An improved stabilizing BFS tree construction

Alain Cournier, Stephane Rovedakis, Vincent Villain

An improved stabilizing BFS tree construction*

Alain Cournier[1] Stephane Rovedakis[2] Vincent Villain[3]

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 recover a correct behaviour from an arbitrary global state resulting from transient faults. A *silent* algorithm always reaches a terminal global state in a finite time.

We present a first silent stabilizing algorithm to resolve a problem in which each node requests a permission (delivered by a subset of network nodes) in order to perform a defined computation. Using this first algorithm, we present a silent stabilizing algorithm constructing a BFS tree working in $O(D^2)$ rounds (D is the diameter of the network) under a distributed daemon without any fairness assumptions. The complexity in terms of steps is $O(mn^4)$ where m and n are the number of edges and nodes of the network, respectively, so it is polynomial with respect to n. To our knowledge, since in general the diameter of a network is much smaller than the number of nodes, this algorithm gets the best compromise of the literature between the complexities in terms of rounds and in terms of steps.

Keywords: Distributed algorithm, Fault-tolerance, Self-stabilization, Spanning tree construction.

1 Introduction

The construction of spanning trees is a fundamental problem in the field of distributed systems. A spanning tree is a virtual structure which contains no cycle and interconnects 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. Spanning trees are also used to obtain algorithms resolving a particular distributed problem with a better time complexity compared to algorithms for the same problem which do not use this structure. There are many different spanning tree construction problems guaranteeing various properties, e.g., the construction of a depth first search (DFS) tree, a spanning tree of minimum weight or a spanning tree of minimum diameter. A crucial class of spanning trees is the construction of a Breadth First Search (BFS) tree, which contains shortest paths (in hops) from every node to the root of the tree. This structure is mainly used in networks to quickly broadcast information from a source node. When a cost is associated to communication links, this problem is known as the construction of a Shortest Path tree.

Self-stabilization introduced first by Dijkstra in [15] and later publicized by several books [16, 23] 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)

[1]Laboratoire MIS, Université de Picardie, 33 Rue St Leu, 80039 Amiens Cedex 01, France.
alain.cournier@u-picardie.fr
[2]Laboratoire CEDRIC, CNAM, 292 Rue St Martin, 75141 Paris Cedex 03, France.
stephane.rovedakis@cnam.fr
[3]Laboratoire MIS, Université de Picardie, 33 Rue St Leu, 80039 Amiens Cedex 01, France.
vincent.villain@u-picardie.fr
*This work was founded by ANR project SPADES

the system is able to recover from this catastrophic situation in finite time without external (e.g., human) intervention. As self-stabilization makes no hypothesis about the nature or the extent of the faults, this paradigm can also be used to handle dynamic changes on the network topology since these modifications are seen as faults by the system. Another kind of stabilization was introduced by Bui *et al* [4], called *snap-stabilization*. These algorithms have the ability to always guarantee a correct system behaviour according to the specifications of the problem to be solved, starting from any arbitrary global state.

Related work. Due to the importance of the construction of spanning trees, there are a lot of works which study this task. Arora and Gouda [2] are interested in designing an algorithm which allows to reset a network by reseting 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 [17] 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 [1] have proposed independently from [17] 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, with n the number of nodes in the network. Moreover, it incorporates a mechanism to transmit requests and acknowledgements for the add 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 [6], which was improved later to construct a BFS tree [19]. The time complexities of these algorithms are $\Theta(n)$ rounds for [6] and $\Theta(D)$ rounds for [19] (with small modifications) as analyzed in [8], with D the network diameter. More recently, Burman and Kutten [5] give a solution to construct a Shortest Path tree in $O(D)$ rounds, extending to the massage passing model a solution proposed by Awerbuch *et al* [3]. Datta, Larmore, and Vemula [14] 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 is reached the values stored in the registers do not change anymore. $O(D)$ additional rounds are needed to the algorithm to become silent.

Some of the algorithms cited above are optimal in terms of rounds for the construction of an arbitrary spanning tree or a BFS tree. However, another important complexity measure for an algorithm is the number of moves needed to compute the solution. As demonstrated in the analysis given in [8], the algorithm presented in [6] has an exponential number of steps, whereas the one given in [19] (with small modifications) has a finite number of steps ($\Omega(n^2 Max)$ steps, with Max the maximum height value of a node in the initial global state). 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) [22]. Recently, in [9] Cournier presented a new stabilizing solution for the construction of an arbitrary spanning tree improving the bound on the number of steps of [22]. 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) [11]. A spanning tree rooted at the source node with the information to propagate is constructed. This algorithm uses also a question mechanism to ensure that every processor in the network belongs to the constructed spanning tree, 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 [12]. 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.

There are many other works on the self-stabilizing construction of a spanning tree with additional properties, e.g., DFS tree [7, 10]. 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 $O(\log \Delta)$ bits [21], whereas Johnen proposes a self-stabilizing algorithm for the construction of a BFS tree using $O(\Delta)$ bits [20], with Δ the maximum node

degree in the network. A survey on several self-stabilizing constructions can be found in [18].

Contributions. In this paper, we present first a snap-stabilizing algorithm for 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. We propose then a snap-stabilizing algorithm based on the first one to resolve the problem of constructing a spanning tree. More precisely, our algorithm computes a BFS tree in $O(D^2)$ rounds with a polynomial number of steps of $O(mn^4)$ steps under a distributed daemon without any fairness assumptions, with D the diameter, m the number of edges and n the number of nodes in the network. To our knowledge, since in general the diameter of a network is much smaller than the number of nodes, this algorithm gets the best compromise of the literature between the complexities in terms of rounds and in terms of steps.

