm uses a question mechanism to ensure that every processor in the network belongs to the constructed spanning tree, and to guarantee that every processor receives the propagated information. Cournier, Devismes, and Villain give also an efficient transformer to obtain a snap-stabilizing version of a distributed algorithm [7]. They use this transformer to obtain a snap-stabilizing algorithm for the BFS tree problem which runs in $O(d^2 + n)$ rounds and $O(\Delta n^3)$ steps, with Δ the maximum degree of a node in the network. Table 1 summarizes the time complexities (round and step complexities) of some self-stabilizing tree construction algorithms.

Notice that the algorithm presented in [7] does not satisfy the definition of a fully polynomial algorithm since it has a round complexity which is related with the network size. Moreover, the algorithm proposed in [26] has a round complexity only polynomial in d and a bounded step complexity, but its step complexity is dependent on the maximum height value Max in the initial configuration. This value can be arbitrarily high (e.g., exponential on the network size) which does not satisfy the definition of a fully polynomial stabilizing algorithm. However, if one considers the latter algorithm in a non fault-tolerant context, then the algorithm satisfies the fully polynomial property since the round and step complexity are respectively $\Theta(d)$ rounds and $O(dn)$ steps (each node can only execute an action to improve its height, which can be done in time polynomial in d). Therefore, another interesting question can be: Does the stabilization property prevent some algorithms to satisfy the fully polynomial property?

Outline of the paper. The paper is organized as follows. In Section 2 we present the model assumed in this paper. Section 3 gives few insights on the difficulties to design fully polynomial stabilizing distributed algorithms for the spanning tree construction problem. In Section 4, we give a high-level overview of the fully polynomial stabilizing algorithm we propose and give an explanation about the time complexity to solve the BFS tree problem. We then present in detail our approach to construct a BFS tree in Section 5, based on a snap-stabilizing algorithm to the Question-Answer problem given in Section 6. We describe in Section 7 how these stabilizing algorithms are composed together. An example of execution of the algorithm is detailed in Section 8. Finally, Section 9 shows the correctness of our algorithm, and we conclude in the last section.

2. Model

Notations. We consider a network as an undirected connected graph $G = (V,E)$ where V is a set of nodes (or *processors*) and E is the set of *bidirectional asynchronous communication links*. We state that n is the size of G ($|V| = n$). We assume that the network is *rooted*, i.e., among the processors, we distinguish a particular one, *r*, which is called the *root* of the network. In the network, p and q are neighbors if and only if a communication link (p,q) exists (i.e., (p,q) $\in E$). Every processor p can distinguish all its links. To simplify the presentation, we refer to a link (p,q) of a processor p by the *label* q. We assume that the labels of p, stored in the set $Neig_p$, are locally ordered by $\prec_p$. We also assume that $Neig_p$ is a constant input from the system. Δ is the maximum degree of the network (i.e., the maximal value among the local degrees of the processors). A tree $T = (V_T, E_T)$ is an acyclic connected subgraph such that $V_T \subseteq V$ and $E_T \subseteq E$, where the root of tree T is noted by $root(T)$. Moreover, any processor has a *parent* in a tree T which is the neighbor on the path leading to $root(T)$. A processor $p \in V_T$ with at least two neighbors in tree T is called an *internal* processor and a *leaf* processor otherwise.

Programs. In our model, protocols are *semi-uniform*, i.e., each processor executes the same program except *r*. We consider the local shared memory model of computation. In this model, the program of every processor consists of a set of *variables* and an *ordered finite set of actions* inducing a *priority*. This priority follows the order of appearance of the actions into the text of the protocol. A processor can write to its own variable only, and read its own variables and that of its neighbors. Each action is constituted as follows: $< label > :: < guard > \rightarrow < statement >$. The label is used as a name to refer to an action in the program. The guard of an action in the program of p is a boolean expression involving variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard is satisfied. The *state* of a processor is defined by the value of its variables. The *state* of a system is the product of the states of all processors. We will refer to the state of a processor and the system as a (*local*) *state* and (*global*) *configuration*, respectively. We note $\mathcal{C}$ the set of all possible configuration of the system. Let $\gamma \in \mathcal{C}$ and A an action of p ($p \in V$). A is said to be *enabled* at p in γ if and only if the guard of A is satisfied by p in γ. Processor p is said to be *enabled* in γ if and only if at least one action is enabled at p in γ. When several actions are enabled simultaneously at a processor p: only the priority enabled action can be activated.

Let a distributed protocol P be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$. An *execution* of a protocol P is a *maximal* sequence of configurations $e = (\gamma_0,\gamma_1,...,\gamma_i,\gamma_{i+1},...)$ such that, $\forall i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$ (called 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 P is enabled in the terminal configuration) or infinite. All executions considered here are assumed to be maximal. $\mathcal{E}$ is the set of all possible executions of P. We can give in Definition 2 a definition of *Snap-Stabilization*.

Definition 2 (Snap-Stabilization). *Let $\mathcal{T}$ be a task and $\mathcal{F}$ be a specification of $\mathcal{T}$. A protocol P is snap-stabilizing for $\mathcal{F}$ if and only if starting from any configuration every execution of P satisfies $\mathcal{F}$.*

As we already said, each execution is decomposed into steps. Each step is shared into three sequential phases atomically executed: (i) every processor evaluates its guards, (ii) a *daemon* (also called *scheduler*) chooses some enabled processors, (iii) each chosen processor executes its priority enabled action. When the three phases are done, the next step begins.

A *daemon* can be defined in terms of *fairness* and *distributivity*. In this paper, we use the notion of *unfairness*: the *unfair* daemon can forever prevent a processor from executing an action except if it is the only enabled processor. Concerning the *distributivity*, we assume that the daemon is *distributed* meaning that, at each step, if one or more processors are enabled, then the daemon chooses at least one of these processors to execute an action.

We consider that any processor p executed a *disabling action* in the computation step $\gamma_i \mapsto \gamma_{i+1}$ if p was *enabled* in γ_i and not enabled in γ_{i+1}, but did not execute any protocol action in $\gamma_i \mapsto \gamma_{i+1}$. The disabling action represents the following situation: at least one neighbor of p changes its state in $\gamma_i \mapsto \gamma_{i+1}$, and this change effectively made the guard of all actions of p false in γ_{i+1}.

To compute the time complexity, we use the definitions of *round* and *step*. The definition of round captures the execution rate of the slowest processor in any execution. Given an execution e ($e \in \mathcal{E}$), the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action (an action of the protocol or a disabling action) of every enabled processor from the initial configuration. Let e'' be the suffix of e such that $e = e'e''$. The *second round* of e is the first round of e'', and so on. On the contrary, given an execution $e \in \mathcal{E}$ the definition of step allows to estimate more accurately the amount of computations needed by a distributed protocol to recover a correct behavior.

Therefore, the time complexity (or convergence time) of a stabilizing algorithm is the maximum time (in steps or rounds) over every possible execution under a considered daemon to reach a terminal (legitimate) configuration, starting from any initial configuration.

3. About the Spanning Tree problem

As presented in Table 1 in Related work, the existing self-stabilizing spanning tree constructions have either a polynomial step complexity but require $\Omega(n)$ rounds, or a round complexity polynomial on the network diameter but with an unbounded or exponential step complexity. So, in this section we focus on the difficulty to obtain a trade-off between the round and step complexities for the spanning tree construction problem.

A spanning tree $T = (V', E')$ of a graph $G = (V, E)$ is a connected sub-graph spanning all the processors of the network (i.e., (i) $V' = V$, (ii) $E' \subseteq E$ and (iii) for any pair of nodes (x, y) there exists a unique path between x and y in T). As a consequence, T is an acyclic sub-graph and we have that $|E'| = |V| - 1$. Most of the existing spanning tree approaches construct a spanning tree of the network rooted at a given node r. To this end, a solution is computed by selecting a neighbor $u \in V$ for each node $v \in V$ as its parent in the tree, i.e., the closest neighbor of v on the path to the root in the computed tree T. Generally, a *level* (or *height*) is associated to each node v in T indicating the number of nodes (or hops) between v and r. The maintenance of height values at each node is largely used to solve the spanning tree construction task since it allows to easily detect cycles (e.g., by comparing them with the network size or with other values in the neighborhood).

In the following, we consider three self-stabilizing approaches for the spanning tree problem whose techniques are the most related to the solution we propose in this paper. We give some insights regarding the time complexity (number of rounds and steps) needed to construct a solution to the considered problem. To compare these algorithms from the step complexity point of view, we will consider the number of changes on the height and parent for any node (except the root) until reaching a legitimate configuration.

3.1. Classic approach

A classic approach to construct a spanning tree rooted at a node r is to use a simple rule based on network distances: Each node is connected to the neighbor closest (according to the network distances) to r in the network. To this end, each node maintains its height in the tree which yields to the construction of a BFS tree. The root r has a height equal to zero, while every node (except the root) selects as its parent the neighbor with the smallest height (and smallest identifier or channel port in case of neighbors with equal height). This approach was used in several works to design a self-stabilizing BFS algorithm, e.g., as the algorithm proposed by Ducourthial et al. in [26].

By induction on the network distances starting from the root node, it is easy to show that a BFS tree can be constructed in a self-stabilizing manner in $O(d)$ rounds, with d the network diameter. Indeed, each new layer of nodes at distance k from the root is stabilized in a constant number of additional rounds, when the layers at distance $j, 0 \leq j < k$, have been stabilized.

However, the use of this rule without any other mechanism can yield to a high step complexity. Indeed, this approach is dependent of the height values initially present in the network, as shown in Figure 1.

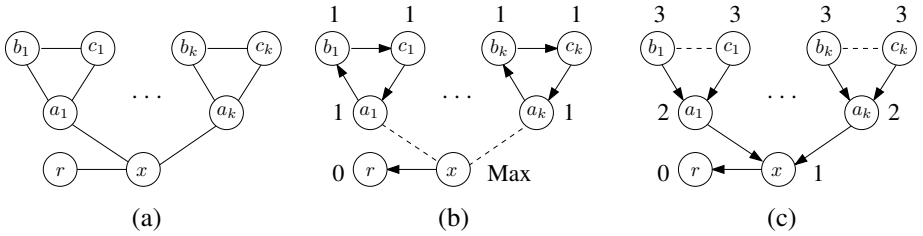

Figure 1: Example used to show that the classic spanning tree approach has a step complexity related to the maximum height value in the network.

In Figure 1(a), a special network is used which is composed of k cycles (we take $k = (n-2)/3$ in Figure 1) of three nodes (this can be generalized by using cliques of at least three nodes) which are connected to the root r via another node x. We can consider a special initial configuration of the system shown in Figure 1(b), where the root has a zero height and all the nodes have a height equal to one, except x which has the maximum possible height value noted Max. Node x is connected to r, and the parent link of the other nodes follows the k cycles. From this particular initial configuration, we can have the following distributed execution. Each node in the k cycles can execute concurrently the rule of the algorithm to increase their height until Max. So, after $Max \times 3k$ steps a new configuration is reached where all the nodes have a height value equal to Max (except the root r). From this new configuration, each node can execute its algorithm to select the neighbor of minimum height and to update its height to be equal to its parent's height plus one. The following order is used to reach in $n-1$ additional steps the legitimate configuration shown in Figure 1(c): x is executed to correct first its height to be equal to 1, then for $1 \leq i \leq k$, nodes a_i select x as their parents with a height equal to 2, nodes b_i and c_i select a_i as their parent with a height equal to 3. This approach leads to a step complexity of $Max \times 3k + n - 1 = \Omega(Max \times n + n)$ steps, since we have $k = (n-2)/3$.

We can notice that the number of parent link changes of a node is also dependent of the Max value, as illustrated in Figure 2. To demonstrate this point, we are interested in the node

y in the network of Figure 2(a). Consider the initial configuration in Figure 2(b). We can have the following distributed execution. For $i, 1 \leq i \leq k$, each pair of nodes a_i and b_i execute concurrently their algorithm to increase their height value by one. This leads y to be connected $k-1$ times to the new parent b_{i+1}. This phase can be iterated Max times (where y is connected again to b_1 between each iteration). We can observe that the node y suffers from $\Omega(Max \times k) = \Omega(Max \times n)$ parent link changes, since $k = (n-3)/2$ in Figure 2.

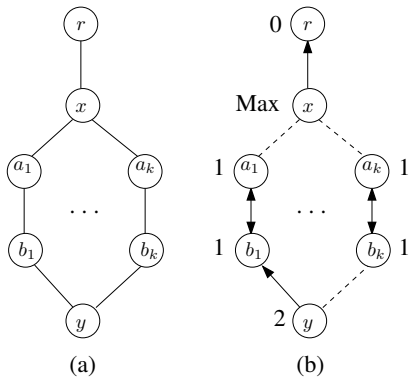

Figure 2: Example used to show that the number of parent link changes with the classic spanning tree approach is related to the maximum height value in the network.

As presented above with Figure 1 and 2, this approach has a step complexity dependent of the maximum initial value in the network, noted Max, which can be an exponential function of the network size. Therefore, to solve the spanning tree construction problem, we cannot establish a polynomial step complexity using only this simple rule. In the following, we present two other approaches for the spanning tree construction problem which have a polynomial step complexity. To achieve this goal, the crucial point considered in these approaches is to bound the number of parent link changes.

3.2. First approach with polynomial bound in steps

To our knowledge, Kosowski et al. proposed in [29] the first self-stabilizing algorithm to construct a spanning tree with a polynomial step complexity. Like the previous approach, height values are maintained for the selection of a parent in the neighborhood of each node. Contrary to the previous classic approach, this algorithm assigns a height to each node by increasing these values using the following rule: Each node v can increase its height value if all its neighbors have an equal or higher height value. In this case, v takes a new height equal to the maximum height in its neighborhood plus one. The idea behind this rule is to reach a configuration in which there is a path between the root and every node in the network with increasing height values (note that the difference on height values between two adjacent nodes in the spanning tree can be higher than one). The configurations which satisfy this property are legitimate and define a correct spanning tree, since no cycle is introduced on the described paths because of increasing height values. We can construct a rooted spanning tree by introducing a second rule (only enabled if the first one is not) to select as the parent of each node the neighbor of smallest height (and of minimum identifier or channel port in case of neighbors with equal height).

This approach has a round complexity of $O(n)$ rounds. This round complexity comes from the fact that for each node the height values must be corrected until having at least one path leading to the root with only increasing height values. The length of such a path can be of at most n nodes, and each node having no neighbor with a smaller height must execute its algorithm.

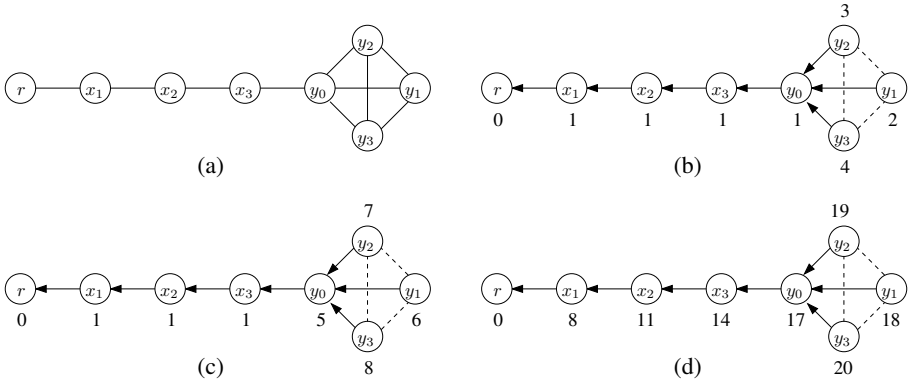

Figure 3: An example used to show the lower bound on step complexity which matches the upper bound proved in [29]. Here we consider $k = 3$, but this example can be generalized with a longer chain (nodes x_i) and a bigger clique (nodes y_i), with $0 \leq i \leq k$.

We will now analyze the step complexity of this algorithm by showing a lower bound of $\Omega(dn^2)$ steps which matches asymptotically the upper bound of $O(dn^2)$ steps shown in [29]. To this end, we consider the network topology given in Figure 3(a) which is composed of a clique of four nodes (y_k with $0 \leq k \leq 3$) connected to the root r via a chain of three nodes (x_k with $0 \leq k \leq 3$ and $r = x_0$). Starting from the initial configuration of Figure 3(b), only the node y_0 can execute the first rule to increase its height value to 5 (maximum height value of neighbors plus one). This yields to the execution of the second rule for each node of the clique (except y_0 and y_1) to select y_1 as their new parent, which involves $k-1$ steps. This process is iterated in the clique when a new node execute its first rule. More formally, we have $j, 0 \leq j \leq k$, iterations of this process in the clique as following: y_j executes the first rule to increase its height, then all the nodes y_i (except y_0 and y_j) execute the second rule to select y_{j+1} as their new parent. Thus, after $k(k-1)$ steps we reach the configuration given in Figure 3(c) in which no node y_k can execute one of its rule. However, the system has not reached a legitimate configuration. Indeed, we have not increased the height values along the chain composed of nodes x_k. So, from configuration of Figure 3(c) only the node x_3 can execute its first rule to increase its height to 6, which triggers a new phase defined by the procedure described above in the clique and involving $k(k-1)$ additional steps. Finally, we can observe that we have at most $k-1$ phases to reach the legitimate configuration given in Figure 3(d). In each phase, the node $x_i, 1 \leq i \leq k$, (by decreasing order of i) executes its first rule followed by the execution procedure described in the clique. Therefore, we have a step complexity of $\Omega(k^3) = \Omega(dn^2)$, since the first factor k is dependent of the length of the chain and in the example of Figure 3 we have $k = (n-2)/2$.

A crucial point with this second approach is that the step complexity of the algorithm is not function of the maximum height value initially present in the network. Moreover, we can notice that the step complexity can be bounded by the fact that the parent link can be changed at most n^2 for each node, as illustrated in the example of Figure 3. However, the number of parent link

changes for a node is important and can be a problem from an application point of view using the spanning tree to broadcast information. So, a natural question is to determine if it is possible to design a self-stabilizing algorithm with a smaller upper bound on parent link changes, which can also lead to a better step complexity. This is the motivation of the third approach presented in the next subsection.

