Fast and Compact Self Stabilizing Verification, Computation, and Fault Detection of an MST

Amos Korman*
CNRS & U. Paris Diderot
Paris, France
Amos.Korman@liafa.jussieu.fr

Shay Kutten†
Faculty of IE&M, The Technion
Haifa, Israel
kutten@ie.technion.ac.il

Toshimitsu Masuzawa
IS&T Grad. Center, Osaka U.
Osaka, Japan
masuzawa@ist.osaka-u.ac.jp

ABSTRACT

This paper demonstrates the usefulness of distributed local verification of proofs, as a tool for the design of algorithms. In particular, it introduces a somewhat generalized notion of distributed local proofs, and utilizes it for improving the memory size complexity, while obtaining time efficiency too.

As a result, we show that optimizing the memory size carries at most a small cost in terms of time, in the context of Minimum Spanning Tree (MST). That is, we present algorithms that are both time and space efficient for constructing an MST, for verifying it, and for detecting the location of the faults. This involves several steps that may be considered contributions in themselves.

First, we generalize the notion of local proofs, trading off the locality (or, really, the time complexity) for memory efficiency. This adds a dimension to the study of distributed local proofs, that has been gaining attention recently.

Second, as opposed to previous studies that presented only the labels verification part of a proof labeling schemes, we present here also a space and time efficient distributed self stabilizing marker algorithm to generates those labels. This presents proof labeling schemes as an algorithmic tool.

Finally, we show how to enhance a known transformer that makes input/output algorithms self stabilizing. It now takes as input an efficient construction algorithm and an efficient self stabilizing proof labeling scheme, and produces an efficient self stabilizing algorithm.

When used for MST, the transformer produces a memory optimal (i.e., $O(\log n)$ bits per node) self stabilizing algorithm, whose time complexity, namely, $O(n)$, is significantly better even than that of previous algorithms that where not space optimal. (The time complexity of previous MST algorithms that used $\Omega(\log^2 n)$ memory bits per node was $O(n^2)$, and the time for optimal space algorithms was $O(n|E|)$.)

Our MST algorithm also has the important property that, if faults occur after the construction ended, then they are detected by some nodes within $O(\log^2 n)$ time in synchronous networks, or within $O(\triangle \log^2 n)$ time in asynchronous ones. This property is inherited from the specific proof labeling scheme we construct. It answers an open problem posed by Awerbuch and Varghese (FOCS 1991). We also show that $\Omega(\log n)$ time is necessary if the memory size is restricted to $O(\log n)$ bits, even in synchronous networks.

Another property is that if f faults occurred, then, within the required detection time above, they are detected by some node in the $O(f \log n)$ locality of each of the faults. We also show how to improve the above detection time and locality, at the expense of some increase in the memory.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms, Graph labeling, Network problems*; E.1 [**Data Structures**]: *Distributed data structures*

General Terms

Algorithms

Keywords

Distributed algorithm, self stabilization, MST, distributed verification, local proof checking, fault detection, locality.

1. INTRODUCTION

In a non-distributed context, solving a problem is believed to be, sometimes, much harder than verifying it (e.g. for NP-Hard problems). Given a tree T, a task introduced by Tarjan [36] is to verify that T is indeed an MST. This non-distributed verification seems to be just slightly easier than the non-distributed computation of an MST. On the other hand, in the distributed context, the time complexity of an MST verification can be 1, when using $\Theta(\log^2 N)$ bits per node. In [29, 31], the MST was assumed to be represented distributively, such that each node stores a pointer to its parent. Similarly, the verification (termed a *proof labeling scheme*) assumed that each node stored some information, the node's *label*, to be used for verification. If the collection

*Supported in part by a France-Israel cooperation grant ("Mutli-Computing" project) from the French and Israeli Ministries of Science, by the ANR projects ALADDIN and PROSE, and by the INRIA project-team GANG.

†Supported in part by a France-Israel cooperation grant ("Mutli-Computing" project) from the French and the Israeli Ministries of Science, by a grant from the Israel Science Foundation, and by a grant from the Gordon Center at the Technion.

PODC'11, June 6–8, 2011, San Jose, California, USA.

of these (node, parent) edges was not an MST, then at least one node *raised an alarm* in one time unit.

To make such a proof labeling scheme a useful algorithmic tool, one needs to present a *marker* algorithm for computing those labels. One of the contributions of the current paper is a time and memory efficient self stabilizing marker.