Outline of the paper. The paper is organized as follows. In Section 2 we present the model assumed in this paper. We then present the Question-Answer problem with a stabilizing algorithm for this problem in Section 3 and propose a stabilizing algorithm for the construction of a BFS tree in Section 4. We then 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 ($|V| = N$) and Δ its degree (i.e., the maximal value among the local degrees of the processors). 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. 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 which is an abstraction of the message-passing model. In this model, the program of every processor consists in a set of *shared variables* (henceforth, referred to as 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 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 *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 $\mathcal{C}$. A *computation* 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 computations considered here are assumed to be maximal. $\mathcal{E}$ is the set of all possible computations of P.

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 *weakly fairness*: if a daemon is *weakly fair*, then every continuously enabled processor is eventually chosen by the daemon to execute an action. We also use the notion of *unfairness*: the *unfair* daemon can forever prevent a processor to execute 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 definition of *round*. This definition captures the execution rate of the slowest processor in any computation. Given a computation 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.

3 Question-Answer problem

In this section, we first present the Question-Answer problem, then a snap-stabilizing algorithm is given to resolve this problem.

Let a static forest $\mathcal{F}$ of trees in a network $G = (V, E)$. Let some processors which request a permission to make a defined computation and a set of processors $AP \subset V$ authorized to deliver permissions. We consider a local predicate $Allowed(p)$ which indicates if processor p is in AP or not. Each processor $p \in AP$ is a root of a tree $T \in \mathcal{F}$, i.e., we have $Allowed(p) \Rightarrow root(T)$. Given a processor p in a tree $T \in \mathcal{F}$ which requests a permission, the *Question-Answer* problem is to deliver a permission (or *acknowledgement*) to p if and only if there is a processor $q \in AP$ such that q is the root of T.

We give a formal specification for the Question-Answer problem defined above.

Specification 1 (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 computation, 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 computation 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 1, that is no acknowledgement is sent to processors in a not allowed tree, from Property [Safety 2] of Specification 1.

3.1 Question-Answer algorithm

In this section, we present a snap-stabilizing algorithm for the Question-Answer problem, a formal description is given by Algorithm 1. This is a non-uniform algorithm because some rules are only executed by one or several processors $p \in V$ satisfying Predicate $Allowed(p)$.

Given a forest $\mathcal{F}$ of trees and a set of processors $Q \in V$ which *request* a permission. A *priority* is associated to each request when permission is requested at any processor $p \in V$. Algorithm 1 must transmit the request of each processor $p \in Q$ to the root $root(T)$ of the tree $T \in \mathcal{F}$ whose p belongs to. If processor $root(T)$ satisfies $Allowed(root(T))$ then an acknowledgement must be delivered to one of these p with the highest priority at least. This is to inform p that it is authorized to make a defined computation. By using correctly the priority at processors, we can ensure with this algorithm that eventually each processor determines if it belongs to a tree T rooted at a processor $root(T)$ which satisfies Predicate $Allowed(root(T))$. (Algorithm 2 given in Section 4 illustrate how to use Algorithm 1 to obtain this property.)

3.1.1 Variables

We define below the different variables used by Algorithm 1.

Shared variable. Each processor $p \in V$ has a local shared variable $p.Req$ which allows an external algorithm, called Algorithm $\mathcal{A}$, to monitor 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 Algorithm $\mathcal{A}$, 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 $WAIT$. At least the request of a requesting processor with the highest priority will reach the root of the tree and then receive a permission (an *acknowledgement*). When a processor p receives an acknowledgement, it sets the shared variable $p.Req$ to REP. Finally, Algorithm $\mathcal{A}$ must set the variable $p.Req$ to OUT to request another permission through Question-Answer algorithm to the root of the tree.

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.
- $p.PQ$: it stores the priority associated to the request sent or transmitted by processor p.

3.1.2 Algorithm description

Each processor $p \in V$ takes different inputs in Algorithm 1: $Neig_p$ gives the set of neighbors of p in the network, $Child(p)$ defines the set of children of p in the tree it belongs to, $Parent(p)$ is the parent of p

Algorithm 1 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 ;
$Priority(p)$: priority of p's local request;
Shared variable: $p.Req \in \{ASK, WAIT, REP, OUT\}$;
Variables: $p.Q \in \{R, W, A\}; p.PQ \in \mathbb{Z}$;

Macros:

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

Global Predicates:

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

Algorithm for p such that $Allowed(p)$:

Predicates:

$$WaitR(p) \equiv p.Q = R \wedge (\forall q \in Child(p) :: q.PQ = p.PQ \Rightarrow q.Q = W)$$
$$AnswerR(p) \equiv p.Q = W$$

Actions:

$$QE\text{-}action :: Error(p) \rightarrow p.Q := A; p.PQ := Priority(p);$$
$$QR\text{-}action :: Request(p) \rightarrow p.Q := R; p.PQ := Priority(p); p.Req = WAIT;$$
$$QRC\text{-}action :: RequestT(p) \rightarrow p.Q := R; p.PQ := (Ch_p).PQ;$$
if $p.PQ > Priority(p) \wedge p.Req = WAIT$ then $p.Req := ASK$; fi
$$QW\text{-}action :: WaitR(p) \rightarrow p.Q := W;$$
$$QA\text{-}action :: AnswerR(p) \rightarrow p.Q := A;$$
if $p.Req = WAIT$ then $p.Req := REP$; fi

Algorithm for p such that $\neg Allowed(p)$:

Predicates:

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

Actions:

$$QE\text{-}action :: Error(p) \rightarrow p.Q := A; p.PQ := Priority(p);$$
$$QR\text{-}action :: Request(p) \rightarrow p.Q := R; p.PQ := Priority(p); p.Req = WAIT;$$
$$QRC\text{-}action :: RequestT(p) \rightarrow p.Q := R; p.PQ := (Ch_p).PQ;$$
if $p.PQ > Priority(p) \wedge p.Req = WAIT$ 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

in the tree (if p satisfies $Allowed(p)$ then $Parent(p) = \perp$) and $Priority(p)$ is the priority (defined by Algorithm $\mathcal{A}$) of the request that p has to send to the root of its tree.