3.3. An approach improving the bound in steps

Cournier [30] proposed a self-stabilizing spanning tree construction which improves the step complexity of the algorithm proposed in [29]. As discussed above, this work addresses the problem related to the maximum number of parent link changes per node. To achieve this goal, an additional mechanism is introduced to better control this aspect, allowing to reduce the upper bound on the step complexity.

This approach constructs an arbitrary spanning tree rooted at a designated node, noted r. As presented in the previous approaches, each node maintains a height and a parent pointer. A node is considered as the root of its tree if its parent has a higher height value than itself. These elements allow to define a forest of trees. To explain the additional mechanism used we introduce the notion of *abnormal* tree. A tree is called *abnormal* if the tree is rooted at a node x such that $x \neq r$, in this case x is an *abnormal* root. A tree is called *correct* if the tree is rooted at node r. In the algorithm proposed in [30], contrary to the previous approaches a node v can change its parent link only when v learns it belongs to an abnormal tree. Moreover, v tries to connect to a tree which is not abnormal via a neighbor u which does not learn it belongs to an abnormal tree. Every node can learn it belongs to an abnormal tree thanks to the use of the additional mechanism which works as follows. When a node x detects it is an abnormal root, then it propagates down in its tree an *Error* status in the tree using a Propagation of Information with Feedback (PIF) protocol [35, 36, 20, 34]. When every node in the subtree of x learn they belong to an abnormal tree (i.e., when the PIF has finished), x propagates down a *Leave* status allowing its descendants to leave the tree. At this point, the considered abnormal tree is frozen and no new node can be connected to it. Every node connected to the correct tree has a *Correct* status. Since a node with a *Correct* status cannot leave the tree it belongs to, in finite time (because the network size is finite) every node will belong to the correct tree rooted at r.

This approach has a round complexity of $O(n)$. Indeed, the additional mechanism allowing to freeze the abnormal trees, with the use of a PIF procedure, is performed in time function of the tree depth (i.e., in $O(n)$ rounds since the depth of a tree is bounded by n). So, in $O(n)$ rounds there is only a correct tree and abnormal trees in the forest, since all abnormal trees are frozen in $O(n)$ rounds. Moreover, no additional node can be connected to any abnormal tree of the forest. As a consequence, in at most $O(n)$ additional rounds every node in all the abnormal trees are connected to the correct tree, which is rooted at r.

To analyze the step complexity, we need to make some observations for each node in the network (except the root which can execute no action). First, no node with a *Correct* status can leave the tree it belongs to. As a consequence, the nodes in the correct tree never leave this one. Second, a node which learns it belongs to an abnormal tree can join another tree via a neighbor with a *Correct* status. Therefore, thanks to the mechanism used to freeze the abnormal trees a node belonging to an abnormal tree can join at most $n - 1$ trees until joining the correct tree. Finally, the author shows that any node in a correct tree can perform at most a constant number of actions, these are used to maintain height values in the tree such that each node has a height value equal to its parent's height plus one. Putting it all together, this algorithm has a step complexity

of $O(n^2)$. Moreover, as with the approach proposed in [29] the step complexity is not dependent of the height values initially present in the network.

4. Outline of the proposed algorithm

In this section, we give an overview of the fully polynomial stabilizing approach proposed in this paper for the construction of a Breadth First Search tree. As stated in the previous sections, the main difficulty is to design an algorithm with a round complexity function of the network diameter with a polynomial step complexity.

The self-stabilizing algorithm we propose in this paper is based on the following principles to construct a BFS tree rooted at node r:

1. First of all, each node p always attempts to connect to the neighbor q of minimum height belonging to the normal tree (i.e., the BFS tree rooted at r). To join the normal tree, p must wait for a connection authorization from q to select it as its new parent.
2. When a node q receives a connection request from at least one neighbor p which want to join the tree q belongs to, q has to check first that it belongs to the normal tree. This verification is performed by using a question-answer mechanism which allows q to ask an authorization from the root of the tree it belongs to. Connection authorizations are only delivered to asking node belonging to the normal tree. Moreover, these authorizations are delivered by increasing order of the height of asking nodes in the normal tree. This allows to control the acceptance of new nodes from the lowest to the highest levels in the normal tree (defining the priority of requests). As soon as the verification is over for a asking node q (i.e., a positive answer is delivered by the root r), q delivers a connection authorization to its neighbors p (out of the normal tree) allowing them to select q as their new parent to join the normal tree.
3. If a node detects an inconsistency between its state and the state of its parents, then the node goes away from the tree it belongs to by setting its status to Error and selecting himself as a root. This allows the detection of abnormal trees.

Note that the use of connection authorizations to add new nodes in the normal tree allows to limit the number of parent changes performed by each node. Moreover, the construction of the BFS tree allows to reduce the cost of the verification (question-answer mechanism) executed in Point 2 above, since it is related to the diameter of the network in the worst case.

To construct a BFS tree, each processor must be connected to its closest neighbor toward the root r. As presented in Section 3, since there is a single root r in the network this general approach allows to connect all the processors to the tree rooted at r in a distributed and self-stabilizing manner in $O(d)$ rounds [26]. However due to a bad initial configuration, we also explained that the construction cannot be done using a polynomial step complexity because of a non polynomial number of connections of a node to a tree which is not rooted at r. This was the motivation to bound the number of connections to this kind of trees by introducing a mechanism to freeze these trees [29, 30]. However, the counter part in these approaches is that their round complexity is driven by the mechanism used to freeze the abnormal trees, i.e., function of the network size. Therefore, in our approach we use an additional questioning mechanism allowing us to control the expansion of the tree rooted at r, while limiting the processor connections to abnormal trees.

Our algorithm $\mathcal{BFS}$ is composed of two sub-algorithms: the first sub-algorithm performs the connection of the processors to the spanning tree (see Section 5), while the second one allows

or not the connections of processors in order to obtain the desired tree (i.e., in our case a BFS tree) (see Section 6). The second sub-algorithm is considered as an oracle by the first one as explained below. As the approaches presented in Section 3, each node maintains a parent pointer to a neighbor and a level value indicating the number hops to the root in the tree it belongs to.

Abnormal trees detection. We consider that there is a processor r designated as the root of the spanning tree. We consider a forest of trees and we use the notion of *abnormal trees* introduced in [30], i.e., trees rooted at a node with a level lower than its parent. The tree rooted at r is called a *normal tree*. We also take the principle to detect the abnormal trees, but without the acknowledgement of all the descendants. That is, each processor p checks if its level is higher than the one of its parent. If this constraint is not satisfied then p takes the *Error* status and propagates this status in its subtree. That is, we do not use a PIF protocol, instead we only use a mechanism to propagate the *Error* status.

Connection of processors. Contrary to the approach proposed in [30], each node can leave the tree it belongs to without the permission of the root. Indeed, we do not use a PIF mechanism to have more flexibility for the connection of processors. However, before leaving the tree it belongs to a node must have the authorization from a neighbor to connect to him. Therefore, the connection of processors is controlled by a questioning mechanism viewed as an oracle by the first sub-algorithm, which delivers connection authorizations in a way allowing to construct a BFS tree. The connection procedure of a processor p is as follows: A processor q sends a request to the oracle, when q detects a neighbor p such that p has an *Error* status or p has a level higher than q's level plus one. If q receives a response from the oracle then p has the authorization to connect to q. Note that we consider that the oracle responds only to the processors which belong to the normal tree. Therefore, a node can only connect to the normal tree, except if q has a response from the oracle because of a bad initial configuration.

Oracle. The questioning mechanism is used by each processor q with a neighbor to connect in order to detect if q really belongs to the normal tree or not. To this end, the request of q is sent to the root of the tree q belongs to. This request is forwarded by the ancestors of q in the tree. If this request is received by the root r, then r sends a (positive) response to q which follows the path used to forward q's request, otherwise q receives no response. This mechanism is a semi-algorithm because only positive responses are sent, which allows to not execute useless actions. The construction of a BFS tree involves the connection of the nodes via the closest neighbor to the root. Therefore, the root r must respond first to the request of processors of smallest level. To this end, the oracle takes into consideration the priority associated to each request, which is related to the height in the tree of the asking processor. Thus, a request sent by a processor q' closer than q from r erases q's request on the common path used to forward these two requests, even if r still respond to the request of q. In this case, when the response to the request of q' is forwarded then q's request is forwarded again on the part of the path erased by the request of q'. The responses are delivered level by level starting from the root. So, we can identify a classic method to construct a BFS tree based on waves used to add a new layer of processors with a synchronization to the root between two consecutive waves. Note that, a processor in a *Error* status cannot execute the actions related to the oracle, it can only execute the first sub-algorithm in order to hook to another neighbor with an authorization.

Time complexity considerations. First of all, we consider the round complexity of the whole algorithm. Algorithm $\mathcal{BFS}$ uses a questioning mechanism which can be viewed as a synchronizer allowing to construct a BFS tree rooted at r layer by layer, the addition of any new layer of processors is dependent on the acknowledgement to a permission request. The requesting processors closest to r at height k receive an acknowledgement to their request from r in $O(k)$ rounds (see Corollary 1) which allows their neighbors to hook on to the normal tree. The same argument holds for the addition of each new layer. Moreover, the height of a BFS tree is lower than or equal to the network diameter. Therefore, summing up the round complexity associated to each layer we obtain a $O(d^2)$ round complexity to construct a BFS tree, with d the network diameter. On the other hand, the mechanism we use for deleting the abnormal trees is obviously in $O(n)$ rounds, since the height of such a tree can be in $O(n)$. But any processor in an abnormal tree far from the root of this tree will become the neighbor of at least a processor of the normal BFS tree in $O(d^2)$ rounds and will hook to it even if the abnormal tree is not yet deleted. So the global round complexity is still $O(d^2)$ (see Lemma 19).
We give below the main arguments allowing to show that Algorithm $\mathcal{BFS}$ has a polynomial step complexity in $O(\Delta mn^3 + mn^4)$. We first consider the step complexity of the first sub-algorithm. The questioning mechanism is designed to avoid that a processor can hook to an abnormal tree several times by the same neighbor. Therefore, a processor can hook to an abnormal tree at most Δ times until reaching the normal tree (with Δ the maximum degree of a node) and at most n times when it belongs to the normal tree until reaching its final position. This involves that there are at most $\Delta n + n^2$ connections until all the processors reach their correct position in the final BFS tree. We consider now the step complexity of the questioning mechanism. The connection of a processor is the result of a request, done by at most each of its neighbors (i.e., at most Δ requests per processor connection). So, to construct a BFS tree at most $\Delta m + mn$ requests are generated by the first sub-algorithm. A processor receives a response from the oracle in $O(n^2)$ steps, since the oracle handles first the closest requests in the tree. Note that a synchronization is performed for the parallel requests of same priority which yields a polynomial number of retransmissions. Thus, in $O(n^3)$ steps every requesting processor in the normal tree receives a response (there are at most n concurrent requests in the network). Given an upper bound on the number of generated requests, we have to multiply this amount by the number of steps needed by the oracle to respond to these requests in order to obtain an upper bound to the total step complexity of Algorithm $\mathcal{BFS}$.
Notice that in one hand using a questioning mechanism allows us to save steps by avoiding the transmission of useless requests, but on the other hand we obtain a higher round complexity ($O(d^2)$ instead of $O(d)$ with standard algorithms constructing BFS trees) due to the fact that permissions must be delivered before the add of new nodes to the constructed tree. Moreover, the step complexity (see Lemma 30) is not related with any initial value of a variable and it holds under any fairness assumptions.

5. Spanning Tree Construction

In this section, we are interested in the problem of constructing a tree spanning all the processors of the network. We consider a particular *root* processor, noted r, which is used to construct a spanning tree. More precisely, we consider the construction of a *Breadth First Search* (BFS) tree rooted at processor r. We can define a BFS tree as in Definition 3.

Definition 3 (BFS Tree). *Let $G = (V, E)$ be a network and r a node called the* root*. A graph $T = (V_T, E_T)$ of G is called a* Breadth First Search *tree if the following conditions are satisfied:*

1. *$V_T = V$ and $E_T \subseteq E$, and*
2. *T is a connected graph (i.e., there exists a unique path in T between any pair of nodes $x, y \in V_T$) and $|E_T| = |V| - 1$, and*
3. *For each node $p \in V_T$, the path between p and r in T is a shortest path (in hops) between p and r in G.*

We give a formal specification to the problem of constructing a BFS tree, stated in Specification 1.

Specification 1 (BFS Tree Construction). *Let $\mathcal{C}$ the set of all possible configurations of the system. An algorithm $\mathcal{A}_{\mathcal{BFS}}$ solving the problem of constructing a stabilizing BFS tree satisfies the following conditions:*

[TC1] *Algorithm $\mathcal{A}_{\mathcal{BFS}}$ reaches a set of terminal configurations $\mathcal{T} \subseteq \mathcal{C}$ in finite time, and*

[TC2] *In every configuration $\gamma \in \mathcal{T}$ there exists a spanning tree satisfying Definition 3.*

5.1. Breadth first search tree algorithm

In this section, we present a snap-stabilizing algorithm, called $\mathcal{BFS}$, to construct a BFS tree. Algorithm $\mathcal{BFS}$ is a semi-uniform algorithm, this means that exactly one of the processors, called the *root* and denoted r, is distinguished. This distinguished processor is used in Algorithm $\mathcal{BFS}$ as the root of the spanning tree.

Algorithm $\mathcal{BFS}$ is a composition of two algorithms. Algorithm 1 is based on the fact that a processor has to choose a neighbor with the minimal distance to the root as its parent in the tree. It is well known that this common idea is enough to get a round complexity in $O(d)$, but does not ensure a step complexity in $O(n^b)$[5]. So we allow a processor to connect to a neighbor only if this neighbor is in the tree rooted at r and in the shortest path to r. The detection of such neighbors is assigned to Algorithm 2 (see Section 6) which can be seen as an oracle by Algorithm 1. The second role of Algorithm 1 is to remove the abnormal trees, i.e., those that are not rooted at r.

5.1.1. Variables

We define below the variables used by Algorithm 1. For Algorithm 1, we characterize r by the predicate $Allowed$ (i.e., $Allowed(p) \equiv (p = r), \forall p \in V$).

Shared variable. Each processor $p \in V$ has a local shared variable $p.Req$ which is used by Algorithm 1 to monitor Algorithm 2 at p. This shared variable can take four values: ASK, $WAIT$, REP, and OUT. By setting the shared variable $p.Req$ to ASK, Algorithm 1 informs Algorithm 2 that a permission from the root of the tree that p belongs to is needed at p. In this case, Algorithm 2 tries to send a request and to obtain a permission for p if it is possible (i.e., if p belongs to an allowed tree and this request has the highest priority during enough time for

[5]Indeed, this approach is used in [17] to construct a BFS tree with a round complexity in $\Theta(d)$ but with a step complexity in $\Omega(Max \times n^2)$, as demonstrated in [18]. However, Max is an upper bound of n and can be arbitrary high with respect to n so the step complexity can be at least exponential. Note that the gap between the lower and the upper bound (see Table 1) of the step complexity leads us to think that the lower bound in [18] is not tight.

an acknowledgment to return). If a permission is delivered to processor p, then Algorithm 2 sets this shared variable to REP in order to inform Algorithm 1. Then, every neighbor of p can execute Algorithm 1 to join the tree that p belongs to. When there is no neighbor of p to connect, then Algorithm 1 sets $p.Req$ to OUT which allows Algorithm 1 to request another permission through Algorithm 2 if needed.

Local variables. Each processor $p \in V$ maintains three local variables:

- $p.P$: it gives the parent of p in the tree it belongs to, $p.P = \perp$ for processor $p = r$.
- $p.L$: it stores the level (or height) of p in the tree it belongs to, $p.L = 0$ for processor $p = r$.
- $p.S$: it defines the status of processor p. It can take two values: E if p does not belong to a tree rooted at a processor x satisfying Predicate $Allowed(x)$, C otherwise. We have $p.S = C$ for processor $p = r$.

5.1.2. Algorithm description

As described before, we consider a forest $\mathcal{F}$ of trees and a distinguished processor r which is the only processor authorized to deliver permissions in the network (i.e., $Allowed(p) \equiv (p = r)$ for every processor $p \in V$). We can notice that in a tree there is a strong constraint between the level of a processor and the level of its parent in the tree: For any processor $p \neq r$, the level of p's parent must be equal to p's level minus 1. Therefore, the root of a tree in forest $\mathcal{F}$ is either (i) processor r, or (ii) a processor $p \neq r$ such that $p.L \leq (p.P).L$ (it is used to detect cycles in the network). Since we want to construct a spanning tree, in case (ii) we say that processor p is an *abnormal root*. Moreover, any processor $p \neq r$ in a tree in $\mathcal{F}$ rooted at an abnormal root belongs to an *abnormal tree*. Every processor $p \in V$ in an abnormal tree can execute $E\text{-}action$ to change its Status to E (i.e., $p.S = E$) and to inform its descendants in the tree (see the formal description of Algorithm 1). Note that to reduce the number of moves executed by Algorithm $\mathcal{BFS}$, a processor $p \in V$ in an abnormal tree does not ask any permission. Processor p waits until a neighbor q in the tree rooted at r authorizes p to connect to q.

When a BFS tree is constructed, the following property is verified at each processor $p \in V, p \neq r$: The level of p's parent is equal to p's level minus 1 (i.e., $(p \neq r) \Rightarrow (p.L = (p.P).L + 1)$). For processor r, we have the following constant values: r has no parent and a level equal to zero (i.e., $(p = r) \Rightarrow (p.P = \perp \wedge p.L = 0)$). Moreover, according to Claim 3 of Definition 3 we must have that the deviation on the level values between any processor $p \in V$ and its neighbors does not exceed one (i.e., $\forall q \in Neig_p, |q.L - p.L| \leq 1$). If one of these above constraints are not verified then a BFS tree is not constructed. Therefore, we have either at least one abnormal tree in $\mathcal{F}$ or there is a processor $p \in V$ with a neighbor q such that $q.L - p.L > 1$ (i.e., Predicate $GP\text{-}REP(p)$ is satisfied at p). In these cases, processor p executes $A\text{-}action$ to set the shared variable $p.Req$ to ASK in order to ask the permission to allow q to connect to p, if p is not already asking a permission (i.e., we have $p.Req = OUT$). To this end, Algorithm 2 sends a request to the root of the tree.