Any processor $p \in V$ in an allowed tree $T \in \mathcal{F}$ is informed that a permission is needed at p using the shared variable $p.Req$ setted to ASK by an external algorithm at p. In this case, we say that p has a *local request* to send to the root of the tree it belongs to. Before to send such a local request, p must verify if it has a child sending a request ($|PrioRC(p)| > 0$) with a higher priority than its local request, i.e., $Priority(p) \geq (Ch_p).PQ$ with Ch_p the child of p sending the request with highest priority among p's children (see Predicate $Request(p)$). If this is not the case, p executes $QR\text{-}action$ to set variables $p.Req, p.Q$, and $p.PQ$ to $WAIT, R$, and to $Priority(p)$ respectively to send its local request to the root. Moreover, the external algorithm is informed that the request is sent since $p.Req = WAIT$.

An internal processor p in the tree could have to transmit requests from its children (the request with highest priority first), only if no permission is received and used by the external algorithm at p (i.e., $p.Req \neq REP$). A processor p transmits a request from a child in the following cases:

- p has a child sending a request with a higher priority than the current request transmitted by p (i.e., $(Ch_p).PQ > p.PQ$);
- there is no request with a higher priority than the current one treated by p and p has received an acknowledgement also received by all its children waiting it. That is, the acknowledgement to a request of highest priority is no more needed at p (see Predicate $Transmit(p)$);
- p is waiting for an acknowledgement and a new request is transmitted by a child of p with the same priority than the current one transmitted by p (see Predicate $Retransmit(p)$).

In all these above cases, p executes $QRC\text{-}action$ to set $p.Q$ to R and $p.PQ$ to the highest priority among the requests that p have to treat (i.e., $p.PQ = (Ch_p).PQ$). If an internal processor q receives several requests from its children, then q transmits first the request with the highest priority (given by Macro Ch_q). Moreover, if p is sending a local request (i.e., $p.Req = WAIT$) then by executing $QRC\text{-}action$, to transmit a request with higher priority than its local request (i.e., $p.PQ > Priority(p)$), p sets its shared variable $p.Req$ to ASK to send later its local request when it is possible.

A processor p sending a local request transmitted by its parent (see Predicate $Wait(p)$) sets its variable $p.Q$ to W using $QW\text{-}action$ in order to notice that it waits for an acknowledgement for its local request. This status is propagated up in the tree to the root using $QW\text{-}action$. Note that every processor q which has sent a local request or transmitted a request from a child with the same priority as p's request waits for an acknowledgement, only if all children of q transmitting a request with the same priority are in Status W (see Predicate $Wait(p)$). Indeed, this allows to remove bad requests due to an incorrect initial configuration and to synchronize request transmissions of same priority.

Unless a request of higher priority than p's request is treated at the root $root(T)$ of the tree T, when $root(T)$ is in Status W then it delivers its permission to p's request (see Predicate $AnswerR$). To this end, since $root(T)$ is the root of an allowed tree (i.e., it satisfies $Allowed(root(T))$) then it executes $QA\text{-}action$ to set its variable $root(T).Q$ to A. This permission is propagated down in the tree. Any processor q waiting for the acknowledgement of p's request on the path between $root(T)$ and p in the tree T with a parent having an acknowledgement to p's request (i.e., $Parent(q).Q = A$ and $q.PQ = Parent(q).PQ = p.PQ$) also executes $QA\text{-}action$ to transmit the acknowledgement (see Predicate $Answer(q)$). Finally, p executes $QA\text{-}action$ to receive the acknowledgement to its local request, since $p.Req = WAIT$ then p sets the shared variable $p.Req$ to REP in order to notify to the external algorithm (Algorithm $\mathcal{A}$) that the permission is delivered for the local request at p. Moreover, if a child x of p is also waiting for an acknowledgement to a request with the same priority than p's local request, then x executes $QA\text{-}action$ too. Note that as soon as a received acknowledgement is no more needed at a processor p (i.e., $p.Req$ is setted to OUT by the external algorithm), then a request transmitted by a child of p can be transmitted by p up in the tree using $QRC\text{-}action$.

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 $p.PQ = Priority(p)$);
- p is transmitting a request from a child, but there is no child of p having a request with the same priority (i.e., $p.Q \neq A \wedge p.PQ \neq Priority(p) \wedge (\forall q \in Child(p), q.PQ = p.PQ \Rightarrow q.Q = A)$).

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 reinitiates p's state like if an acknowledgement to a local request was received (without changing the state of the shared variable $p.Req$), i.e., to set $p.Q$ to A and $p.PQ$ to $Priority(p)$.

3.2 Proof of Question-Answer algorithm

3.2.1 Definitions

Definition 1 (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 2 (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 3 (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 $Priority(p_i) > Priority(p_j)$ with $Priority(p)$ be the priority given in input of Algorithm 1 at processor p. 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 $Priority(p_j) < Priority(p_i)$.*

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.

3.2.2 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., $(\forall x \in T :: x.Req = ASK \wedge Priority(p) > Priority(x))$. 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 $q.PQ \neq p.PQ$. Moreover, since p's request is of highest priority in T then q satisfies $p.PQ = (Ch_q).PQ$ and $|PrioRC(q)| > 0$. We assume that $p.Req \neq REP$, otherwise by hypothesis $p.Req$ is setted to OUT in finite time. In this case,

either we have $(p.PQ > q.PQ \Rightarrow RequestT(q))$ and $QRC\text{-}action$ is enabled at q, a contradiction. Otherwise, q has transmitted a request with the same priority, i.e., we have $p.PQ = q.PQ$ 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, $p.Q = R, p.PQ = Priority(p)$ and $p.Req = WAIT$ at p and $\forall q \in \mathcal{P}(root(T), p), q.PQ = p.PQ$. 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, $\forall x \in \mathcal{P}(root(T), p), x.PQ = p.PQ \wedge x.Q = R$. Then, we have $(Parent(p).PQ = p.PQ \wedge p.PQ = Priority(p)) \Rightarrow Wait(p)$ 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, $\forall x \in \mathcal{P}(root(T), p), x.PQ = p.PQ \wedge x.Q = W$. 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 (or $WaitR(p) \Rightarrow p.Q = R$ if $p = root(T)$). □

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., $Priority(p) \geq (Ch_p).PQ$). $QR\text{-}action$ 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 $p.PQ \neq Priority(p)$. 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 1, 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 $Priority(p) \geq (Ch_p).PQ$). 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 $(Ch_p).PQ \leq p.PQ$ and we must consider two subcases at p: $p.Q \neq A$ or $\exists s \in Child(p)$ such that $s.Q = W \wedge s.PQ = p.PQ$. Either $p.Q \neq A$ then this implies that $(Ch_p).PQ = p.PQ$ and p has already transmitted the request of q, a contradiction. Or $\exists s \in Child(p)$ such that $s.Q = W \wedge s.PQ = p.PQ$. This implies that either $s = q$ and p has already transmitted q's request or $s \neq q$ and $s.PQ > q.PQ$, a contradiction because $(Ch_p) = q$. Thus, QR-$action$ or QRC-$action$ 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-$action$ or QRC-$action$ 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-$action$ or QRC-$action$ are not the action of highest priority at p then this implies that QE-$action$ is always enabled. According to Lemma 1, after executing QE-$action$ it is not enabled at p (unless QR-$action$ or QRC-$action$ is executed), a contradiction. So, QE-$action$ is disabled at p. According to Lemma 2, QR-$action$ is enabled until it is executed at every processor $p \in T$ having a request of priority higher than its children requests (i.e., $Priority(p) \geq (Ch_p).PQ$). Otherwise, we have QR-$action$ is disabled. Therefore, since QE-$action$ and QR-$action$ are disabled then QRC-$action$ 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 1 *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 1, to wait for an acknowledgement to a request a processor executes QW-$action$. 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-$action$ is disabled or it is not the enabled action of highest priority at p.

We first show that QW-$action$ is enabled at p. According to Lemma 3, for processor p we have $p.Q = R$, $p.PQ = Priority(p)$, and $p.Req = WAIT$ if p has sent a local request, or $p.Q = R$, $p.PQ \neq Priority(p)$, 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., $\neg[Parent(p).Q = R \wedge Parent(p).PQ = p.PQ]$ or $\exists q \in Child(p)$ such that $q.PQ = p.PQ \wedge q.Q \neq W$). Note that for $root(T)$ only the second case must be considered. If $\neg[Parent(p).Q = R \wedge Parent(p).PQ = p.PQ]$ 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 $q.PQ = p.PQ \wedge q.Q \neq W$. Then there is a path $\mathcal{P}(p, s)$ in T such that $x.PQ = Priority(s) = q.LQ$ 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-$action$ 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-$action$ then $q.Q = W$, a contradiction.

We must show that QW-$action$ 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-$action$ is not the enabled action of highest priority at p. Suppose that QE-$action$ is the enabled action of highest priority at p. According to Lemma 1, after executing QE-$action$ it is not enabled at p, a contradiction. So, QE-$action$ is disabled at p. Suppose that QR-$action$ or QRC-$action$ 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 2 *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 1, a processor $p \in T$ waiting for an acknowledgement executes QRC-$action$ 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-$action$ to transmit again the request with the same priority. Either for every child q of p we have $q.Q \neq R$ or $q.PQ \neq p.PQ$ because $(\forall q \in Child(p) :: q.Q \neq R \vee q.PQ \neq p.PQ) \Rightarrow \neg Retransmit(p)$. 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 $q.PQ \neq p.PQ$ then the request to transmit again by p is not of highest priority in p's neighborhood (i.e., $(Ch_p).PQ \neq q.PQ$), 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 1, to transmit the acknowledgement to a request a processor executes QA-$action$. 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-$action$ is disabled or it is not the enabled action of highest priority at p.

We first show that QA-$action$ is enabled at p. According to Lemma 4, for processor p we have $p.Q = W$, $p.PQ = Priority(s)$, and $p.Req = WAIT$ if $p = s$, or $p.Q = W$, $p.PQ = Priority(s) \neq Priority(p)$ 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-$action$ 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-$action$ is disabled at p then either $Parent(p).Q \neq A$ or $Parent(p).PQ \neq p.PQ$ 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 $p.Q = W$, $p.PQ = Priority(s)$. 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-$action$ is enabled for every processor $q \in \mathcal{P}(root(T), s)$.