Inputs for Algorithm 2. In order to allows Algorithm 2 to send a request the following inputs are given at processor p: (i) $Child(p)$ is the set of children of p in the tree (i.e., $Child(p) \equiv \{q \in Neig_p : q.P = p\}$), (ii) $Parent(p)$ is the parent of p in the tree (i.e., $Parent(p) \equiv p.P$), (iii) $Height(p)$ is the height in the tree of the requesting processor p, and (iv) $Allowed(p)$ is a

Algorithm 1 Spanning Tree Construction for any $p \in V$

Inputs: $Neig_p$: set of (locally) ordered neighbors of p;
Shared variable: $p.Req \in \{ASK, WAIT, REP, OUT\}$;

Macros:

$Child(p) = \{q \in Neig_p :: q.P = p \wedge q.L = p.L + 1\}$
$Parent(p) = p.P$
$Height(p) = p.L$
$ChPar(p) = \{q \in Neig_p \backslash Child(p) :: q.S = C\}$
$MinChPar(p) = \min\{q \in ChPar(p) :: \forall t \in ChPar(p), q.L \leq t.L\}$

Global Predicates:

$GoodT(p) \equiv p.S \neq E \wedge (p \neq r \Rightarrow p.L = (p.P).L + 1)$
$GoodL(p) \equiv (\forall q \in Neig_p :: |p.L - q.L| > 1 \Rightarrow (p.L < q.L \vee q.S = E))$
$GP\text{-}REP(p) \equiv (\exists q \in Neig_p :: q.S = E \vee q.L - p.L > 1)$
$Start(p) \equiv p.Req = OUT \wedge GP\text{-}REP(p)$
$End(p) \equiv p.Req = REP \wedge \neg GP\text{-}REP(p)$

Algorithm for $p = r$:

Constants: $p.S = C; p.P = \perp; p.L = 0;$
Predicates:
$Allowed(p) \equiv true$
Actions:
$A\text{-}action :: Start(p) \rightarrow p.Req := ASK;$
$O\text{-}action :: End(p) \rightarrow p.Req := OUT;$

Algorithm for $p \neq r$:

Variables: $p.S \in \{C, E\}; p.P \in Neig_p; p.L \in \mathbb{N};$
Predicates:
$Allowed(p) \equiv false$
$AbnormalTree(p) \equiv p.S = C \wedge ((p.P).S = E \vee (p.P).L \geq p.L)$
$Connect(p) \equiv (\exists q \in Neig_p :: q.Req = REP \wedge q = MinChPar(p) \wedge (p.S = C \Rightarrow p.L - q.L > 1))$
Actions:
$E\text{-}action :: AbnormalTree(p) \rightarrow p.S := E;$
$C\text{-}action :: Connect(p) \rightarrow p.S := C; p.P := MinChPar(p); p.L := (p.P).L + 1; p.Req := OUT;$
$A\text{-}action :: Start(p) \rightarrow p.Req := ASK;$
$O\text{-}action :: End(p) \rightarrow p.Req := OUT;$

predicate which notifies if p can deliver permissions (i.e., $Allowed(p) \equiv (p = r)$). We remind that $Allowed(p)$ must be satisfied only at processor $p = r$ in Algorithm 2 to allow that eventually every processor joins the tree rooted at r, since eventually the processors cannot join another tree in the forest $\mathcal{F}$.

In the case a permission is delivered at processor p (i.e., we have $p.Req = REP$), then each neighbor q of p can execute $C\text{-}action$ to connect to p. However to construct a BFS tree without an overcost on moves, processor q waits for until its neighbor x with the smallest level in a normal tree gives a connection authorization to q by executing $C\text{-}action$ (i.e., we have $x.Req = REP \wedge x = MinChPar(q)$). When processor q executes $C\text{-}action$ then it sets its variables $q.P$ and $q.L$ according to its new parent in the tree, and it changes its status to Status C and its shared variable $q.Req$ to OUT. Finally, if there is no neighbor for which processor p needs a permission (i.e., Predicate $GP\text{-}REP(p)$ is no more satisfied at p), then p executes $O\text{-}action$ to set its shared variable $p.Req$ to OUT. This informs Algorithm 2 that the permission can be removed at p, then this allows p to ask a new permission later.

6. Question-Answer problem

In this section, we present a snap-stabilizing algorithm to implement the oracle used by the BFS tree construction given in Section 5. Formally, this oracle has to solve the Question-Answer

problem which can be stated as following, a formal specification is given in Specification 2.

Given a static forest $\mathcal{F}$ of trees in a network $G = (V, E)$, a set of processors $De \subseteq V$ requesting a permission to make a defined computation and a set of processors $AP \subset V$ authorized to deliver permissions. Each $p \in AP$ is a root of a tree $T \in \mathcal{F}$. The *Question-Answer* problem is to deliver a permission (or *acknowledgement*) to a processor p in a tree $T \in \mathcal{F}$ if and only if the root q of T is in AP.

Specification 2 (Question-Answer). *Let $G = (V, E)$ be a network and $\mathcal{F}$ the static forest of trees in G. Let a tree $T \in \mathcal{F}$ and $root(T)$ the root of T. T is an* allowed tree *if $root(T) \in AP$ and* not allowed *otherwise. A protocol P which resolves the Question-Answer problem satisfies:*

- ***Liveness 1:*** *During an infinite execution, if a processor has to send infinitely often a request and it cannot send its request in an allowed tree, then there exist an infinite number of requests which were sent.*
- ***Liveness 2:*** *For every execution suffix, if a processor in an allowed tree has sent a request at time t, then there exist at least one processor in the same tree which receives an acknowledgement to its own sent request at time $t' > t$.*
- ***Safety 1:*** *Every processor which has sent a request receives at most one acknowledgement causally related to its sent request.*
- ***Safety 2:*** *Every processor in a not allowed tree which has sent a request never receives an acknowledgement.*

Remark that only semi-algorithms can satisfy Specification 2, that is no acknowledgement is sent to processors in a not allowed tree, from Property (Safety 2) of Specification 2.

6.1. Question-Answer algorithm

In this section, we present a snap-stabilizing algorithm for the Question-Answer problem, a formal description is given by Algorithm 2. This is a non-uniform algorithm because some rules are only executed by a subset of processors $p \in V$ satisfying a local Predicate $Allowed(p)$ (i.e., p can deliver a permission or not).

6.1.1. Variables

We define below the different variables used by Algorithm 2.

Shared variable. Each processor $p \in V$ has a local shared variable $p.Req$ which allows an external algorithm to require the Question-Answer algorithm at p. This shared variable can take four values: ASK, $WAIT$, REP, and OUT. By setting the shared variable $p.Req$ to ASK in the external algorithm, p requests a permission through the Question-Answer algorithm to its root of the tree. To this end, Question-Answer algorithm tries to send a request to the root of the tree and sets the shared variable $p.Req$ to $WAIT$. The request sent by requesting processors with the lowest level (or height) in the tree will reach the root and then receive a permission (an *acknowledgement*). When p receives an acknowledgement, it sets $p.Req$ to REP. Finally, the external algorithm must set $p.Req$ to OUT to request another permission through Question-Answer algorithm.

Local variables. Each processor $p \in V$ maintains two local variables:

- $p.Q$: it defines the status of the Question-Answer algorithm at processor p. There are three distinct status: R, W, and A. Status R notifies that p transmits a request to the root of the tree, whereas Status W indicates that p waits for an acknowledgement from the root for the transmitted request. The third status, Status A, indicates that p has received an acknowledgement from the root.

- $p.HQ$: it stores at p the height of the processor which has sent the request.

6.1.2. Algorithm description

To simplify the presentation of the algorithm, consider a forest of allowed trees (i.e., trees rooted at nodes p satisfying Predicate $Allowed(p)$) and a fixed set of requests. In the following, we explain the way our algorithm handles requests focusing on a single tree T of the forest, but this is the same for the other trees since the requests in each tree are handled independently. In the algorithm, the requests sent by nodes of lowest height in the tree are handled in priority.

When a processor p has a *local request* requested by the external algorithm (i.e., $p.Req = ASK$), p can execute $QR\text{-}action$ to set its variables $p.Req, p.Q$, and $p.HQ$ to $WAIT, R$, and to $Height(p)$ respectively, in order to send its request to the root of the tree it belongs to. The external algorithm is informed that the request is sent since $p.Req = WAIT$. Otherwise, an internal processor p in the tree with no local request (i.e., $p.Req \neq REP$) could have to transmit requests from its children (the request from a requesting descendant of lowest height first) in the following cases:

- a child of p given by Ch_p (see Algorithm 2) is sending a request with a highest priority (i.e., $(Ch_p).HQ < p.HQ$);

- the acknowledgement received for the transmitted request is no more needed at p (all its children waiting it have transfered the acknowledgment, see Predicate $Transmit(p)$);

- p is waiting for an acknowledgement for a request and a new request is transmitted by a child of p with the same height (see Predicate $Retransmit(p)$).

In all these above cases, p executes $QRC\text{-}action$ to set $p.Q$ to R and $p.HQ$ to the lowest height among requesting descendant of p (i.e., $p.HQ = (Ch_p).HQ$).

A processor p waits for an acknowledgement for a current request when its parent has transmitted the request (see Predicate $Wait(p)$). Moreover, all p's children transmitting the same request (i.e., with the same height) have to wait for an acknowledgement. Hence, Status W allows to remove bad requests due to an incorrect initial configuration and to synchronize request transmissions of same priority. In this case, p sets its variable $p.Q$ to W using $QW\text{-}action$.

When the root $root(T)$ of the tree T has no local request and is waiting for an acknowledgement for requesting descendant(s) (see Predicate $Answer$), then it executes $QA\text{-}action$ to set its variable $root(T).Q$ to A. This permission is propagated down in the tree to the requesting descendant(s) following the path(s) used to transmit the request. Finally, a processor p waiting for an acknowledgement to a local request (i.e., $p.Q = W$ and $p.Req = WAIT$) executes $QA\text{-}action$ to receive the acknowledgement and sets the shared variable $p.Req$ to REP to notify to the external algorithm of the delivered permission. Note that as soon as a received acknowledgement is no more needed at a processor p (i.e., $p.Req$ is set to OUT by the external algorithm), then another request transmitted by a child of p can be transmitted by p up in the tree.

Algorithm 2 Question-Answer algorithm for any $p \in V$

Inputs: $Neig_p$: set of (locally) ordered neighbors of p;
$Child(p)$: set of neighbors considered as children of p in the tree;
$Allowed(p)$: predicate which indicates if p is able to acknowledge to a request;
$Parent(p)$: parent of p in the tree, equal to a processor $q \in Neig_p$ if $\neg Allowed(p)$ or equal to $\perp$ otherwise ;
$Height(p)$: height of p in the tree;
Shared variable: $p.Req \in \{ASK, WAIT, REP, OUT\}$;
Variables: $p.Q \in \{R, W, A\}$; $p.HQ \in \mathbb{N}$;

Macros:

$RC(p) = \{q \in Child(p) :: q.Q \in \{R, W\}\}$
$PrioRC(p) = \{q \in RC(p) :: \forall t \in RC(p), q.HQ \leq t.HQ\}$
$Ch_p = \min\{q \in PrioRC(p)\}$

Global Predicates:

$Transmit(p) \equiv p.Q = A \wedge (\forall q \in Child(p) :: q.Q = W \Rightarrow q.HQ \neq p.HQ)$
$Retransmit(p) \equiv p.Q = W \wedge (\exists q \in Child(p) :: q.Q = R \wedge q.HQ = p.HQ)$
$Error(p) \equiv p.Q \neq A \wedge [(p.Req \notin \{ASK, WAIT\} \wedge p.HQ = Height(p)) \vee (p.HQ \neq Height(p)$
$\wedge (p.Req \neq REP \Rightarrow (\forall q \in Child(p) :: q.HQ = p.HQ \Rightarrow q.Q = A)))]$
$Request(p) \equiv p.Req = ASK \wedge (|PrioRC(p)| > 0 \Rightarrow Height(p) \leq (Ch_p).HQ)$
$RequestT(p) \equiv p.Req \neq REP$
$\wedge |PrioRC(p)| > 0 \wedge [((Ch_p).HQ \geq p.HQ \Rightarrow Transmit(p)) \vee Retransmit(p)]$

Algorithm:

Predicates:

$Wait(p) \equiv (Allowed(p) \wedge p.Q = R \wedge (\forall q \in Child(p) :: q.HQ = p.HQ \Rightarrow q.Q = W)) \vee$
$(\neg Allowed(p) \wedge Parent(p).Q = R \wedge p.Q = R \wedge Parent(p).HQ = p.HQ$
$\wedge (\forall q \in Child(p) :: q.HQ = p.HQ \Rightarrow q.Q = W))$
$Answer(p) \equiv (Allowed(p) \wedge p.Q = W) \vee$
$(\neg Allowed(p) \wedge Parent(p).Q = A \wedge p.Q = W \wedge Parent(p).HQ = p.HQ)$

Actions:

$QE\text{-}action :: Error(p) \rightarrow p.Q := A; p.HQ := Height(p);$
$QR\text{-}action :: Request(p) \rightarrow p.Q := R; p.HQ := Height(p); p.Req = WAIT;$
$QRC\text{-}action :: RequestT(p) \rightarrow p.Q := R; p.HQ := (Ch_p).HQ;$
if $p.HQ < Height(p) \wedge p.Req = WAIT$
then $p.Req := ASK$; fi
$QW\text{-}action :: Wait(p) \rightarrow p.Q := W;$
$QA\text{-}action :: Answer(p) \rightarrow p.Q := A;$
if $p.Req = WAIT$ then $p.Req := REP$; fi

However, a processor must be able to detect *wrong* requests due to an incorrect initial configuration. A request treated by a processor p is a *wrong request* in the following cases (see Predicate $Error(p)$):

- p is sending a local request whereas it has no local request (i.e., $p.Q \neq A \wedge p.Req \notin \{ASK, WAIT\}$ and $p.HQ = Height(p)$);
- p is transmitting a request from a child, however no child of p has a request with the same height (i.e., $p.Q \neq A \wedge p.HQ \neq Height(p) \wedge (\forall q \in Child(p), q.HQ = p.HQ \Rightarrow q.Q = A)$).

When a processor p detects a wrong request, then p executes $QE\text{-}action$. This action has the highest priority among the actions at p, and it resets p's state like if an acknowledgement to a local request was received, i.e., to set $p.Q$ to A and $p.HQ$ to $Height(p)$ (without changing the state of the shared variable $p.Req$).

A questioning mechanism close to the mechanism presented here was used in [34] to design a snap-stabilizing solution to the problem of Propagation of Information with Feedback (PIF) with a round complexity in $O(n)$ and a step complexity in $O(\Delta n^3)$. However, solving the PIF problem involves a strong synchronization in the network to insure that all the nodes in the network belong

to the same broadcast tree before initiating the feedback phase. Indeed, each time a node is added to the broadcast tree the questioning mechanism is reset leading to a $O(n)$ round complexity. Contrary to this questioning mechanism, here our mechanism needs a weakest synchronization to resolve the Question-Answer problem. Let De be the set of requesting nodes and h_{min} the height of closest requesting nodes from the root in T. The first requests acknowledged by $root(T)$ are the requests from nodes at height h_{min}. Then, if the set of requests is static then the requests at height $h_{min}+1$ are acknowledged by $root(T)$ (if any) and so on. In fact, only a synchronization for the requests of requesting nodes at height h_{min} (whose requests are of highest priority) in tree T is required leading to a round complexity function of the height of T. The transmission of a request requires $O(n)$ steps, however this transmission can be interrupted only by a requesting node with the same height in T, that is at most $|De|$ times.

The following corollary summarizes the above discussion:

Corollary 1. *Let T be an allowed tree and h_{min} be the height of the closest requesting nodes in De from the root in T. In $O(h_{min})$ rounds and $O(n|De|)$ steps, at least one requesting node in De receive an acknowledgement from $root(T)$ to its request.*

7. Composition and complexities

Algorithm $\mathcal{BFS}$ is obtained by the composition of Algorithm 2 and Algorithm 1. These two algorithms are composed together at each processor $p \in V$ with a conditional composition (first introduced in [37]): Algorithm 1 $\circ\ |_{Cond(p)}$ Algorithm 2, where each guard g of the actions of Algorithm 2 at each processor $p \in V$ has the form $Cond(p) \wedge g$ with Predicate $Cond(p)$ defined below (see Algorithm 1 for the description of predicates): $Cond(p) \equiv \quad GoodT(p) \wedge GoodL(p)$.

Using this composition, each processor $p \in V$ can execute Algorithm 2: (i) to transmit requests and acknowledgements only if the tree containing p is locally correct (i.e., Predicate $GoodT(p)$ is satisfied), and (ii) to ask a permission if needed (i.e., Predicate $GoodL(p)$ is satisfied). Indeed, actions in Algorithm 2 can be locked to avoid processors belonging to a tree not rooted at r (abnormal tree) to transmit useless requests since no acknowledgement can be received (only r can deliver acknowledgements). Therefore, processors in abnormal trees can only execute actions in Algorithm 1 to hook on to another tree in the forest via a neighbor with a permission (acknowledgement delivered by Algorithm 2). Moreover, actions of Algorithm 2 and Algorithm 1 can be enabled at p simultaneously. In this case, Algorithm 2 is executed before Algorithm 1 at p.

8. An example of execution

In this section we give an example of execution of Algorithm $\mathcal{BFS}$. To this end, we consider the network with the initial configuration illustrated in Figure 4(a). In Figure 4, the values of the variables of each processor p is given in three parts: the first pair $(p.L, p.S)$ related to Algorithm 1, followed by the shared variable $p.Req$, and a last pair $(p.Q, p.HQ)$ related to Algorithm 2.

In the following, we explain the evolution of the system states by (synchronous) round until reaching a terminal and legitimate configuration illustrated in Figure 4(h):

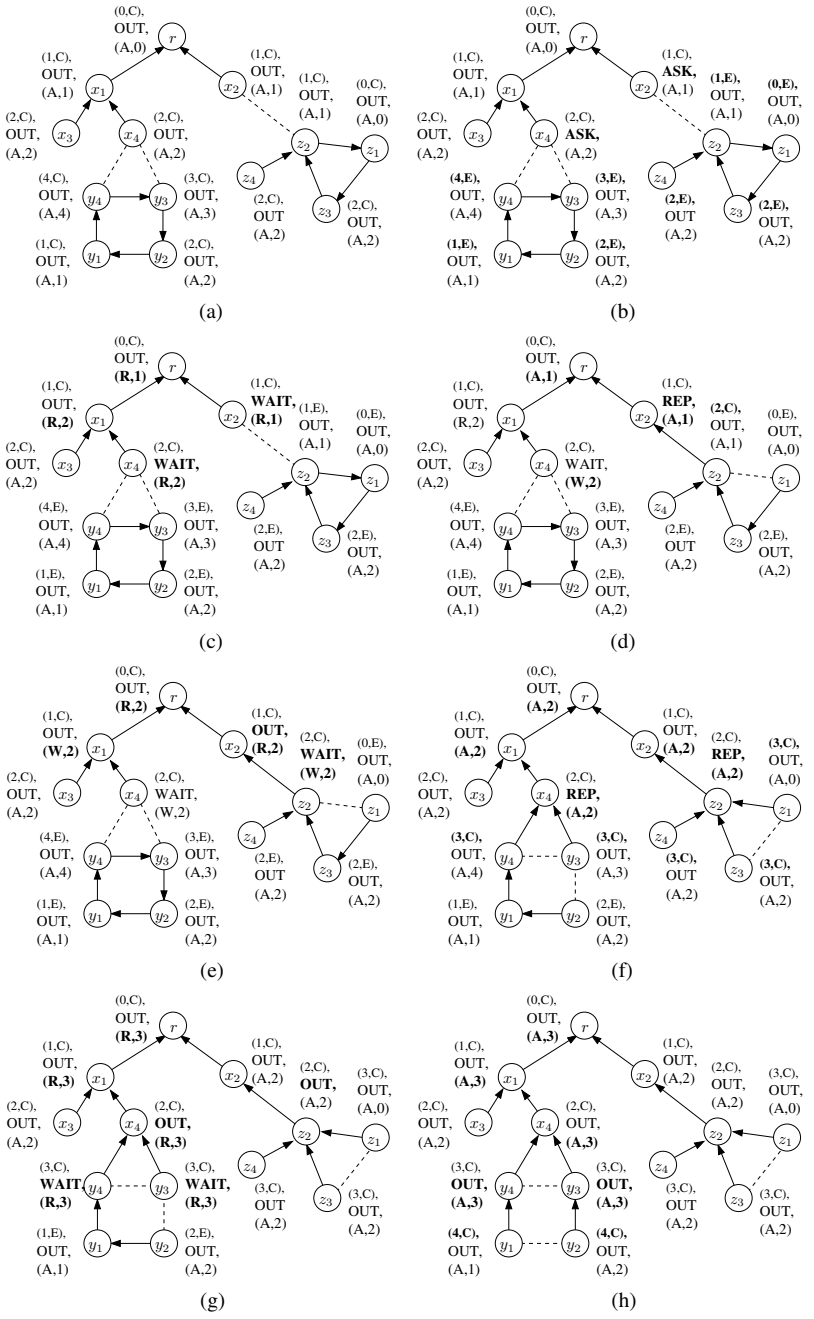

Figure 4: Network considered for the example of execution of Algorithm $\mathcal{BFS}$. For each processor p, the values of the variables are separated in three parts: the first pair $(p.L, p.S)$ related to Algorithm 1, followed by the shared variable $p.Req$, and a last pair $(p.Q, p.HQ)$ related to Algorithm 2. (a) initial configuration with a forest of trees (dotted lines are edges not used in the forest), (b)-(g) several intermediate states with modifications in bold text until reaching the terminal configuration given in (h).

Round 0.

1. First of all, the processors belonging to an abnormal tree are informed by the abnormal root of this situation. So, every processor y_i and $z_i, 1 \leq i \leq 4$, execute E-action of Algorithm 1 to propagate the *Error* status E in the abnormal trees.
2. After this propagation of *Error* status, the processors x_2 and x_4 have neighbors satisfying Predicate $GP\text{-}REP()$. So, these two processors execute A-action of Algorithm 1 to set the shared variable Req to ASK for sending their local request to the questioning mechanism (i.e., Algorithm 2). The configuration obtained after the execution of these actions is illustrated in Figure 4(b).

Round 1.

1. The questioning mechanism sets the shared variable Req to $WAIT$ by the execution of QR-action of Algorithm 2 at x_2 and x_4 to inform these processors that their requests are taken into account. These processors cannot send another request (local or not) until this shared variable has OUT value. Moreover, by the execution of QR-action the variables Q and HQ (related to the sent request) are set to R and 1 for x_2 or 2 for x_4 (level of x_2 and x_4).
2. Then processors x_1 and r transmit the request of highest priority (i.e., with the lowest HQ value) from their descendants by executing QRC-action of Algorithm 2. So, x_1 transmits the request initiated by x_4, while r transmits the request from x_2. The new configuration reached after all these actions is given in Figure 4(c).

Round 2.

1. The processors on the requesting path related to the request initiated by x_2 and x_4 execute QW-action of Algorithm 2 to notice that the request has been transmitted by the parent in the tree. So, the variable Q is set to value W at x_2 and r, this is also the case for x_4.

Round 3.

1. The authorization for x_2's request is given by the root r of the normal tree and is propagated down following the requesting path of x_2's request. So, r and x_2 execute QA-action of Algorithm 2 to set variable Q to value A. Moreover, since x_2 receives an authorization for its local request then the shared variable Req is also set to value REP to inform Algorithm 1 that neighbor processors can be connected to x_2.
2. Processor z_2 executes C-action of Algorithm 1 to hook to x_2 because of the authorization given to x_2. So, z_2 sets its variables S, P and L to C, x_2 and 2 (x_2's level plus one) respectively. The new reached configuration is illustrated in Figure 4(d).

Round 4.

1. The request initiated by x_4 is transmitted to r, since the authorization for x_2's request has been propagated down. So, r executes QRC-action of Algorithm 2 to set variables Q and HQ to R and 2 respectively.
2. The processors on the requesting path related to x_4's request execute QW-action of Algorithm 2 to notice that the request has been transmitted by the parent in the tree.

3. Processor x_2 executes O-action of Algorithm 1 to set its shared variable Req to OUT since all its neighbors are connected (no neighbor satisfying Predicate $GP\text{-}REP()$) and the obtained authorization is no more needed. This allows x_2 to send a new request if needed.
4. Processor z_2 has neighbors which satisfy Predicate $GP\text{-}REP()$. So, it executes A-action of Algorithm 1 to set the shared variable Req to ASK to send a local request to the questioning mechanism.

Round 5.

1. r executes QW-action of Algorithm 2 to notice that x_4's request has been correctly transmitted. So, the variable Q is set to value W at r.
2. The questioning mechanism set the shared variable Req to $WAIT$ by executing QR-action of Algorithm 2 at z_2 to inform Algorithm 1 at z_2 that the local request is taken into account. Moreover, by the execution of QR-action the variables Q and HQ are set to R and 2.
3. x_2 executes QRC-action to transmit the request initiated by z_2. So, variables Q and HQ are set to R and 2 respectively.

Round 6.

1. At this point of the execution, x_4 and z_2 have initiated two requests of same priority (x_4 and z_2 have the same level). A synchronization is done by the questioning mechanism in order to synchronize the connection of processors at layer 3 of the constructed BFS. To this end, the transmission of the request is reseted at all the processors on the common requesting path of x_4's and z_2's request. So, here only r execute QRC-action of Algorithm 2 to set the variable Q to R.
2. z_2 executes QW-action of Algorithm 2 to notice that its local request has been correctly transmitted by its parent in the tree. So, the variable Q is set to value W at z_2. The configuration obtained after the execution of all these actions is given in Figure 4(e).

Round 7.

1. x_2 and r execute QW-action of Algorithm 2 to notice that the request initiated by z_2 has been correctly transmitted by its parent in the tree. So, the variable Q is set to value W at x_2 and r.

Round 8.

1. The authorization for the request of x_4 and z_2 is given by the root r and is propagated down following the requesting path corresponding to the request of x_4 and z_2. So, r, x_1, x_4 execute QA-action of Algorithm 2 to set variable Q to value A for x_4's request, the same is done by x_2 and z_2 for z_2's request. Moreover, since x_4 and z_2 receive an authorization for their local request then the shared variable Req is set to REP at x_4 and z_2 to inform Algorithm 1 that neighbor processors can be connected to x_4 and z_2.
2. Processors z_1, z_3 and z_4 execute C-action of Algorithm 1 to hook to z_2 because of the authorization given to z_2. The same is done at y_3 and y_4 to hook to x_4. So, variables S, P and L are set to C, z_2 (or x_4 accordingly) and 3. The new reached configuration is illustrated in Figure 4(f).

Round 9.

1. Processors x_4 and z_2 execute O-action of Algorithm 1 to set their shared variable Req to OUT since all their neighbors are connected (no neighbor satisfying Predicate GP-$REP()$) and the obtained authorization is no more needed. This allows x_4 and z_2 to send a new request if needed.
2. Processors y_3 and y_4 have neighbors which satisfy Predicate GP-$REP()$. So, A-action of Algorithm 1 is executed to set the shared variable Req to ASK to send a local request to the questioning mechanism.

Round 10.

1. The questioning mechanism set the shared variable Req to $WAIT$ by the execution of QR-action of Algorithm 2 at y_3 and y_4 to inform Algorithm 1 that the local request is taken into account. Moreover, by the execution of QR-action the variables Q and HQ are set to R and 3 at y_3 and y_4.
2. x_4 and x_1 execute QRC-action to transmit the request initiated by y_3 and y_4 of same priority. So, variables Q and HQ are set to R and 3 respectively. The new obtained configuration is given in Figure 4(g).

Round 11.

1. y_3, y_4, x_4, x_1 and r execute QW-action of Algorithm 2 to notice that the requests initiated by y_3 and y_4 have been correctly transmitted by their parent in the tree. So, the variable Q is set to value W.

Round 12.

1. The authorization for the request of y_3 and y_4 is given by the root r and is propagated down following the requesting path corresponding to the request of y_3 and y_4. So, r, x_1, x_4, y_3 and y_4 execute QA-action of Algorithm 2 to set variable Q to value A. Moreover, since y_3 and y_4 receive an authorization for their local request then the shared variable Req is set to REP at y_3 and y_4 to inform Algorithm 1 that neighbors can be connected to y_3 and y_4.
2. Processor y_2 executes C-action of Algorithm 1 to hook to y_3 because of the authorization given to y_3. The same is done at y_1 to hook to y_4. So, variables S, P and L are set to C, y_3 (or y_4 accordingly) and 4.

Round 13.

1. Finally, processors y_3 and y_4 execute O-action of Algorithm 1 to set their shared variable Req to OUT since all their neighbors are connected (no neighbor satisfying Predicate GP-$REP()$) and the obtained authorization is no more needed. The new and legitimate configuration reached by the system is illustrated in Figure 4(h).

9. Correctness proof

In this section, we show the correctness of the two algorithms proposed in this paper. In a first part, we prove that the Question-Answer algorithm (Algorithm 2) is stabilizing under a weakly fair daemon (Section 9.1.2), and then its correctness under an unfair daemon by showing a polynomial step complexity (Section 9.1.3). In a second part, we consider the BFS algorithm (Algorithm 1). We show its correctness under a weakly fair daemon (Section 9.2.2), and then under an unfair daemon (Section 9.2.3).

9.1. Proof of Question-Answer algorithm

In the following proofs, we consider a more general context than the one used to present Algorithm 2 for the Question-Answer problem. So $Height(p)$ could be replaced by any priority which can be independent from the height of a requesting processor p (which is a particular case).

To prove the correctness of the Question-Answer algorithm (Algorithm 2), we first introduce some useful definitions, and we show that in an illegitimate configuration there is always at least one enabled processor (Theorem 1). Then, we establish that in at most $O(k+1)$ rounds the request and the corresponding answer of every processor at height k in the normal tree are transmitted by the Question-Answer algorithm (Lemmas 8, 9, and 10). We show that the Question-Answer algorithm is a silent algorithm (Lemma 11), which allows us to conclude that the Question-Answer algorithm satisfies Specification 2 under a weakly fair daemon (Lemma 12). Finally, we consider an unfair daemon and given a set of local requests from processors belonging to the normal tree T, we prove that these requests and their corresponding answers are transmitted in at most $O(n^3)$ steps (Corollary 4).

9.1.1. Definitions

Definition 4 (Path). *The sequence of processors $\mathcal{P}(x,y) =< p_0 = x, p_1, \ldots, p_k = y >$ is called a* path *if $\forall i, 1 \leq i \leq k, Parent(p_i) = p_{i-1}$. The processors p_0 and p_k are termed as the* extremities *of $\mathcal{P}$. The length of $\mathcal{P}$ is noted $|\mathcal{P}| = k$.*

Definition 5 (Allowed tree). *A tree T rooted at processor p such that $(p = root(T) \wedge Allowed(p))$ is called an* allowed tree. *Any tree T' rooted at processor q such that $(q = root(T') \wedge \neg Allowed(q)$ is called a* not allowed tree.

In the following, we consider a static forest $\mathcal{F}$ of trees constructed in the network $G = (V, E)$.

Definition 6 (Request priority). *Given a tree T in forest $\mathcal{F}$ and k processors in T sending a request. Let $R = \{R_{p_1}, \ldots, R_{p_k}\}$ be the set of requests sent by processors $p_i, 1 \leq i \leq k$, in T. A request R_{p_i} has a* higher priority *than request $R_{p_j}, 1 \leq i, j \leq k$, if $Height(p_i) < Height(p_j)$. A request R_{p_i} sent (or transmitted) by a processor $p \in T$ is of* highest priority in the neighborhood *of p if $\forall q \in Neig_p \backslash \{Parent(p)\}$ the request R_{p_j} sent (or transmitted) by q we have $Height(p_j) > Height(p_i)$.*

In the reminder, we make the hypothesis that the extern algorithm (Algorithm $\mathcal{A}$) sets in finite time the shared variable $p.Req$ from REP to OUT when a permission delivered at p is no more needed.

9.1.2. Proof assuming a weakly fair daemon

The following theorem proves that any execution of Question-Answer algorithm is deadlock-free.

Theorem 1. *Let the set of configurations $\mathcal{B} \subseteq \mathcal{C}$ such that there is at least one processor $p \in V$ in an allowed tree which has a request to send or has sent a request and it does not receive an acknowledgement in every configuration $\gamma \in \mathcal{B}$. $\forall \gamma \in \mathcal{B}, \exists q \in V$ such that q is enabled in γ.*

Proof. Assume, by the contradiction, that $\exists \gamma \in \mathcal{B}$ such that $\forall q \in V$ no action is enabled at q in γ. Assume then that there exists at least one allowed tree T in γ in which $\exists p \in T$ such that $p.Req = ASK$. Consider the processor $p \in T$ with the request of highest priority in T, i.e.,

$(\forall x \in T :: x.Req = ASK \wedge Height(p) < Height(x))$. In this case, either $p.Req = ASK$ and $QR\text{-}action$ is enabled at p, a contradiction, or $\exists q \in \mathcal{P}(root(T), p)$ such that $q.HQ \neq p.HQ$. Moreover, since p's request is of highest priority in T then q satisfies $p.HQ = (Ch_q).HQ$ and $|PrioRC(q)| > 0$. We assume that $p.Req \neq REP$, otherwise by hypothesis $p.Req$ is set to OUT in finite time. In this case, either we have $(p.HQ < q.HQ \Rightarrow RequestT(q))$ and $QRC\text{-}action$ is enabled at q, a contradiction. Otherwise, q has transmitted a request with the same priority, i.e., we have $p.HQ = q.HQ$ and $q.Q = W$ (see Predicate $Retransmit(q)$), and $QRC\text{-}action$ is enabled at q, a contradiction. The execution of $QR\text{-}action$ sets $p.Req$ to $WAIT$. Hence, by contradiction, $p.Q = R, p.HQ = Height(p)$ and $p.Req = WAIT$ at p and $\forall q \in \mathcal{P}(root(T), p), q.HQ = p.HQ$. If $\exists q \in \mathcal{P}(root(T), p)$ such that $q.Q = W$ then $QRC\text{-}action$ is enabled at q (see Predicate $Retransmit(q)$), a contradiction. Thus, $\forall x \in \mathcal{P}(root(T), p), x.HQ = p.HQ \wedge x.Q = R$. Then, we have $(Parent(p).HQ = p.HQ \wedge p.HQ = Height(p)) \Rightarrow Wait(p)$ and $QW\text{-}action$ is enabled at p, a contradiction. If $\exists q \in \mathcal{P}(root(T), p)$ such that $q.Q = R \wedge (\exists s \in Child(q) :: s.Q = W)$ then $QW\text{-}action$ is enabled at q, a contradiction. Hence, by contradiction, $\forall x \in \mathcal{P}(root(T), p), x.HQ = p.HQ \wedge x.Q = W$. Thus, $QA\text{-}action$ is enabled at $root(T)$, a contradiction. If $\exists q \in \mathcal{P}(root(T), p)$ such that $Parent(q).Q = A \wedge q.Q = W$ then $QA\text{-}action$ is enabled at q, a contradiction. □

Lemma 1. *Let an allowed tree T in a static forest $\mathcal{F}$. After executing $QE\text{-}action$ at a processor $p \in T$, $QE\text{-}action$ is disabled at p until p sends or transmits another request.*

Proof. Assume, by the contradiction, that $QE\text{-}action$ is enabled at a processor $p \in T$ before p sends or transmits another request. After the first execution of $QE\text{-}action$, we have $p.Q = A$ at p. If p can execute $QE\text{-}action$ again then this implies that we have $p.Q \neq A$ (because $(p.Q = A \Rightarrow \neg Error(p))$). Since we assume that p does not execute $QR\text{-}action$ and $QRC\text{-}action$, then this implies that $p.Q = W$ obtained by executing $QW\text{-}action$ at p, a contradiction because $Wait(p) \Rightarrow p.Q = R$ at p. □

Lemma 2. *Let an allowed tree T in a static forest $\mathcal{F}$. When $QR\text{-}action$ is enabled at processor $p \in T$, it remains enabled until p executes it and p remains in T.*

Proof. Let $\gamma \mapsto \gamma'$ be a step. Assume, by the contradiction, that $QR\text{-}action$ is enabled at p in γ and not in γ' (i.e., $\neg Request(p)$ in γ') but p did not execute $QR\text{-}action$ in $\gamma \mapsto \gamma'$. According to the hypothesis of the lemma, we assume that p has no child with a request of priority higher than p's request (i.e., $Height(p) \leq (Ch_p).HQ$). $QR\text{-}action$ is the enabled action at p which has the highest priority, otherwise according to Lemma 1 after executing $QE\text{-}action$ then it is disabled at p. Moreover, we assume that p remains in T in γ', so $p.Req = ASK$ in γ'. Since p did not move in $\gamma \mapsto \gamma'$, we have $p.HQ \neq Height(p)$. Thus, $Request(p)$ is satisfied in γ', a contradiction. □

Lemma 3. *Let any allowed tree T in a static forest $\mathcal{F}$. Every processor $p \in T$ transmits the request with highest priority in its neighborhood.*

Proof. According to formal description of Algorithm 2, to transmit a request a processor executes $QR\text{-}action$ or $QRC\text{-}action$. Assume, by the contradiction, that there is a processor $p \in T$ which does not transmit a request. That is, $QR\text{-}action$ and $QRC\text{-}action$ are disabled or they are not the enabled actions of highest priority at p.