We must show that QA-$action$ is the enabled action of highest priority for every processor $p \in \mathcal{P}(root(T), s)$. Assume, by the contradiction, that QA-$action$ is not the enabled action of highest priority at p. According to Lemma 1, after executing QE-$action$ it is not enabled at p, a contradiction. So, QE-$action$ is disabled at p. Suppose that QR-$action$ or QRC-$action$ 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-$action$ 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 1, to receive an acknowledgement to a request in an allowed tree T a processor executes QA-$action$. 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 $Answer(p) \Rightarrow p.Q = W$ (or $AnswerR(p) \Rightarrow p.Q = W$, if $p = root(T)$). 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 $Wait(p) \Rightarrow p.Q = R$ (or $WaitR(p) \Rightarrow p.Q = R$, if $p = root(T)$). So, by formal description of Algorithm 1 $QR\text{-}action$ 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 $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.PQ = Priority(p)$ 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 $p.PQ = Priority(p)$ 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 $p.PQ = Priority(p)$. 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 $p.PQ = Priority(p)$ 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 $q.Q = R \wedge q.PQ = Priority(p)$. 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 $(x.Q = A \wedge x.PQ = Priority(x)) \Rightarrow \neg Error(x)$ 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 $q.PQ = Priority(p)$. Either $Priority(p) > x.PQ$, then $QRC\text{-}action$ is enabled at x in round $j + 1$. Or $Priority(p) \leq x.PQ$, then as the daemon is weakly fair we have $(\forall s \in Child(x) :: s.Q = W \Rightarrow s.PQ \neq x.PQ)$, 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 $x.PQ = Priority(p)$ 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 $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.PQ = Priority(p)$. □

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 $p.Req = WAIT \wedge p.Q = R \wedge p.PQ = Priority(p)$ at processor $p \in T$ at height k then in at most $k + 1$ rounds we have $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.PQ = Priority(p)$.

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 $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.PQ = Priority(p)$,

then in at most $j+1$ rounds we have $\forall q \in \mathcal{P}(root(T),p), q.Q = W \wedge q.PQ = Priority(p)$ at height $\geq k-j$ in T.
In the base case $j = 0$ and we consider p. We have $(\forall q \in \mathcal{P}(root(T),p), q.Q = R \wedge q.PQ = Priority(p))$, in particular for $Parent(p)$ and p. Thus, the proposition is verified for p because QW-$action$ is enabled at p in round 0, and in the first configuration of round 1 we have $p.Q = W$ and $p.PQ = Priority(p)$ 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-$action$ 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 $q.Q = W \wedge q.PQ = Priority(p)$. 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 $s.Q = W \wedge s.PQ = Priority(p)$. Thus, $\exists s \in Child(q), s.Q = W \wedge s.PQ = q.PQ$, and $q.Q = R$ so QW-$action$ is enabled at q in round $j+1$. So, since the daemon is weakly fair we have $q.Q = W$ and $q.PQ = Priority(p)$ 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 $\forall q \in \mathcal{P}(root(T),p), q.Q = W \wedge q.PQ = Priority(p)$. □

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 $\forall q \in \mathcal{P}(root(T),p), q.Q = W \wedge q.PQ = Priority(p)$.

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-$action$ is enabled at q, and $\forall q \in \mathcal{P}(root(T),p), q.Q = W \wedge q.PQ = Priority(p)$, then in at most $j+1$ rounds we have $x.Q = A \wedge x.PQ = Priority(p)$ 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 $(\forall q \in \mathcal{P}(root(T),p), q.Q = W \wedge q.PQ = Priority(p))$, in particular for $root(T)$. The proposition is verified for x because we have $(x.Q = W \Rightarrow AnswerR(x))$ and QA-$action$ is enabled at x in round 0. Thus, in the first configuration of round 1 we have $x.Q = W \wedge x.PQ = Priority(p)$ 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-$action$ is enabled at q, then in round $j+1$ for processor $x \in \mathcal{P}(root(T),p)$ at height $j+1$, we have $x.Q = A \wedge x.PQ = Priority(p)$. By induction hypothesis, in the first configuration of round $j+1$ we have $Parent(x).Q = A \wedge Parent(x).PQ = x.PQ \wedge x.Q = W$ at x, so QA-$action$ is enabled at x in round $j+1$. Therefore, since the daemon is weakly fair we have $x.Q = A$ and $x.PQ = Priority(p)$ 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 1, if $x.PQ = Priority(x)$ when QA-$action$ is executed at x then we have $x.Req = REP$. So we have $x = p$, and in most $k+1$ additional rounds we have $p.Req = REP \wedge p.Q = A \wedge p.PQ = Priority(p)$.
□

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 1 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 1 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 1. 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 1.*

Proof. We have to show that starting from any configuration the execution of Algorithm 1 verifies all the properties of Specification 1.

We first show that Property [Liveness 1] 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 1. 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 1.

We now show that Property [Liveness 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 1 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 1.

We now show that Property [Safety 1] 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 1.