We first show that $QR\text{-}action$ and $QRC\text{-}action$ are enabled at p. We must consider two cases: p has a local request to send with a priority higher than its children requests or p has a

request from a child to transmit of highest priority. If p has a local request to send then $p.Req \in \{ASK, WAIT\}$. Since $QR\text{-}action$ is not enabled at p, this implies that $p.Req = WAIT$ and p has already sent its request, a contradiction. In first case, p's request has the highest priority in p's neighborhood (i.e., $|PrioRC(p)| = 0$ or $Height(p) \leq (Ch_p).HQ$). So $QR\text{-}action$ is enabled at p, a contradiction. Otherwise, p has a child request with a priority higher than the priority of its local request (i.e., we have $p.Req \neq REP$ and $|PrioRC(p)| > 0$). Consider the child q of p such that $Ch_p = q$. We have that $QR\text{-}action$ is disabled and by contradiction $QRC\text{-}action$ is not enabled at p. Thus, to have $\neg RequestT(p)$ this implies we have $(Ch_p).HQ \geq p.HQ$ and we must consider two sub-cases at p: $p.Q \neq A$ or $\exists s \in Child(p)$ such that $s.Q = W \wedge s.HQ = p.HQ$. Either $p.Q \neq A$ then this implies that $(Ch_p).HQ = p.HQ$ and p has already transmitted the request of q, a contradiction. Or $\exists s \in Child(p)$ such that $s.Q = W \wedge s.HQ = p.HQ$. This implies that either $s = q$ and p has already transmitted q's request or $s \neq q$ and $s.HQ < q.HQ$, a contradiction because $(Ch_p) = q$. Thus, $QR\text{-}action$ or $QRC\text{-}action$ is enabled at every processor $p \in T$ which has a local request or a request from a child to transmit of highest priority.

We must show that $QR\text{-}action$ or $QRC\text{-}action$ is the enabled action of highest priority for every processor $p \in T$ which has a local request or a request from a child to transmit of highest priority. If $QR\text{-}action$ or $QRC\text{-}action$ are not the action of highest priority at p then this implies that $QE\text{-}action$ is always enabled. According to Lemma 1, after executing $QE\text{-}action$ it is not enabled at p (unless $QR\text{-}action$ or $QRC\text{-}action$ is executed), a contradiction. So, $QE\text{-}action$ is disabled at p. According to Lemma 2, $QR\text{-}action$ is enabled until it is executed at every processor $p \in T$ having a request of priority higher than its children requests (i.e., $Height(p) \leq (Ch_p).HQ$). Otherwise, we have $QR\text{-}action$ is disabled. Therefore, since $QE\text{-}action$ and $QR\text{-}action$ are disabled then $QRC\text{-}action$ is the enabled action of highest priority for every processor $p \in T$ which has a request of highest priority from a child to transmit. □

Corollary 2. *Let an allowed tree T in a static forest $\mathcal{F}$. The request with highest priority in T is transmitted to $root(T)$.*

Lemma 4. *Let any allowed tree T in a static forest $\mathcal{F}$. Every processor $p \in T$ waits for an acknowledgement if p's parent transmits the request of highest priority in p's neighborhood.*

Proof. According to formal description of Algorithm 2, to wait for an acknowledgement to a request a processor executes $QW\text{-}action$. Assume, by the contradiction, that there is a processor $p \in T$ which does not wait for an acknowledgement while p transmits the request of highest priority in its neighborhood. That is, $QW\text{-}action$ is disabled or it is not the enabled action of highest priority at p.

We first show that $QW\text{-}action$ is enabled at p. According to Lemma 3, for processor p we have $p.Q = R$, $p.HQ = Height(p)$, and $p.Req = WAIT$ if p has sent a local request, or $p.Q = R$, $p.HQ \neq Height(p)$, and $p.Req \neq REP$ otherwise. We must consider two cases: p's parent has not transmitted p's request or there is a child of p with a request of same priority which is not waiting for the acknowledgement (i.e., $\neg[Parent(p).Q = R \wedge Parent(p).HQ = p.HQ]$ or $\exists q \in Child(p)$ such that $q.HQ = p.HQ \wedge q.Q \neq W$). Note that for $root(T)$ only the second case must be considered. If $\neg[Parent(p).Q = R \wedge Parent(p).HQ = p.HQ]$ then this implies that the request transmitted by p's parent is not the request of highest priority in the neighborhood of p's parent (since its parent has transmitted another request), a contradiction with assumption of lemma to prove. Otherwise, $\exists q \in Child(p)$ such that $q.HQ = p.HQ \wedge q.Q \neq W$. Then there is a path $\mathcal{P}(p, s)$ in T such that $x.HQ = Height(s) = q.LQ$ for every processor $x \in \mathcal{P}(p, s)$.

Moreover, there is a processor $y \in \mathcal{P}(p, s)$ such that $y.Q = W$ and $Parent(y).Q = R$. Thus, $QW\text{-}action$ is enabled at $Parent(y)$ from the first case. By induction on the length of path $\mathcal{P}(p, s)$ when every processor x has executed $QW\text{-}action$ then $q.Q = W$, a contradiction.

We must show that $QW\text{-}action$ is the enabled action of highest priority for every processor which transmits the request of highest priority in its neighborhood also transmitted by their parent. Assume, by the contradiction, that $QW\text{-}action$ is not the enabled action of highest priority at p. Suppose that $QE\text{-}action$ is the enabled action of highest priority at p. According to Lemma 1, after executing $QE\text{-}action$ it is not enabled at p, a contradiction. So, $QE\text{-}action$ is disabled at p. Suppose that $QR\text{-}action$ or $QRC\text{-}action$ is enabled at p, a contradiction because we assume that p has transmitted the request of highest priority in its neighborhood (i.e., $((p.Req = WAIT \wedge p.Q \neq A) \Rightarrow \neg Request(p))$ or $((p.Req \neq REP \wedge p.Q \neq A) \Rightarrow \neg RequestT(p))$). □

Corollary 3. *Let an allowed tree T in a static forest $\mathcal{F}$. $root(T)$ waits for an acknowledgement for the request of highest priority in T.*

Lemma 5. *Let an allowed tree T in a static forest $\mathcal{F}$. A processor $p \in T$ waiting for an acknowledgement to a transmitted request transmits again the request of a child with the same priority, if it is the request of highest priority in p's neighborhood.*

Proof. According to formal description of Algorithm 2, a processor $p \in T$ waiting for an acknowledgement executes $QRC\text{-}action$ to transmit again a request from a child with the same priority.

As there is a child request of highest priority in p's neighborhood transmitted by p, then we have $p.Req \neq REP \wedge |PrioRC(p)| > 0$. Assume, by the contradiction, that p does not execute $QRC\text{-}action$ to transmit again the request with the same priority. Either for every child q of p we have $q.Q \neq R$ or $q.HQ \neq p.HQ$ because $(\forall q \in Child(p) :: q.Q \neq R \vee q.HQ \neq p.HQ) \Rightarrow \neg Retransmit(p)$. Either $q.Q \neq R$ then q has no request to transmit because its request was already transmitted (i.e., $q.Q = W$) or an acknowledgement was received (i.e., $q.Q = A$), a contradiction. Or $q.HQ \neq p.HQ$ then the request to transmit again by p is not of highest priority in p's neighborhood (i.e., $(Ch_p).HQ \neq q.HQ$), a contradiction with the hypothesis of the lemma. □

Lemma 6. *Let any allowed tree T in a static forest $\mathcal{F}$ and a processor $s \in T$ sending the request of highest priority in T. Every processor $p \in \mathcal{P}(root(T), s)$ transmits the acknowledgement to the request of s.*

Proof. According to formal description of Algorithm 2, to transmit the acknowledgement to a request a processor executes $QA\text{-}action$. Assume, by the contradiction, that there is a processor $p \in \mathcal{P}(root(T), s)$ which does not transmit the acknowledgement to the request of s. That is, $QA\text{-}action$ is disabled or it is not the enabled action of highest priority at p.

We first show that $QA\text{-}action$ is enabled at p. According to Lemma 4, for processor p we have $p.Q = W$, $p.HQ = Height(s)$, and $p.Req = WAIT$ if $p = s$, or $p.Q = W$, $p.HQ = Height(s) \neq Height(p)$ and $p.Req \neq REP$ otherwise. We must consider two cases: $p = root(T)$ or $p \neq root(T)$. Consider processor $root(T)$, if $QA\text{-}action$ is disabled then this implies that $root(T).Q \neq W$, a contradiction with the assumption that for every processor $q \in \mathcal{P}(root(T), s)$ we have $q.Q = W$. Now, $p \neq root(T)$. Consider p is the child of $root(T)$ such that $p \in \mathcal{P}(root(T), s)$. If $QA\text{-}action$ is disabled at p then either $Parent(p).Q \neq A$ or

$Parent(p).HQ \neq p.HQ$ or $p.Q \neq W$, a contradiction because $root(T).Q = A$ from first case and we assume for every processor $q \in \mathcal{P}(root(T), s)$ we have $p.Q = W$, $p.HQ = Height(s)$. Otherwise, $p \neq root(T)$ and p is not the child of $root(T)$. By induction on the length of path $\mathcal{P}(root(T), s)$, the arguments used for processor p can be applied for every processor $q \in \mathcal{P}(root(T), s)$. Thus, $QA\text{-}action$ is enabled for every processor $q \in \mathcal{P}(root(T), s)$.

We must show that $QA\text{-}action$ is the enabled action of highest priority for every processor $p \in \mathcal{P}(root(T), s)$. Assume, by the contradiction, that $QA\text{-}action$ is not the enabled action of highest priority at p. According to Lemma 1, after executing $QE\text{-}action$ it is not enabled at p, a contradiction. So, $QE\text{-}action$ is disabled at p. Suppose that $QR\text{-}action$ or $QRC\text{-}action$ is enabled at p, a contradiction because we assume that p has transmitted the request of highest priority in its neighborhood (i.e., $((p.Req = WAIT \wedge p.Q \neq A) \Rightarrow \neg Request(p))$ or $((p.Req \neq REP \wedge p.Q \neq A) \Rightarrow \neg RequestT(p))$). Suppose that $QW\text{-}action$ is enabled at p, a contradiction because $p.Q \neq R$. □

Lemma 7. *Let an allowed tree T in a static forest $\mathcal{F}$. Between the reception of two acknowledgements to a request, every processor $p \in T$ has sent a new request.*

Proof. According to formal description of Algorithm 2, to receive an acknowledgement to a request in an allowed tree T a processor executes $QA\text{-}action$. Assume, by the contradiction, that there is a processor $p \in T$ which receives two acknowledgements for the same request. That is, $QR\text{-}action$ and $QRC\text{-}action$ are not executed by p between two consecutive executions of $QA\text{-}action$.

After the first execution of $QA\text{-}action$ by p, we have $p.Q = A$ in configuration γ_i. To execute $QA\text{-}action$ in step $\gamma_{j-1} \mapsto \gamma_j$, with $i < j$, this implies we had $p.Q = W$ in γ_{j-1} because $Answer(p) \Rightarrow p.Q = W$. Thus, $QW\text{-}action$ was executed in step $\gamma_{j-2} \mapsto \gamma_{j-1}$. However, to execute $QW\text{-}action$ in step $\gamma_{j-2} \mapsto \gamma_{j-1}$ this implies we had $p.Q = R$ in γ_{j-2} because $Wait(p) \Rightarrow p.Q = R$. So, by formal description of Algorithm 2 $QR\text{-}action$ or $QRC\text{-}action$ was executed in step $\gamma_{j-3} \mapsto \gamma_{j-2}$, with $i < j - 3$, a contradiction. □

Lemma 8. *Let an allowed tree T in a static forest $\mathcal{F}$ and a processor $p \in T$ at height k with a local request of highest priority in T. From any configuration, in at most $k+1$ rounds p's request is transmitted to $root(T)$.*

Proof. We show by induction the following proposition: If at height less than k in T there is no processor $q \in T$ such that $QR\text{-}action$ is enabled at q and $\exists p \in T$ at height k such that $p.Req = ASK$, then in at most $j+1$ rounds we have $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.HQ = Height(p)$ at height $\geq k - j$ in T.
In the base case $j = 0$ and we consider p. According to Lemma 2, if $Request(p)$ is satisfied at p then p executes $QR\text{-}action$ and we have $p.Q = R$ and $p.HQ = Height(p)$ at p. Consider that in first configuration of round 0 p satisfies $Error(p)$, then p can execute $QE\text{-}action$ and as the daemon is weakly fair at the end of round 0 we have $p.Q = A$ and $p.HQ = Height(p)$. At the first configuration of round 1, p satisfies $Request(p)$ and it can execute $QR\text{-}action$. Since the daemon is weakly fair, thus the proposition is verified because at the last configuration of round 1 we have $p.Q = R$ and $p.HQ = Height(p)$ at p.
Induction case: We assume that in round $j = k - 1$ the proposition is true for any processor at height h, $k - j \leq h \leq k$ in $\mathcal{P}(root(T), p)$. We have to show that if at height less than k in T there is no processor $q \in T$ such that $QR\text{-}action$ is enabled at q, then in round $j + 1$ for any processor $q \in \mathcal{P}(root(T), p)$ at height h, $k - (j + 1) \leq h \leq k$, we have $q.Q = R \wedge q.HQ =$

$Height(p)$. So, we consider the processor $x \in \mathcal{P}(root(T), p)$ at height $k - (j + 1)$ in T. If $QE\text{-}action$ is enabled at x in the beginning of round j then as the daemon is weakly fair we have $(x.Q = A \wedge x.HQ = Height(x)) \Rightarrow \neg Error(x)$ at the first configuration of round $j + 1$. Since there is no processor $s \in T, s \neq p$ at height lower than k such that $QR\text{-}action$ is enabled at s, then $|PrioRC(x)| > 0$ and $Ch_x = q$ such that $q.Q = R$ and $q.HQ = Height(p)$. Either $Height(p) < x.HQ$, then $QRC\text{-}action$ is enabled at x in round $j+1$. Or $Height(p) \geq x.HQ$, then as the daemon is weakly fair we have $(\forall s \in Child(x) :: s.Q = W \Rightarrow s.HQ \neq x.HQ)$, so $Transmit(x)$ is satisfied (remind that x has no request to send so $x.Req \neq REP$ and $x.Q = A$) and $QRC\text{-}action$ is enabled at x in round $j + 1$. In all the above cases, as the daemon is weakly fair in the last configuration of round $j + 1$ so we have $x.Q = R$ and $x.HQ = Height(p)$ at $x \in \mathcal{P}(root(T), p)$, which verifies the proposition. Therefore, since $|\mathcal{P}(root(T), p)| = k$ in at most $k + 1$ rounds we have $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.HQ = Height(p)$. □

Lemma 9. *Let an allowed tree T in a static forest $\mathcal{F}$ and a processor $p \in T$ at height k with a local request of highest priority in T transmitted to $root(T)$. In at most $k + 1$ additional rounds, every processor $q \in T$ waits for an acknowledgement if q transmits p's request.*

Proof. According to Lemma 8, since $p.Req = WAIT \wedge p.Q = R \wedge p.HQ = Height(p)$ at processor $p \in T$ at height k then in at most $k + 1$ rounds we have $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.HQ = Height(p)$.

We show by induction the following proposition: If at height less than k in T there is no processor $q \in T$ such that $QR\text{-}action$ is enabled at q, and $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.HQ = Height(p)$, then in at most $j + 1$ rounds we have $\forall q \in \mathcal{P}(root(T), p), q.Q = W \wedge q.HQ = Height(p)$ at height $\geq k - j$ in T.
In the base case $j = 0$ and we consider p. We have $(\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.HQ = Height(p))$, in particular for $Parent(p)$ and p. Thus, the proposition is verified for p because $QW\text{-}action$ is enabled at p in round 0, and in the first configuration of round 1 we have $p.Q = W$ and $p.HQ = Height(p)$ at p (since the daemon is weakly fair).
Induction case: We assume that in round $j = k - 1$ the proposition is true for any processor at height h, $k - j \leq h \leq k$ in $\mathcal{P}(root(T), p)$. We have to show that if at height less than k in T there is no processor $q \in T$ such that $QR\text{-}action$ is enabled at q, then in round $j + 1$ for any processor $q \in \mathcal{P}(root(T), p)$ at height h, $k - (j + 1) \leq h \leq k$, we have $q.Q = W \wedge q.HQ = Height(p)$. By induction hypothesis, in the first configuration of round $j + 1$ we have for any processor $s \in \mathcal{P}(root(T), p)$ at height $\geq j$ we have $s.Q = W \wedge s.HQ = Height(p)$. Thus, $\exists s \in Child(q), s.Q = W \wedge s.HQ = q.HQ$, and $q.Q = R$ so $QW\text{-}action$ is enabled at q in round $j + 1$. So, since the daemon is weakly fair we have $q.Q = W$ and $q.HQ = Height(p)$ at q, in the last configuration of round $j + 1$, which verifies the proposition. Therefore, since $|\mathcal{P}(root(T), p)| = k$ in at most $k + 1$ additional rounds we have $\forall q \in \mathcal{P}(root(T), p), q.Q = W \wedge q.HQ = Height(p)$. □

Lemma 10. *Let an allowed tree T in a static forest $\mathcal{F}$ and a processor $p \in T$ at height k with a local request of highest priority in T transmitted to $root(T)$. In at most $k + 1$ additional rounds, every processor $q \in T$ transmits the acknowledgement to p's request if q has transmitted p's request.*

Proof. According to Lemmas 8 and 9, in at most $2(k + 1)$ rounds we have $\forall q \in \mathcal{P}(root(T), p)$, $q.Q = W \wedge q.HQ = Height(p)$.

We show by induction the following proposition: If at height less than k in T there is no processor $q \in T$ such that $QR\text{-}action$ is enabled at q, and $\forall q \in \mathcal{P}(root(T), p), q.Q = W \wedge q.HQ = Height(p)$, then in at most $j+1$ rounds we have $x.Q = A \wedge x.HQ = Height(p)$ at processor $x \in \mathcal{P}(root(T), p)$ of height $\leq j$ in T.
In the base case $j = 0$ and we consider $x = root(T)$. We have $(\forall q \in \mathcal{P}(root(T), p), q.Q = W \wedge q.HQ = Height(p))$, in particular for $root(T)$. The proposition is verified for x because we have $(x.Q = W \Rightarrow Answer(x))$ and $QA\text{-}action$ is enabled at x in round 0. Thus, in the first configuration of round 1 we have $x.Q = W \wedge x.HQ = Height(p)$ at x (since the daemon is weakly fair).
Induction case: We assume that in round j the proposition is true for every processor at height $\leq j$ in $\mathcal{P}(root(T), p)$. We have to show that if at height less than k in T there is no processor $q \in T$ such that $QR\text{-}action$ is enabled at q, then in round $j+1$ for processor $x \in \mathcal{P}(root(T), p)$ at height $j+1$, we have $x.Q = A \wedge x.HQ = Height(p)$. By induction hypothesis, in the first configuration of round $j+1$ we have $Parent(x).Q = A \wedge Parent(x).HQ = x.HQ \wedge x.Q = W$ at x, so $QA\text{-}action$ is enabled at x in round $j+1$. Therefore, since the daemon is weakly fair we have $x.Q = A$ and $x.HQ = Height(p)$ at x, in the first configuration of round $j+1$ which verifies the proposition. Moreover, we have $|\mathcal{P}(root(T), p)| = k$ because p is at height k in T. According to formal description of Algorithm 2, if $x.HQ = Height(x)$ when $QA\text{-}action$ is executed at x then we have $x.Req = REP$. So we have $x = p$, and in most $k+1$ additional rounds we have $p.Req = REP \wedge p.Q = A \wedge p.HQ = Height(p)$. □

Lemma 11. *Let the set of configurations $\mathcal{B} \subseteq \mathcal{C}$ such that in every $\gamma \in \mathcal{B}$ there is no request and every processor $p \in V$ has received an acknowledgement. In every configuration $\gamma \in \mathcal{B}$, for every processor $p \in V$ no action of Algorithm 2 is enabled.*

Proof. Since there is no request in γ then for every processor $p \in V$ we have $p.Req \neq ASK$ and $p.Req \neq WAIT$. Moreover, observe that according to formal description of Algorithm 2 for every processor $p \in V$ we have $p.Q \neq A$ either when $p.Req = WAIT$ or when $p.Req = OUT$ or $p.Req = ASK$ with a descendant x of p such that $x.Req = WAIT$. However, as $\forall p \in V, p.Req \neq ASK$ and therefore $p.Req \neq WAIT$ this implies we have $\forall p \in V, p.Q = A$ in γ.

Assume, by the contradiction, that $\exists \gamma \in \mathcal{B}$ such that $\exists p \in V$ with an enabled action of Algorithm 2. If $QE\text{-}action$ is enabled at p then this implies that $p.Q \neq A$, a contradiction. If $QR\text{-}action$ is enabled at p then this implies that $p.Req = ASK$, a contradiction since $\forall p \in V, p.Req \neq ASK$. If $QRC\text{-}action$ is enabled at p then there is a child q of p such that $q.Q \neq A$ (i.e., $|PrioRC(p)| > 0$), a contradiction because $((\forall p \in V, p.Q = A) \Rightarrow |PrioRC(p)| = 0)$. If $QW\text{-}action$ is enabled at p then this implies that $p.Q = R$, a contradiction because $\forall p \in V, p.Q = A$. If $QA\text{-}action$ is enabled at p then this implies that $p.Q = W$, a contradiction because $\forall p \in V, p.Q = A$. □

Lemma 12. *Let a tree T in a static forest $\mathcal{F}$. From any configuration where a processor $p \in T$ executes $QR\text{-}action$, the execution satisfies Specification 2.*

Proof. We have to show that starting from any configuration the execution of Algorithm 2 verifies all the properties of Specification 2.

We first show that Property (Liveness 1) of Specification 2 is satisfied. Let an allowed tree T in a static forest $\mathcal{F}$. From any configuration according to Lemmas 2 and 8 a processor in T which has a local request of highest priority in T sends this request to $root(T)$ in finite time

with Algorithm 2. Assume, by the contradiction, that there is a processor $p \in T$ which has infinitely often a request to send but it can not send its request to $root(T)$, although there are a finite number of requests sent in T. This implies either that an infinite time is needed to send a request from p to $root(T)$, a contradiction with Lemmas 2 and 8, or the request sent by p is never the request of highest priority in T, a contradiction with the hypothesis of a finite number of requests sent in T. This satisfies Property (Liveness 1) of Specification 2.

We now show that Property (Liveness 2) of Specification 2 is satisfied. Let a processor $p \in T$ which has sent a request in an allowed tree T and waits for the acknowledgement to its request. According to Theorem 1, the execution of Algorithm 2 is not done. Moreover, by Lemma 6 a processor which has sent a request with highest priority in T receives an acknowledgement from $root(T)$ in finite time. Thus, at least one processor receives an acknowledgement from $root(T)$ in a finite time, the processor waiting for the acknowledgement to the request of highest priority in T. This satisfies Property (Liveness 2) of Specification 2.

We now show that Property (Safety 1) of Specification 2 is satisfied. According to Lemma 6, a processor p which has sent a local request in an allowed tree T receives at least one acknowledgement to its request. Moreover, by Lemma 7 a processor p receives at most one acknowledgement to a sent request. This satisfies Property (Safety 1) of Specification 2.

We now show that Property (Safety 2) of Specification 2 is satisfied. Assume, by the contradiction, that there is a processor p sending a request in a not allowed tree T which receives an acknowledgement from $root(T)$. Since $root(T)$ is the root of a not allowed tree, we have $\neg Allowed(root(T))$ and $Parent(root(T)) \in Neig_{root(T)}$. So, there is a cycle in T because every processor in T has a parent. Moreover, if p receives an acknowledgement from $root(T)$ then $root(T)$ can execute $QA\text{-}action$. This implies that $Parent(root(T)).Q = A$ because $Answer(p) \Rightarrow Parent(root(T)).Q = A$. So, either $root(T).Q = R$ or $root(T).Q = W \wedge root(T).HQ \neq Parent(root(T)).HQ$ then $Parent(root(T))$ executes $QRC\text{-}action$ (because $Transmit(Parent(root(T))) \Rightarrow RequestT(Parent(root(T)))$), a contradiction. Otherwise, we have $Parent(root(T)).Q = A \wedge (\forall q \in Child(Parent(root(T))), q.Q = W \wedge q.HQ = Parent(root(T)).HQ)$ given by an initial configuration of the system, a contradiction. This satisfies Property (Safety 2) of Specification 2. □

By Theorem 1 and Lemmas 11 and 12, the result below follows:

Theorem 2. *Algorithm 2 is snap-stabilizing for Specification 2 under a weakly fair daemon.*

9.1.3. Proof assuming an unfair daemon

Lemma 13. *Let any allowed tree T in a static forest $\mathcal{F}$ and any processor $p \in T$ with a local request of highest priority in T. If there is no new request with higher or equal priority than p's request in T, then p's request is transmitted to $root(T)$ in at most $2n$ steps, with n the number of processors in the network.*

Proof. According to Lemma 3, if there is no new request with higher or equal priority than p's request in T then every processor $q \in \mathcal{P}(root(T), p)\backslash\{p\}$ executes $QRC\text{-}action$ to transmit p's request to $root(T)$. Observe that $|\mathcal{P}(root(T), p)| \leq n$ and $QR\text{-}action$ is disabled at every processor $q \in \mathcal{P}(root(T), p)\backslash\{p\}$. Suppose that for every processor q the enabled action of highest priority is $QE\text{-}action$, then after executing $QE\text{-}action$ we have $q.Q = A$ and $q.HQ = Height(q)$ and $QE\text{-}action$ is disabled at q according to Lemma 1. Then, $QRC\text{-}action$ is the enabled action of highest priority at q. As $|\mathcal{P}(root(T), p)| \leq n$, in at most $2n$ steps p's request is transmitted to $root(T)$. □

Lemma 14. *Let any allowed tree T in a static forest $\mathcal{F}$ and any processor $p \in T$ with a local request of highest priority in T transmitted to $root(T)$. If there is no new request with higher or equal priority than p's request in T, then p receives an acknowledgement from $root(T)$ in at most $2n$ steps, with n the number of processors in the network.*

Proof. We assume there is no new request with higher or equal priority than p's request in T. Thus according to Lemma 3, we have $q.Q = R$ and $q.HQ = Height(p)$ for every processor $q \in \mathcal{P}(root(T), p)$. Moreover, the following actions are disabled for every processor $q \in \mathcal{P}(root(T), p)$: $QE\text{-}action$ because there exists a child s of q such that $s.HQ = q.HQ \wedge s.Q \neq A$ (in case of $p, p.Req = WAIT$); $QR\text{-}action$ because $q.Req \neq ASK$ or $Height(q) > q.HQ = (Ch_q).HQ$; and $QRC\text{-}action$ because $q.Q = R \wedge (\forall s \in Child(q), s.Q = W \wedge s.HQ = q.HQ)$. According to Lemmas 4 and 6, since there is no new request with higher or equal priority than p's request in T thus every processor $q \in \mathcal{P}(root(T), p)$ executes $QW\text{-}action$ to wait for an acknowledgement to p's request and then executes $QA\text{-}action$ to transmit the acknowledgement from $root(T)$ to p. Observe that $|\mathcal{P}(root(T), p)| \leq n$, thus in at most $2n$ steps p receives the acknowledgement from $root(T)$ to its local request. □

Lemma 15. *Let any allowed tree T in a static forest $\mathcal{F}$. In at most $O(n^2)$ steps, at least one processor p with a local request receives an acknowledgement from $root(T)$ to its request.*

Proof. Assume without loss of generality that forest $\mathcal{F}$ is composed of a single tree T containing the n processors of the network. By Lemma 3, a request of highest priority stops the transmission of the acknowledgement of a request of lowest priority at a processor $q \in T$ because $(Ch_q).HQ < q.HQ \Rightarrow RequestT(q)$. Moreover, by Lemma 7 it is also the case at a processor $q \in T$ if there is a new request with the same priority than the previous request of highest priority because $Retransmit(q) \Rightarrow RequestT(q)$. According to Lemma 13, if there is no new request with higher or equal priority than p's request in T then in at most $2n$ steps the processor p receives an acknowledgement. However, since there is at most n requests in parallel in T then the acknowledgement of p's request can be stopped at most $n - 1$ times. □

Corollary 4. *Let a static forest of trees $\mathcal{F}$ and a given set of requests. If there is no new request in $\mathcal{F}$ then in at most $O(n^3)$ steps every processor with a local request has received an acknowledgement to its request.*

Proof. First observe that given a static forest $\mathcal{F}$, we can have a local request from at most each processor in $\mathcal{F}$, i.e., at most n processors have a local request to send. According to Lemma 15, in at most $O(n^2)$ steps at least one processor sending a request receives an acknowledgement and as we have at most n processors with a local request in $\mathcal{F}$, then the corollary follows. □

9.2. Proof of Spanning Tree algorithm

To prove the correctness of the BFS algorithm (Algorithm 1), we first introduce some useful definitions, and we show that in an illegitimate configuration there is always at least one enabled processor (Theorem 3). Then, we establish that in at most $\Theta(d^2)$ rounds the BFS algorithm constructs a BFS tree (Lemmas 19 and 20). We show that the BFS algorithm is a silent algorithm (Lemma 21), which allows us to conclude that the BFS algorithm satisfies Specification 1 under a weakly fair daemon (Lemma 22). To establish the step complexity of the BFS algorithm under an unfair daemon, we first give several additional definitions, particularly we define a *topological change* in the forest of trees which mainly represents the parent changes. First of all, we

show that any processor can be connected at most once to the normal tree via the same neighbor (Lemma 24). Based on this result, we then prove that at most $2\Delta m + mn$ requests are initiated by the BFS algorithm to construct a BFS tree (Lemma 29) starting from any configuration, since in any execution there are at most $2\Delta n + n^2$ topology changes (Corollary 9) and each topology change produce at most Δ requests (Lemma 28). Finally, we show that starting from any configuration the BFS algorithm constructs a BFS tree in at most $O(\Delta mn^3 + mn^4)$ steps (Lemma 30), since each request is handled using at most $O(n^3)$ steps by the Question-Answer algorithm.

9.2.1. Definitions

We give below the definitions used in this section, in particular we define precisely the notion of *tree* and *normal tree*.

Definition 7 (Tree). *$\forall p \in V$ such that $Allowed(p) \vee (p.P).L \geq p.L$, we define a set $Tree(p)$ of processors as follows: $\forall q \in V, q \in Tree(p)$ if and only if there exists a unique path $\mathcal{P}(p, q)$.*

Definition 8 (Normal tree). *A tree T rooted at processor $root(T)$ is called a* normal tree *if it contains only processors p such that either (i) $(p = root(T) \wedge Allowed(p))$, or (ii) $(p.S = C \wedge p.L = (p.P).L+1)$. Any tree T' rooted at processor $root(T')$ such that $\neg Allowed(root(T'))$ is called an* abnormal tree.

In the following, we consider there is only one processor $p \in V$ which is allowed to send an acknowledgement to a request, the root r, i.e., $Allowed(p) \equiv (p = r)$. Therefore, there is only one normal tree, the tree $Tree(r)$ rooted at r. Moreover, given two processors $u, v \in V$ we define by $d_H(u, v)$ the distance (in hops) between u and v in the subgraph H.

Remark 1. The system always contains one normal tree: the tree rooted at processor r.

Remark 2. All actions of Question-Answer algorithm are disabled for every processor $p \in V\backslash\{r\}$ such that $p.S = E$ or $p.L \neq (p.P).L + 1$ or $(\exists q \in Neig_p :: p.L > q.L + 1)$.

The above remark comes from the conditional composition of Algorithm $\mathcal{BFS}$. In the two first cases, a processor p cannot execute Question-Answer algorithm because Predicate $GoodT(p)$ is not satisfied, whereas the third case does not satisfy Predicate $GoodL(p)$.