We now show that Property [Safety 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 $root(T).Q = W \wedge root(T).PQ \neq Parent(root(T)).PQ$ then $Parent(root(T))$ executes $QRC\text{-}action$ (because $Transmit(Parent(root(T))) \Rightarrow RequestT(Parent(root(T)))$), a contradiction. Otherwise, we have $Parent(root(T)).Q = A \wedge (\forall q \in Child(Parent(root(T))), q.Q = W \wedge q.PQ = Parent(root(T)).PQ)$ given by an initial configuration of the system, a contradiction. This satisfies Property [Safety 2] of Specification 1. □

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

Theorem 2 *Algorithm 1 is snap-stabilizing for Specification 1 under a weakly fair daemon.*

3.2.3 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 $q.PQ = Priority(q)$ 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 $q.PQ = Priority(p)$ 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 $s.PQ = q.PQ \wedge s.Q \neq A$ (in case of $p, p.Req = WAIT$); $QR\text{-}action$ because $q.Req \neq ASK$ or $Priority(q) < q.PQ = (Ch_q).PQ$; and $QRC\text{-}action$ because $q.Q = R \wedge (\forall s \in Child(q), s.Q = W \wedge s.PQ = q.PQ)$. 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 $(Ch_q).PQ > q.PQ \Rightarrow RequestT(q)$. 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 3 *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. □

4 Spanning Tree Construction

In this section, we are interested in to the problem of constructing a tree spanning all the processors of the network. To this end, we give a snap-stabilizing algorithm which uses the algorithm presented in the precedent section as a black box. Moreover, we consider there is 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 4.

Definition 4 (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 path in T between any pair of nodes $x, y \in V_T$) and $|E_T| = |V| - 1$, and*
3. *For each node $p \in V_T$, there exists no shorter path (in hops) between p and r in G than the path between p and r in T.*

We give a formal specification to the problem of constructing a stabilizing BFS tree, stated in Specification 2.

Specification 2 (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] *Every configuration $\gamma \in \mathcal{T}$ satisfies Definition 4.*

4.1 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 which solves the Question-Answer problem (see Section 3) and Algorithm 2 which allows to a processor to connect to a tree. These two algorithms are executed concurrently at each processor $p \in V$. To construct a BFS tree, Algorithm 2 plays the role of Algorithm $\mathcal{A}$ for Algorithm 1 as described in Section 3. That is, Algorithm 2 interrogates Algorithm 1 to obtain permissions allowing processors to connect correctly in order to construct a BFS tree. More precisely, given a forest of trees $\mathcal{F}$ we must designate for every processor $p \in V$ with Predicate $Allowed(p)$ the set of processors which are allowed to deliver permissions, according to Algorithm 1. For the construction of a BFS tree rooted at processor r, we must define that $Allowed(p) \equiv (p = r)$ for every processor $p \in V$. The idea behind this is to only authorize processor connections to the tree rooted at r and to forbid the connections to the other trees of forest $\mathcal{F}$. Each processor $p \in V$ has a

status in Algorithm 2 which is used to notify if p belongs to the tree rooted at r or not. When a processor $p \neq r$ determines it is the root of a tree in $\mathcal{F}$, then it informs the processors in its tree that they are not in the tree rooted at r and these processors set their status to E. In the same way, each processor in the tree rooted at r sets its status to C. Thereby, when a processor p in the tree rooted at r (i.e., in Status C) detects a neighbor q in Status E or whose q's parent level is bigger than p's level, then Algorithm 2 generates a local request at p to obtain a permission, delivered by Algorithm 1. The computation associated to a delivered permission at a processor p is to authorize its neighbors to connect to p. Therefore, if a permission is given to processor p then its neighbors q can execute Algorithm 2 to join the tree p belongs to.

Algorithm 1 can be viewed as a synchronizer allowing the BFS tree construction layer by layer, the addition of any new layer of processors depending of a permission request. It is easy to see that this construction needs $O(D^2)$ rounds. In another 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 neighbour 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). In fact, the role of the deleting part is to ensure that any processor cannot hook the same abnormal tree arbitrarily often and finally we limit the step complexity to $O(mn^4)$ (see Lemma 29).

4.1.1 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 is used by Algorithm 2 to monitor Algorithm 1 at p. This shared variable can take four values: $ASK, WAIT, REP$, and OUT. By setting the shared variable $p.Req$ to ASK, Algorithm 2 informs Algorithm 1 that a permission from the root of the tree that p belongs to is needed at p. In this case, Algorithm 1 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). If a permission is delivered to processor p, then Algorithm 1 sets this shared variable to REP in order to inform Algorithm 2. Then, every neighbor of p can execute Algorithm 2 to join the tree that p belongs to. When there is no neighbor of p to connect, then Algorithm 2 sets $p.Req$ to OUT which allows to Algorithm 2 to request another permission through Algorithm 1 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 to a processor x satisfying Predicate $Allowed(x)$, C otherwise. We have $p.S = C$ for processor $p = r$.

4.1.2 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 2). 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 4 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., $\forall q \in Neig_p, |q.L - p.L| < 1$). 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 1 sends a request to the root of the tree.

Inputs for Algorithm 1. In order to allows Algorithm 1 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) $Priority(p)$ is the priority of the local request of p which is equal to the opposite of p's level in the tree for the task of constructing a BFS tree (i.e., $Priority(p) \equiv -p.L$), and (iv) $Allowed(p)$ is a predicate which notifies if p can deliver permissions (i.e., $Allowed(p) \equiv (p = r)$). Remind that $Allowed(p)$ must be satisfied only at processor $p = r$ in Algorithm 1 to allow that eventually every processor joins the tree rooted at r, since eventually the processors cannot join another tree in 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 its authorization to q to connect 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 $p.P$ and $p.L$ according to its new parent in the tree, and it changes its status to Status C and its shared variable $p.Req$ 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 1 that the permission can be removed at p, then this allows p to ask a new permission later.

Note that a request mechanism was also used in previous stabilizing spanning tree construction algorithms [1, 11]. However the mechanism in [11] gives strong guarantees due to the PIF task. Indeed, the nodes in abnormal trees are frozen and these nodes can leave the tree after the agreement of the root. The goal is to insure that every processor of the network belongs to the same tree before the propagation of the information. In our approach, the nodes in an abnormal tree are autonomous to take a decision and to leave the tree if possible, which leads to a time complexity in terms of rounds independent of the number of network nodes. Moreover, contrary to [1], abnormal trees are detected more efficiently since the detection is done locally and not using node identifiers. This allows to avoid a part of useless node additions.