Definition 9 (Locally healthy processor). *Let a tree $T \in \mathcal{F}$. A processor $p \in T$ is called* locally healthy *if p satisfies the following predicate: $p.S = C \wedge p.L = (p.P).L + 1 \wedge \neg GP\text{-}REP(p)$.*

9.2.2. Proof assuming a weakly fair daemon

Theorem 3. *Let the set of configurations $\mathcal{B} \subseteq \mathcal{C}$ such that every configuration $\gamma \in \mathcal{B}$ satisfies Definition 3. $\forall \gamma \in (\mathcal{C} - \mathcal{B}), \exists p \in V$ such that p is enabled in γ.*

Proof. Assume, by the contradiction, that $\exists \gamma \in (\mathcal{C} - \mathcal{B})$ such that $\forall p \in V$ no action is enabled at p in γ. Since $\gamma \notin \mathcal{B}$, there is at least one abnormal tree T in γ. Consider first every node $p \in T$ such that $p.S = C$. According to formal description of Algorithm 1, every processor $p \in V, p \neq r$, has a parent (i.e., $p.P \in Neig_p$). So, if $p = root(T)$ then we have $(p.P).L \geq p.L$ (see Definition 7), and $E\text{-}action$ is enabled at p, a contradiction. If $\exists p \in T$ such that $(p.P).S = E$, then $E\text{-}action$ is enabled at p, a contradiction. Now, in any abnormal tree T

we have $\forall p \in T, p.S = E$. Since $\gamma \notin \mathcal{B}$, then there is at least one abnormal tree T or $\exists q \in Neig_p, q.L - p.L > 1$ for a processor $p \in V$. So, $\exists p \in Tree(r)$, such that $GP\text{-}REP(p)$. In this case, either $p.Req = OUT$ then $A\text{-}action$ is enabled at p, a contradiction. Hence, by the contradiction, $\forall p \in Tree(r), p.Req \neq OUT$. If $p.Req = ASK$ then according to Lemma 18 in a finite time $p.Req = REP$. Thus, we assume that $p.Req = REP$. Either, $GP\text{-}REP(p)$ then there exists a processor $q \in Neig_p$ such that $C\text{-}action$ is enabled at q since $((GP\text{-}REP(p) \wedge p.Req = REP) \Rightarrow (\exists q \in Neig_p, Connect(q)))$, a contradiction. Or, $\neg GP\text{-}REP(p)$ then $O\text{-}action$ is enabled at p, a contradiction. □

Lemma 16. *Let an abnormal tree T of height h. From any configuration, in at most $h+1$ rounds we have $\forall p \in T, p.S = E$.*

Proof. We show by induction the following proposition: In at most $j + 1$ rounds, we have $\forall p \in T, (d_T(root(T), p) \leq j \Rightarrow p.S = E)$.
In base case $j = 0$. Consider any processor p such that $(p.P).L \geq p.L$. If $p.S \neq E$ then $E\text{-}action$ is enabled at p in round 0. Therefore, since the daemon is weakly fair then in the first configuration of round 1, we have $p.S = E$ at p which verifies the proposition.
Induction case: We assume that in round $j = h - 1$ we have $\forall q \in T, (d_T(root(T), q) \leq j \Rightarrow q.S = E)$. We have to show that in round $j + 1$ we have $\forall p \in T, (d_T(root(T), p) \leq j + 1 \Rightarrow p.S = E)$. Consider any node $p \in T$ of height $j + 1$ in T. By induction hypothesis, we have $(p.P).S = E$ and if $p.S \neq E$ then $E\text{-}action$ is enabled at p in round j. Thus, since the daemon is weakly fair then in the first configuration of round $j + 1$ we have $p.S = E$ and we have also $\forall q \in T, (d_T(root(T), q) \leq j \Rightarrow q.S = E)$. Therefore, in at most $h + 1$ rounds we have $\forall p \in T, (d_T(root(T), p) \leq h \Rightarrow p.S = E)$. □

Lemma 17. *Let a normal tree T in a static forest $\mathcal{F}$ and a processor $p \in T$ at height k with a local request. From any configuration, in at most $O(k^2)$ rounds the request of p is transmitted to $root(T)$.*

Proof. We show by induction the following proposition: For any node $p \in T$ at height $j \geq 0$ in T such that $p.Req = WAIT$, in at most $O(j^2)$ rounds we have $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.HQ = Height(p)$.
In the base case $j = 0$ and we consider $p = root(T)$. According to Lemma 8, in at most $j + 1 = 1$ round we have $p.Q = R \wedge p.HQ = Height(p)$, which verifies the proposition.
Induction case: We assume that for $j = k - 1$ after $O(j^2)$ rounds for each node $p \in T$ at height $k - 1$ in T such that $p.Req = WAIT$ we have $\forall q \in \mathcal{P}(root(T), p), q.Q = R$ and $q.HQ = Height(p)$. Consider any node $p \in T$ of height $j + 1$ in T. We have to show that in at most $O((j + 1)^2)$ rounds we have $\forall q \in \mathcal{P}(root(T), p), (Height(q) \leq j + 1 \Rightarrow (q.Q = R \wedge q.HQ = Height(p)))$. According to Lemmas 9 and 10, in at most $O(j)$ additional rounds we have $x.Req \neq REP$ and $x.Q = A$ at each node x of height j in T (in particular at node $x = p.P$). According to Lemma 8, in at most $j + 1$ additional rounds we have $\forall q \in \mathcal{P}(root(T), p), (Height(q) \leq j + 1 \Rightarrow (q.Q = R \wedge q.HQ = Height(p)))$. Thus, in at most $j^2 + (j - 1) + (j + 1) < O(j^2)$ rounds we have $\forall q \in \mathcal{P}(root(T), p), (Height(q) \leq j + 1 \Rightarrow (q.Q = R \wedge q.HQ = Height(p)))$, and the proposition is verified at p on height $j + 1$ in T. □

Lemma 18. *Let a normal tree T in a static forest $\mathcal{F}$ and a processor $p \in T$ at height k with a local request. From any configuration, in at most $O(k^2)$ rounds p receives an acknowledgement to its local request.*

Proof. Let a processor $p \in T$ such that $p.Req = ASK$ of height k in T. According to Lemma 17, from any configuration in at most $O(k^2)$ rounds we have $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.HQ = Height(p)$. Thus, we can apply Lemma 10 and in at most $k+1$ additional rounds we have $p.Req = REP \wedge p.Q = A \wedge p.HQ = Height(p)$ at p. □

Lemma 19. *From any configuration, in at most $O(d^2)$ rounds Algorithm $\mathcal{BFS}$ reaches a configuration $\gamma \in \mathcal{C}$ satisfying Definition 3, with d the diameter of the network.*

Proof. Note that by definition of Predicate $Allowed(p) \equiv (p = r)$ and according to Property (Safety 2) of Specification 2, only the nodes sending a request in the tree rooted at r can receive an acknowledgement to a request. Moreover, we have the following constant values at r: $r.S = C, r.P = \perp$, and $r.L = 0$.

We first show by induction on the distances of the network the following proposition: in at most $O(j^2)$ rounds, $\forall p \in V, (d_G(r,p) \leq j \Rightarrow (p \in Tree(r) \wedge (\forall q \in Neig_p, q \in Tree(r) \wedge q.L - p.L \leq 1)))$.

In base case $j = 0$. We have first that $r \in Tree(r)$. To verify the proposition at r, we must consider any neighbor q of r in the network such that $r = MinChPar(q)$.

- First consider that $q \notin Tree(r), q \in T$ with T an abnormal tree of the forest. Either case (A) $d_G(root(T), q) \leq j$ then according to Lemma 16 in at most $O(1)$ rounds q has detected it is in an abnormal tree, thus we have $(q.S = E \Rightarrow GP\text{-}REP(r))$. In this case, $E\text{-}action$ is not enabled at q and according to Property (Liveness 2) of Specification 2 and to Lemma 18 in at most $O(j^2) = O(1^2)$ rounds we have $r.Req = REP$ at r. Thus, q can execute $C\text{-}action$, so in $O(1)$ additional rounds we have $q.S = C, q.P = r$, and $q.L = 1$ at q. Or case (B) $d_G(root(T), q) > j$, we must consider two sub-cases: (B1) $d_G(root(T), q) = j + 1 = 2$ or (B2) $d_G(root(T), q) > 2$.
  - In the sub-case (B1), $d_G(root(T), q) = 2$. According to Lemma 16 in at most $j + 1 = O(1)$ rounds $q.P$ has detected it is in an abnormal tree T and we have $(q.P).S = E$, thus q can execute $E\text{-}action$ and in $O(1)$ additional rounds we have $q.S = E$ leading to the case (A).
  - In the sub-case (B2), $d_G(root(T), q) > 2$. $E\text{-}action$ is not enabled at q and $d_G(root(T), q) > 2 \Rightarrow q.L - r.L > 1$. Thus, according to Property (Liveness 2) of Specification 2 and to Lemma 18 in at most $O(j^2) = O(1^2)$ rounds we have $r.Req = REP$ at r. Then, $C\text{-}action$ is the enabled action with the highest priority at q and in $O(1)$ additional rounds it is executed by q to obtain $q.S = C, q.P = r$, and $q.L = 1$.
- Otherwise, consider that $q \in Tree(r)$ then we have $q.S = C$ and $E\text{-}action$ is not enabled at q. We must consider the case such that $q.L - r.L > 1$ at q. We have $(q.L - r.L > 1 \Rightarrow GP\text{-}REP(r))$ and according to Property (Liveness 2) of Specification 2 and to Lemma 18 in at most $O(j^2) = O(1^2)$ rounds we have $r.Req = REP$ at r. So q can execute $C\text{-}action$ and in $O(1)$ additional rounds we have $q.S = C, q.P = r$, and $q.L = 1$ at q.

Therefore, since the daemon is weakly fair in at most $O(1)$ rounds for every neighbor q of r the parent of q is r (i.e., $q \in Tree(r)$) and $q.L - r.L \leq 1$, which verifies the proposition.

Induction case: We assume the proposition is verified for every node at distance $j-1$ from r in the network. We have to show the proposition is also verified for every node at distance j from

r. Consider any node p at distance j from r. By induction hypothesis, we have $p \in Tree(r)$. Let any node $q \in Neig_p$ such that $p = MinChPar(q)$.

- First consider that $q \notin Tree(r), q \in T$. Either case (A) $d_G(root(T), q) \leq j$, then according to Lemma 16 in at most $j+1$ rounds q has detected it is in an abnormal tree T and we have $(q.S = E \Rightarrow GP\text{-}REP(p))$. In this case, $E\text{-}action$ is not enabled at q and according to Property (Liveness 2) of Specification 2 and to Lemma 18 in at most $O(j^2)$ rounds we have $p.Req = REP$ at p. Thus, q can execute $C\text{-}action$, so in $O(1)$ additional rounds we have $q.S = C, q.P = p$, and $q.L = p.L+1$ at q. Or case (B) $d_G(root(T), q) > j$, we must consider two sub-cases: (B1) $d_G(root(T), q) = j + 1$ or (B2) $d_G(root(T), q) > j + 1$.
  - In the sub-case (B1), $d_G(root(T), q) = j + 1$. According to Lemma 16 in at most $j + 1$ rounds $q.P$ has detected it is in an abnormal tree T and we have $(q.P).S = E$, thus q can execute $E\text{-}action$ and in $O(1)$ additional rounds we have $q.S = E$ leading to the case (A).
  - In the sub-case (B2), $d_G(root(T), q) > j + 1$. $E\text{-}action$ is not enabled at q and $d_G(root(T), q) > j+1 \Rightarrow q.L - p.L > 1$. Thus, according to Property (Liveness 2) of Specification 2 and to Lemma 18 in at most $O(j^2)$ rounds we have $p.Req = REP$ at p. Then, $C\text{-}action$ is the enabled action with the highest priority at q and in $O(1)$ additional rounds it is executed by q to obtain $q.S = C, q.P = p$, and $q.L = p.L+1$.
- Otherwise, consider that $q \in Tree(r)$ then we have $q.S = C$ and $E\text{-}action$ is not enabled at q. We must consider the case such that $q.L - p.L > 1$ at q. We have $(q.L - p.L > 1 \Rightarrow GP\text{-}REP(p))$ and according to Property (Liveness 2) of Specification 2 and to Lemma 18 in at most $O(j^2)$ rounds we have $p.Req = REP$ at p. So q can execute $C\text{-}action$ and in $O(1)$ additional rounds we have $q.S = C, q.P = p$, and $q.L = p.L + 1$ at q.

Therefore, since the daemon is weakly fair in at most $O(j^2)$ rounds for every neighbor q of p we have $q \in Tree(r)$ and $q.L - p.L \leq 1$, which verifies the proposition. Note that, at distance j from r when the proposition is verified for any processor $p \in Tree(r)$ then p can execute $O\text{-}action$. So, since the daemon is weakly fair in at most $O(j^2)$ rounds we have $\forall p \in V, (d_G(r, p) \leq j \Rightarrow (p \in Tree(r) \wedge p.Req = OUT)$.

We now show that the configuration γ reached by Algorithm $\mathcal{BFS}$ in $O(d^2)$ rounds verifies Definition 3. Let d the diameter of the network G. In the proof above, any processor $p \in V$ at distance d from r belongs to the subgraph $Tree(r)$ in at most $O(d^2)$ rounds, otherwise G is not a connected network. Moreover, there is a path between any processor $p \in V$ and r in $Tree(r)$, so the subgraph $Tree(r)$ is connected. Observe that, the subgraph $Tree(r)$ is a spanning tree of the network G. Indeed, every processor $p \in V$ has a parent in $Tree(r)$ except r which has no parent (i.e., there is an unique path between p and r) and $Tree(r)$ is connected, so the subgraph $Tree(r)$ contains no cycle. Thus, these remarks imply that the configuration γ verifies Claims 1 and 2 of Definition 3. To show the last Claim of Definition 3, assume by the contradiction that $Tree(r)$ is not a breadth first search tree. This implies that $\exists p \in Tree(r)$ such that $(\exists q \in Neig_p :: q.L < (p.P).L)$. That is, we have $p.L - q.L > 1$ which contradicts the proposition verified by every processor $p \in Tree(r)$ according to the induction proof above. Therefore, Claim 3 of Definition 3 is verified, which finishes to show the lemma. □

Corollary 5. *From any configuration, in at most $O(d^2)$ rounds there is no abnormal tree in forest $\mathcal{F}$, with d the diameter of the network.*

Now, in the following lemma we will establish a lower bound on the round complexity of Algorithm $\mathcal{BFS}$.

Lemma 20. *Algorithm $\mathcal{BFS}$ reaches a configuration $\gamma \in \mathcal{C}$ satisfying Definition 3 in $\Omega(d^2)$ rounds, with d the diameter of the network.*

Proof. To show the lower bound we will use the network illustrated in Figure 5(a). This is a star network where the root r is at one extremity of a branch of the star. This topology can be generalized by extending the length k of the branches (here $k = 2$) or by adding more branches to the star, however the lower bound is preserved.

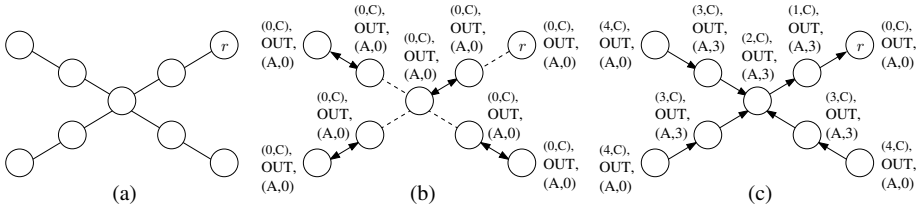

Figure 5: An example used to show the lower bound on round complexity of Algorithm $\mathcal{BFS}$. (a) the network topology, (b) the initial configuration, and (c) the legitimate configuration.

We consider the initial configuration given in Figure 5(b) with $\lceil \frac{n}{2} \rceil$ trees in the forest $\mathcal{F}$. Moreover, for each processor p, the values of the variables are separated in three parts: the first pair $(p.L, p.S)$ related to Algorithm 1, followed by the shared variable $p.Req$, and a last pair $(p.Q, p.HQ)$ related to Algorithm 2. In the initial configuration, each processor has the same state: $(0, C), OUT, (A, 0)$.

We will now determine the number of rounds necessary to Algorithm $\mathcal{BFS}$ to reach a legitimate configuration:

- In the first round, every processor $p, p \neq r$, executes E-action of Algorithm 1 to set $p.S$ to E, since we have $(p.P).L \geq p.L \Rightarrow AbnormalTree(p)$.
- By recurrence following the increasing order on network distances $i, 0 \leq i \leq d-1$:
  1. Processors at distance i send a local request to Algorithm 2 and receive an acknowledgement from Algorithm 2 after $3i + 3$ rounds according to Lemmas 8, 9 and 10, since the request of processors i has the highest priority in the network.
  2. Processors at distance $i + 1$ can hook to their neighbor at distance i since an acknowledgement from the root r has been given. To this end, in round $3i + 4$ these processors p execute C-action of Algorithm 1 to set their variables $p.S, p.P$ and $p.L$ accordingly.
  3. Finally, in round $3i + 5$ processors at distance i execute O-action of Algorithm 1 to set their variable Req to OUT since all their neighbors at distance $i + 1$ are connected (i.e., they have no neighbor satisfying $GP\text{-}REP()$). This allows to reach the legitimate configuration in Figure 5(c).

By summing up all the rounds, Algorithm $\mathcal{BFS}$ needs $3i + 5$ rounds to add each layer $i + 1$ to the BFS tree. Since there are d layers in a BFS tree, we have the following equation which shows the lemma: $\sum_{i=0}^{d-1} 3i + 5 = 3 \times \frac{d(d+1)}{2} + 5 = \Omega(d^2)$ rounds. □

From Lemmas 19 and 20, we have the following result.

Corollary 6. *From any configuration, in $\Theta(d^2)$ rounds Algorithm $\mathcal{BFS}$ reaches a configuration $\gamma \in \mathcal{C}$ satisfying Definition 3, with d the diameter of the network.*

Lemma 21. *In every configuration $\gamma \in \mathcal{C}$ satisfying Definition 3, for every processor $p \in V$ no action of Algorithm 1 is enabled in γ.*

Proof. Observe first that since γ satisfies Definition 3, then for every processor $p \in V$ we have $\forall q \in Neig_p, |p.L - q.L| \leq 1$. Moreover, there is a single tree spanning every processor $p \in V$, thus there exists no abnormal tree and by Definition 7 for every processor $p \in V$ we have $p.S = C \wedge p.L = (p.P).L + 1$. These two observations imply that every processor $p \in V$ is locally healthy in γ (see Definition 9).

Assume, by the contradiction, that $\exists \gamma \in \mathcal{C}$ satisfying Definition 3 such that $\exists p \in V$ with an enabled action of Algorithm 1 at p. If $E\text{-}action$ is enabled at p then $(p.P).S = E$ or $(p.P).L \geq p.L$, a contradiction because p is a locally healthy processor in γ. If $C\text{-}action$ is enabled at p and $p.S = C$ then $\exists q \in Neig_p$ such that $p.L - q.L > 1$, a contradiction because p is locally healthy. If $A\text{-}action$ is enabled at p then $\exists q \in Neig_p$ such that either $q.S = E$, a contradiction because we have $\forall p \in V, p.S = C$ in γ, otherwise $\exists q \in Neig_p, q.L - p.L > 1$, a contradiction because p is locally healthy. Finally, if $O\text{-}action$ is enabled at p then $p.Req = REP$ and p can execute $O\text{-}action$ in step $\gamma \mapsto \gamma'$. In configuration γ', we have $p.S = C \wedge p.Req = OUT$ so $O\text{-}action$ is disabled. Moreover, there is no request because every processor $p \in V$ is locally healthy in γ', a contradiction. □

By Lemmas 11 and 21, we have the following corollary.

Corollary 7. *In every configuration $\gamma \in \mathcal{C}$ satisfying Definition 3, every action of Algorithm $\mathcal{BFS}$ is disabled at each processor $p \in V$ in γ.*

Lemma 22. *From any configuration, the execution satisfies Specification 1.*

Proof. We have to show that starting from any configuration the execution of Algorithm $\mathcal{BFS}$ verifies Property [TC1] and [TC2] of Specification 1.

According to Lemma 19 and Corollary 7, from any configuration Algorithm $\mathcal{BFS}$ reaches a configuration $\gamma \in \mathcal{C}$ in finite time and γ is a terminal configuration, which verifies Property [TC1] of Specification 1. Moreover, according to Lemma 19 the terminal configuration γ reached by Algorithm $\mathcal{BFS}$ satisfies Definition 3, which verifies Property [TC2] of Specification 1. □

Theorem 3 and Lemma 22 imply the following theorem.