Composition. Algorithm $\mathcal{BFS}$ is obtained by composition of Algorithm 1 and Algorithm 2. These two algorithms are composed together at each processor $p \in V$ with a conditional composition (first

introduced in [13]): Algorithm 2 $\circ\ |_{Cond(p)}$ Algorithm 1, where each guard g of the actions of Algorithm 1 at each processor $p \in V$ has the form $Cond(p) \wedge g$ with Predicate $Cond(p)$ defined below (see Algorithm 2 for the description of predicates): $Cond(p) \equiv \ GoodT(p) \wedge GoodL(p)$.

Using this composition, each processor $p \in V$ can execute Algorithm 1 (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). Moreover, actions of Algorithm 1 and Algorithm 2 can be enabled at p simultaneously. In this case, Algorithm 1 is executed before Algorithm 2 at processor p.

Algorithm 2 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:

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

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:
$E\text{-}action :: AbnormalTree(p) \rightarrow p.S := E;$
$C\text{-}action :: Connect(p) \rightarrow p.S := C; p.P := MinChPar(p); p.L := (p.P).L + 1; p.Req := OUT;$
$A\text{-}action :: Start(p) \rightarrow p.Req := ASK;$
$O\text{-}action :: End(p) \rightarrow p.Req := OUT;$

4.2 Proof of Spanning Tree algorithm

4.2.1 Definitions

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

Definition 5 (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 $\exists \mathcal{P}(p, q)$.*

Definition 6 (Normal tree) *A tree T rooted at processor $root(T)$ containing only processors p such that $(p = root(T) \wedge Allowed(p)) \vee (p.S = C \wedge p.L = (p.P).L + 1)$ is called a* normal tree. *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 7 (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: $p.S = C \wedge p.L = (p.P).L + 1 \wedge \neg GP\text{-}REP(p)$.*

4.2.2 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 4. $\forall \gamma \in (\mathcal{C} - \mathcal{B}), \exists p \in V$ 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 2, 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 5), 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 $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.PQ = Priority(p)$.
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 $p.Q = R \wedge p.PQ = Priority(p)$, 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 $q.PQ = Priority(p)$. 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 $\forall q \in \mathcal{P}(root(T), p), (Priority(q) \leq j+1 \Rightarrow (q.Q = R \wedge q.PQ = Priority(p)))$. 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 $\forall q \in \mathcal{P}(root(T), p), (Priority(q) \leq j + 1 \Rightarrow (q.Q = R \wedge q.PQ = Priority(p)))$. Thus, in at most $j^2 + (j - 1) + (j + 1) < O(j^2)$ rounds we have $\forall q \in \mathcal{P}(root(T), p), (Priority(q) \leq j + 1 \Rightarrow (q.Q = R \wedge q.PQ = Priority(p)))$, 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 $\forall q \in \mathcal{P}(root(T), p), q.Q = R \wedge q.PQ = Priority(p)$. Thus, we can apply Lemma 10 and in at most $k + 1$ additional rounds we have $p.Req = REP \wedge p.Q = A \wedge p.PQ = Priority(p)$ 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 4, 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 1, 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 distance 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 $q \notin Tree(r), q \in T$. 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 1 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 subcases: (B1) $d_G(root(T), q) = j + 1 = 2$ or (B2) $d_G(root(T), q) > 2$.
  - In the subcase (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 subcase (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 1 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 1 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 1 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 subcases: (B1) $d_G(root(T), q) = j + 1$ or (B2) $d_G(root(T), q) > j + 1$.

  – In the subcase (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 subcase (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 1 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 1 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 4. 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 4. To show the last Claim of Definition 4, 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 4 is verified, which finishes to show the lemma. □

Corollary 4 *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.*

Lemma 20 *In every configuration $\gamma \in \mathcal{C}$ satisfying Definition 4, for every processor $p \in V$ no action of Algorithm 2 is enabled in γ.*

Proof. Observe first that since γ satisfies Definition 4, 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 5 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 7).

Assume, by the contradiction, that $\exists \gamma \in \mathcal{C}$ satisfying Definition 4 such that $\exists p \in V$ with an enabled action of Algorithm 2 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 20, we have the following corollary.

Corollary 5 *In every configuration $\gamma \in \mathcal{C}$ satisfying Definition 4, every action of Algorithm $\mathcal{BFS}$ is disabled at each processor $p \in V$ in γ.*

Lemma 21 *From any configuration, the execution satisfies Specification 2.*

Proof. We have to show that starting from any configuration the execution of Algorithm $\mathcal{BFS}$ verifies Property [TC1] and [TC2] of Specification 2.

According to Lemma 19 and Corollary 5, 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 2. Moreover, according to Lemma 19 the terminal configuration γ reached by Algorithm $\mathcal{BFS}$ satisfies Definition 4, which verifies Property [TC2] of Specification 2. □

Theorem 3 and Lemma 21 imply the following theorem.

Theorem 4 *Algorithm $\mathcal{BFS}$ is snap-stabilizing for Specification 2 under a weakly fair daemon.*

4.2.3 Proof assuming an unfair daemon

Definition 8 (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-$action$, or p executes C-$action$.*

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

Remark 4 *E-$action$, C-$action$, and A-$action$ are disabled at a locally healthy processor $p \in V$.*

Proposition 1 *Every processor $p \in V$ 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 2, a processor hooks on to a neighbor using C-$action$. 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 1 s never receives an acknowledgement and we have that $s.Req \neq REP \Rightarrow \neg Connect(p)$, otherwise C-$action$ is enabled at p, a contradiction. If s is in a normal tree then by Property [Liveness 2] of Specification 1 we have that $s.Req = REP$ and C-$action$ is enabled at p, a contradiction. □

Lemma 22 *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 2, a processor p must execute C-$action$ 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-$action$ and they are hooked on to T in configuration γ_k. Note that after executing C-$action$, 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 1 because q can not receive an acknowledgement from $root(T)$ since T is an abnormal tree. □

Corollary 6 *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 22, 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 1. □

Lemma 23 *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-$action$, 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 24 *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 23, 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 8 the lemma follows. □

Lemma 25 *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 8 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 26 *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 24 and 25. □

Corollary 7 *From any configuration, Algorithm 2 produces at most $2\Delta n + n^2$ topological changes in forest $\mathcal{F}$.*

Lemma 27 *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 8, 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 28 *From any configuration, Algorithm 2 produces at most $2\Delta m + mn$ requests to reach a configuration satisfying Definition 4.*

Proof. This comes from Corollary 7 and Lemma 27. □

Corollary 8 *In an execution, A-$action$ and O-$action$ are executed at most $2\Delta m + mn$ times in the network.*

Lemma 29 *From any configuration, at most $O(\Delta mn^3 + mn^4)$ steps are needed by Algorithm $\mathcal{BFS}$ to reach a configuration satisfying Definition 4.*

Proof. By Corollary 7, from any configuration Algorithm 2 generates at most $2\Delta n+n^2$ topological changes to reach a configuration satisfying Definition 4. Thus, by Definition 8 this implies that E-$action$ and C-$action$ are executed at most $\Delta n + n^2$ times. Moreover, by Corollary 8 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 3, 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. □

5 Conclusion

In this paper a silent snap-stabilizing algorithm resolving the Question-Answer problem has been given, in which each node requests a permission (delivered by a subset of network nodes) in order to perform a defined computation. Based on this first algorithm, a silent snap-stabilizing algorithm for the construction of a Breadth First Search tree has been presented. The complexity of this algorithm in terms of rounds is $O(D^2)$ and in terms of steps is $O(mn^4)$, with D the diameter, m the number of edges and n the number of nodes in the network. Moreover, a distributed daemon without any fairness assumptions is considered. To our knowledge, since in general the diameter of a network is much smaller than the number of nodes, the presented BFS construction algorithm gets the best compromise of the literature between the complexities in terms of rounds and in terms of steps.

References

[1] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilizing protocols for general networks. In Springer, editor, *4th International Workshop on Distributed Algorithm (WDAG)*, volume LNCS 486, pages 15–28, 1991.

[2] Anish Arora and Mohamed G. Gouda. Distributed reset (extended abstract). In *10th Conference on Foundations of Software Technology and theoretical Computer Science (FSTTCS)*, pages 316–331, 1990.

[3] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 652–661, 1993.

[4] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing pif in tree networks. In Anish Arora, editor, *Workshop on Self-stabilizing Systems (WSS)*, pages 78–85. IEEE Computer Society, 1999.

[5] Janna Burman and Shay Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *21st International Symposium on Distributed Computing (DISC)*, pages 92–107, 2007.

[6] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Inf. Process. Lett.*, 39(3):147–151, 1991.

[7] Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Inf. Process. Lett.*, 49(6):297–301, 1994.

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

[9] Alain Cournier. A new polynomial silent stabilizing spanning-tree construction algorithm. In Shay Kutten and Janez Zerovnik, editors, *16th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 5869 of *Lecture Notes in Computer Science*, pages 141–153. Springer, 2009.

[10] Alain Cournier, Stéphane Devismes, and Vincent Villain. A snap-stabilizing dfs with a lower space requirement. In Ted Herman and Sébastien Tixeuil, editors, *7th International Symposium on Self-Stabilizing Systems (SSS)*, volume 3764 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2005.

[11] Alain Cournier, Stéphane Devismes, and Vincent Villain. Snap-stabilizing pif and useless computations. In *12th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 39–48. IEEE Computer Society, 2006.

[12] Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1), 2009.

[13] Ajoy Kumar Datta, Shivashankar Gurumurthy, Franck Petit, and Vincent Villain. Self-stabilizing network orientation algorithms in arbitrary rooted networks. *Stud. Inform. Univ.*, 1(1):1–22, 2001.

[14] Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space. In Sandeep S. Kulkarni and André Schiper, editors, *10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 5340 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2008.

[15] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[16] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[17] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *9th ACM symposium on Principles of distributed computing (PODC)*, pages 103–117, 1990.

[18] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, EPFL, October 2003.

[19] Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett.*, 41(2):109–117, 1992.

[20] Colette Johnen. Memory-efficient self-stabilizing algorithm to construct bfs spanning trees. In *3rd Workshop on Self-stabilizing Systems (WSS)*, pages 125–140, 1997.

[21] Colette Johnen and Joffroy Beauquier. Distributed self-stabilizing depth-first token circulation with constant memory. In *2nd Workshop on Self-Stabilizing System (WSS)*, pages 4.1–4.15, 1995.

[22] Adrian Kosowski and Lukasz Kuszner. A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski, editors, *6th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, volume 3911 of *Lecture Notes in Computer Science*, pages 75–82. Springer, 2005.

[23] Gerard Tel. *Introduction to distributed algorithm*. Cambridge University Press, Second edition, 2000.