Theorem 4. *Algorithm $\mathcal{BFS}$ is snap-stabilizing for Specification 1 under a weakly fair daemon.*

9.2.3. Proof assuming an unfair daemon

Definition 10 (Topological change). *Given a forest $\mathcal{F}$ of trees in a configuration $\gamma \in \mathcal{C}$. A* topological change *in $\mathcal{F}$ is obtained by the execution of one of the following actions at a processor $p \in V$ in step $\gamma \mapsto \gamma'$: p executes $E\text{-}action$, or p executes $C\text{-}action$.*

Remark 3. For every processor $p \in Tree(r)$, $E\text{-}action$ is disabled at p.

Remark 4. $E\text{-}action$, $C\text{-}action$, and $A\text{-}action$ are disabled at every locally healthy processor $p \in V$.

Proposition 1. *Every $p \in V$ is hooked on to the neighbor q such that $\forall s \in Neig_p, q.L \leq s.L$.*

Proof. According to formal description of Algorithm 1, a processor hooks on to a neighbor using $C\text{-}action$. Assume, by the contradiction, that there is a processor $p \in V$ such that $\exists s \in Neig_p, (p.P).L > s.L$. We must consider two cases: s is in an abnormal tree or not. If s is in an abnormal tree then either $s.S = E$ then $s \notin MinChPar(p) \Rightarrow \neg Connect(p)$ a contradiction, or $s.S = C$ then by Property (Safety 2) of Specification 2 s never receives an acknowledgement and we have that $s.Req \neq REP \Rightarrow \neg Connect(p)$, otherwise $C\text{-}action$ is enabled at p, a contradiction. If s is in a normal tree then by Property (Liveness 2) of Specification 2 we have that $s.Req = REP$ and $C\text{-}action$ is enabled at p, a contradiction. □

Lemma 23. *Let any abnormal tree $T \in \mathcal{F}$ and the set of processors $B = \{p \in V : p \notin T \wedge (\exists q \in Neig_p :: q \in T)\}$. In an execution, only processors in B can hook on to T.*

Proof. Consider any abnormal tree $T \in \mathcal{F}$ in configuration $\gamma \in \mathcal{C}$. According to formal description of Algorithm 1, a processor p must execute $C\text{-}action$ to hook on to a tree, i.e., there is a neighbor q such that $q.Req = REP$. Suppose that every processor $q \in B$ executes $C\text{-}action$ and they are hooked on to T in configuration γ_k. Note that after executing $C\text{-}action$, we have $q.Req = OUT$ at every processor $q \in B$. Assume, by the contradiction, that there is a processor $p \notin T$ in configuration γ_k which hooks on to T in step $\gamma_k \mapsto \gamma_{k+j}, j > 0$. This implies that p hooks on to a neighbor $q \in B$ (by definition of B) such that $q.Req = REP$, a contradiction by Property (Safety 2) of Specification 2 because q can not receive an acknowledgement from $root(T)$ since T is an abnormal tree. □

Corollary 8. *Let any abnormal tree $T \in \mathcal{F}$ and the set of processors $B = \{p \in V : p \notin T \wedge (\exists q \in Neig_p :: q \in T)\}$. In an execution, at most $|B|$ processors can hook on to T.*

Proposition 2. *Let a processor $p \in V$ which hooks on to a tree T in configuration $\gamma_i \in \mathcal{C}$. If another processor $q \in V$ hooks on to T by p in $\gamma_{i+j}, j > 0$, then T is a normal tree.*

Proof. According to Lemma 23, the expansion of an abnormal tree T' is limited at distance one from T'. After p hooks on to T, to allow the processor q to hook on to T by p then p receives an acknowledgement from $root(T)$. Therefore, T is a normal tree by Specification 2. □

Lemma 24. *Let any abnormal tree $T \in \mathcal{F}$. A processor $p \in V$ can hook on to T at most once by the same neighbor $q \in T$.*

Proof. Assume, by the contradiction, that there is a configuration $\gamma_k \in \mathcal{C}$ such that there is a processor $p \in V$ which hooks on to T by the same neighbor $q \in T$ a second time. To hook on to T, p must execute $C\text{-}action$, i.e., there is a neighbor $x \in T$ of p such that $x.S = C$ and $x.Req = REP$. According to Proposition 1, p hooks on to the neighbor $x \in V$ such that $x.S = C \wedge (\forall s \in Neig_p, x.L \leq s.L)$. Suppose that p hooks on to T by the neighbor q a first time in step $\gamma_{i-1} \mapsto \gamma_i \in \mathcal{C}$, then p hooks on to another neighbor s of p, $s \neq q$, in step $\gamma_{j-1} \mapsto \gamma_j \in \mathcal{C}, j > i$. Now, we must consider several cases in configuration $\gamma_k, i < j < k$. If p is hooked on to s in γ_j because $q.S = E$ and $s.Req = REP$ in γ_i then since $q \in T$ we have $q.S = E$ in γ_k and $q \notin MinChPar(p) \Rightarrow \neg Connect(p)$, a contradiction. Otherwise

$s.S = q.S = C$ and p is hooked on to s in $\gamma_j, i < j < k$, because $s.L < q.L$ and $s.Req = REP$. When p hooks on to q the first time in step $\gamma_{i-1} \mapsto \gamma_i$, we have $s.S = E$ or $s.L > q.L$. Since we have $s.S = C \wedge s.L < q.L \wedge s.Req = REP$ and p hooks on to s in step $\gamma_{j-1} \mapsto \gamma_j$, this implies that s is in a normal tree in γ_j according to Proposition 2. Thus, we have $s.S = C \wedge s.L < q.L$ in γ_k and $q \notin MinChPar(p) \Rightarrow \neg Connect(p)$, a contradiction. □

Lemma 25. *In an execution, every processor $p \in V \backslash \{r\}$ produces at most 2Δ topological changes in forest $\mathcal{F}$ while $p \notin Tree(r)$, with Δ the maximum degree of a processor in the network.*

Proof. To hook on to a tree, a processor $p \in V$ must execute $C\text{-}action$. According to Lemma 24, p cannot hooks on to an abnormal tree $T \in \mathcal{F}$ twice by the same neighbor q of p. Since a processor can have at most Δ neighbors, p can hook on at most Δ times to an abnormal tree. Observe that $E\text{-}action$ has a higher priority than $C\text{-}action$ and $E\text{-}action$ can be executed between two executions of $C\text{-}action$, i.e., at most Δ times while $p \notin Tree(r)$. Therefore, by Definition 10 the lemma follows. □

Lemma 26. *In an execution, every processor $p \in V \backslash \{r\}$ produces at most n topological changes in forest $\mathcal{F}$ while $p \in Tree(r)$, with n the number of processors in the network.*

Proof. Observe that for every processor $p \in Tree(r)$ we have $p.S = C$. Moreover, by Remark 3 for every processor $p \in Tree(r)$ we have that $E\text{-}action$ is disabled. So, by Definition 10 the only topological change in $\mathcal{F}$ that a processor $p \in Tree(r)$ can produce is to execute $C\text{-}action$ in order to reduce its level in $Tree(r)$. Thus, by Proposition 1 each execution of $C\text{-}action$ by a processor $p \in Tree(r)$ in step $\gamma_i \mapsto \gamma_{i+1}$ implies that p hooks on to the neighbor with the lowest level in γ_{i+1} and $p.L$ in γ_i is higher than $p.L$ in γ_{i+1}. Therefore, since the size of $Tree(r)$ is bounded by n then any processor p can hook on to at most $n - 1$ processors by executing $C\text{-}action$ while $p \in Tree(r)$. □

Lemma 27. *In an execution, every processor $p \in V \backslash \{r\}$ produces at most $2\Delta + n$ topological changes in forest $\mathcal{F}$.*

Proof. This comes from Lemmas 25 and 26. □

Corollary 9. *From any configuration, Algorithm 1 produces at most $2\Delta n + n^2$ topological changes in forest $\mathcal{F}$.*

Lemma 28. *In an execution, each topological change in forest $\mathcal{F}$ generates at most Δ requests.*

Proof. Let any processor $p \in V$ which produces a topological change in forest $\mathcal{F}$. By Definition 10, we must consider two cases: $p.S = E$ (in this case $p \neq r$) or $p.S = C \wedge (\exists q \in Neig_p, q.L - p.L > 1)$. If $p.S = E$ then we can have $p.S = E \Rightarrow GP\text{-}REP(q)$ at a neighbor q of p, so since a processor can have at most Δ neighbors this can generate at most Δ requests. Otherwise we have $p.S = C \wedge (\exists q \in Neig_p, q.L - p.L > 1)$ at p, then p sends a request in order to allow each neighbor q such that $q.L - p.L > 1$ to hook on to p. Therefore, at most Δ requests are generated by a topological change at p. □

Lemma 29. *From any configuration, Algorithm 1 produces at most $2\Delta m + mn$ requests to reach a configuration satisfying Definition 3.*

Proof. This comes from Corollary 9 and Lemma 28. □

Corollary 10. *In an execution, A-action and O-action are executed at most $2\Delta m + mn$ times in the network.*

Lemma 30. *From any configuration, at most $O(\Delta mn^3 + mn^4)$ steps are needed by Algorithm $\mathcal{BFS}$ to reach a configuration satisfying Definition 3.*

Proof. By Corollary 9, from any configuration Algorithm 1 generates at most $2\Delta n+n^2$ topological changes to reach a configuration satisfying Definition 3. Thus, by Definition 10 this implies that E-action and C-action are executed at most $\Delta n + n^2$ times. Moreover, by Corollary 10 from any configuration A-action and O-action are executed at most $2\Delta m + mn$ times to send a local request. According to Corollary 4, an acknowledgement to a request is received in at most $O(n^3)$ steps. Therefore, from any configuration in at most $O(\Delta mn^3 + mn^4)$ steps a legitimate configuration is reached. □

10. Conclusion

In this paper we define a particular class of distributed algorithms, called *fully polynomial* algorithms, having good properties suitable for large scale systems. These algorithms are characterized by a round complexity polynomial on the network diameter and a step complexity polynomial on the network size. Moreover, we show that this class of distributed algorithms is not empty for the global distributed task of spanning tree construction. To this end, we proposed the first fully polynomial stabilizing algorithm constructing a Breadth First Search tree in $\Theta(d^2)$ rounds and in $O(n^6)$ steps for any topology network, with d the diameter and n the number of nodes in the network. Moreover, a distributed daemon without any fairness assumptions is considered.

Several open questions follow from this work:

- There is a need to classify other global distributed tasks such as leader election, Propagation of Information with Feedback, routing table construction ..., to determine if there exist also fully polynomial algorithms for these problems.
- We show an upper bound on the step complexity of our algorithm. However, is it possible to improve the step complexity of the algorithm while preserving a round complexity of $O(d^2)$?
- In the same way, is it possible to design a fully polynomial stabilizing algorithm for the spanning tree construction problem with an optimal round complexity of $\Theta(d)$?

The positive result proposed in this paper for the spanning tree construction problem allows to consider the design of stabilizing algorithms for large scale systems. Therefore, it is crucial to continue the investigation of *totally effective* fault-tolerant distributed algorithms (with optimal round and step complexities) for global tasks.

References

[1] B. Bollobás, O. Riordan, The diameter of a scale-free random graph, Combinatorica 24 (1) (2004) 5–34.

[2] A. Cournier, S. Rovedakis, V. Villain, The first fully polynomial stabilizing algorithm for bfs tree construction, in: 15th International Conference on Principles of Distributed Systems, Vol. 7109 of Lecture Notes in Computer Science, Springer, 2011, pp. 159–174.

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

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

[5] G. Tel, Introduction to distributed algorithm, Cambridge University Press, 2e edition, 2000.

[6] A. Bui, A. Datta, F. Petit, V. Villain, State-optimal snap-stabilizing pif in tree networks, in: Workshop on Self-stabilizing Systems, IEEE Computer Society, 1999, pp. 78–85.

[7] A. Cournier, S. Devismes, V. Villain, Light enabling snap-stabilization of fundamental protocols, ACM Transactions on Autonomous and Adaptive Systems 4 (1).

[8] S. Devismes, T. Masuzawa, S. Tixeuil, Communication efficiency in self-stabilizing silent protocols, in: 29th IEEE International Conference on Distributed Computing Systems, 2009, pp. 474–481.

[9] T. Masuzawa, Silence is golden: Self-stabilizing protocols communication-efficient after convergence, in: 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, 2011, pp. 1–3.

[10] T. Masuzawa, T. Izumi, Y. Katayama, K. Wada, Brief announcement: Communication-efficient self-stabilizing protocols for spanning-tree construction, in: 13th International Conference on Principles of Distributed Systems, 2009, pp. 219–224.

[11] S. Kutten, D. Zinenko, Low communication self-stabilization through randomization, in: 24th International Symposium on Distributed Computing, 2010, pp. 465–479.

[12] F. Gärtner, A survey of self-stabilizing spanning-tree construction algorithms, Tech. rep., EPFL (October 2003).

[13] A. Arora, M. Gouda, Distributed reset (extended abstract), in: 10th Conference on Foundations of Software Technology and theoretical Computer Science, 1990, pp. 316–331.

[14] S. Dolev, A. Israeli, S. Moran, Self-stabilization of dynamic systems assuming only read/write atomicity, in: 9th ACM symposium on Principles of distributed computing, 1990, pp. 103–117.

[15] Y. Afek, S. Kutten, M. Yung, Memory-efficient self-stabilizing protocols for general networks, in: 4th International Workshop on Distributed Algorithm, Vol. LNCS 486, Springer, 1991, pp. 15–28.

[16] N.-S. Chen, H.-P. Yu, S.-T. Huang, A self-stabilizing algorithm for constructing spanning trees, Information Processing Letters 39 (3) (1991) 147–151.

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

[18] A. Cournier, Mémoire d'Habilitation à Diriger les Recherches : Graphes et algorithmique distribuée stabilisante, Université de Picardie Jules Verne (2009).

[19] J. Burman, S. Kutten, Time optimal asynchronous self-stabilizing spanning tree, in: 21st International Symposium on Distributed Computing, 2007, pp. 92–107.

[20] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese, Time optimal self-stabilizing synchronization, in: 25th Annual ACM Symposium on Theory of Computing, 1993, pp. 652–661.

[21] A. Datta, L. Larmore, P. Vemula, Self-stabilizing leader election in optimal space, in: 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Vol. 5340 of Lecture Notes in Computer Science, Springer, 2008, pp. 109–123.

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

[23] A. Cournier, S. Devismes, V. Villain, A snap-stabilizing dfs with a lower space requirement, in: 7th International Symposium on Self-Stabilizing Systems, Vol. 3764 of Lecture Notes in Computer Science, Springer, 2005, pp. 33–47.

[24] C. Johnen, J. Beauquier, Distributed self-stabilizing depth-first token circulation with constant memory, in: 2nd Workshop on Self-Stabilizing System, 1995, pp. 4.1–4.15.

[25] C. Johnen, Memory-efficient self-stabilizing algorithm to construct bfs spanning trees, in: 3rd Workshop on Self-stabilizing Systems, 1997, pp. 125–140.

[26] B. Ducourthial, S. Tixeuil, Self-stabilization with path algebra, Theoretical Computer Science 293 (1) (2003) 219–236.

[27] A. K. Datta, L. L. Larmore, P. Vemula, Self-stabilizing leader election in optimal space under an arbitrary scheduler, Theoretical Computer Science 412 (40) (2011) 5541–5561. doi:10.1016/j.tcs.2010.05.001.
URL https://doi.org/10.1016/j.tcs.2010.05.001

[28] A. K. Datta, L. L. Larmore, P. Vemula, An O(n)-time self-stabilizing leader election algorithm, Journal of Parallel and Distributed Computing 71 (11) (2011) 1532–1544. doi:10.1016/j.jpdc.2011.05.008.
URL https://doi.org/10.1016/j.jpdc.2011.05.008

[29] A. Kosowski, L. Kuszner, A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves, in: 6th International Conference on Parallel Processing and Applied Mathematics, Vol. 3911 of LNCS, Springer, 2005, pp. 75–82.

[30] A. Cournier, A new polynomial silent stabilizing spanning-tree construction algorithm, in: 16th International Colloquium on Structural Information and Communication Complexity, Vol. 5869 of Lecture Notes in Computer Science, Springer, 2009, pp. 141–153.

[31] A. Cournier, S. Devismes, F. Petit, V. Villain, Snap-stabilizing depth-first search on arbitrary networks, The Computer Journal 49 (3) (2006) 268–280.

[32] S. Devismes, C. Johnen, Silent self-stabilizing bfs tree algorithms revisited, Journal of Parallel and Distributed Computing 97 (2016) 11–23. doi:https://doi.org/10.1016/j.jpdc.2016.06.003.
URL http://www.sciencedirect.com/science/article/pii/S0743731516300685

[33] K. Altisen, A. Cournier, S. Devismes, A. Durand, F. Petit, Self-stabilizing leader election in polynomial steps, Information and Computation 254 (2017) 330–366. doi:10.1016/j.ic.2016.09.002.
URL https://doi.org/10.1016/j.ic.2016.09.002

[34] A. Cournier, S. Devismes, V. Villain, Snap-stabilizing pif and useless computations, in: 12th International Conference on Parallel and Distributed Systems, IEEE Computer Society, 2006, pp. 39–48.

[35] E. J. H. Chang, Echo algorithms: Depth parallel operations on general graphs, IEEE Transactions on Software Engineering 8 (4) (1982) 391–401. doi:10.1109/TSE.1982.235573.
URL https://doi.org/10.1109/TSE.1982.235573

[36] A. Segall, Distributed network protocols, IEEE Transactions on Information Theory 29 (1) (1983) 23–34. doi:10.1109/TIT.1983.1056620.
URL https://doi.org/10.1109/TIT.1983.1056620

[37] A. Datta, S. Gurumurthy, F. Petit, V. Villain, Self-stabilizing network orientation algorithms in arbitrary rooted networks, Studia Informatica Universalis 1 (1) (2001) 1–22.