Every decidable graph property (not just an MST) can be verified in a short time given a large enough memory [31]. A second contribution is a generalization of such schemes to allow a reduction in the memory requirements, by trading off the locality (or the time). In the context of MST, yet another (third) contribution is a reduced space proof labeling scheme for MST. It uses just $O(\log n)$ bits of memory per node (the same as the amount needed for merely representing distributively the MST). This is below the lower bound of $\Omega(\log^2 n)$ of [29]. The reason this is possible is that the verification time is increased to $O(\log^2 n)$ in synchronous networks and to $O(\triangle \log^2 n)$ in asynchronous ones. Another important property of the new scheme is that any fault is detected rather close to the node where it happened.

Given a long enough time, one can verify T by recomputing the MST. An open problem posed by Awerbuch and Varghese [9] is to find an MST verification algorithm whose time complexity is smaller than the MST computation time, yet with a small memory. The above mentioned third contribution solves this open problem by showing a $Polylog(n)$ time penalty when using $O(\log n)$ memory size for the MST verification algorithm. In contrast, lower bounds which are polynomial in n for MST construction follow from [35, 34] (even for constant diameter graphs). Interestingly, it turns out that a logarithmic penalty is unavoidable. That is, we show that $\Omega(\log n)$ time for an MST verification scheme is necessary if the memory size is restricted to $O(\log n)$ bits, even in synchronous networks.

One known application of some methods of distributed verification is for general transformers, that transform non-stabilizing algorithms to produce self stabilizing ones. The fourth contribution of this paper is an adaptation of the transformer of [9] such that it can transform algorithms in our context (asynchronous network of unknown size and diameter, where the verification method is a proof labeling scheme whose verifier part is self stabilizing). Based on the strength of the original transformer of [9] (and that of its companion paper [8] it uses), our adaptation yields a result that is rather useful even without plugging in the new verification scheme. This is demonstrated by plugging in the proof labeling schemes of [31, 29], yielding an algorithm which already improves the time of previous $O(\log^2 n)$ memory self stabilizing MST construction algorithms, and also detects faults using 1 time and at distance f at most from the faults (if f faults occurred).

Finally, we obtain an optimal memory size, $O(n)$ time asynchronous self stabilizing MST construction algorithm. The state of the art time bound for such optimal memory algorithms was $O(n|E|)$ [10, 25]. In fact, our time bound improves significantly even the best time bound for algorithms using polylogarithmic memory, which was $O(n^2)$ [11].

Moreover, our self stabilizing MST algorithm inherits two important properties from our verification scheme, which are: (1) the time it takes to detect faults is small: $O(\log^2 n)$ time in a synchronous network, or $O(\triangle \log^2 n)$ in asynchronous ones; and (2) if some f faults occur, then each fault is detected within its $O(f \log n)$ neighborhood.

Outline: The MST verification and the MST construction algorithms are given in Sections 3 and 4. Throughout, to save space, many details and proofs are deferred to the full paper (some of which can be found also in [30]). This includes, for example, the logarithmic time lower bound.

The intuition behind the main technical problem: Informally, in [29, 31], each node v stores some $\log n$ "pieces" of information, each of size $\log n$. As explained later (Section 2), Node v uses its own pieces, as well as the pieces of its neighbors to verify the MST. In the current paper, each node has room for only a constant number of such pieces. One immediate idea is to store some of v's pieces in some other nodes. Whenever v needs a piece, some algorithm should move it for v. Moving pieces would cost time, hence, realizing some time versus memory size trade-off.

Unfortunately, the total (over all the nodes) number of pieces in the schemes of [31, 29] is $\Omega(n \log n)$. Any way one would assign these pieces to nodes would result in the memory of a single node needing to store $\Omega(\log^2 n)$ bits. Hence, our first technical step was to reduce the total number of pieces to $O(n)$, so that we could store at each node just a constant number of such pieces. However, each node still needs to use $\Omega(\log n)$ pieces. That is, each piece may be needed by many nodes. Next, we solve a combinatorial problem: locate each piece "close" to each of the nodes needing it, while storing only a constant number of pieces per node.

The solution of this combinatorial problem would have sufficed to construct the desired scheme in the local model. There, Node v can "see" the storage of nearby nodes [26]. However, in the congestion aware model, one actually needs to move pieces from node to node, while not violating the $O(\log n)$ memory per node constraint. This is difficult, since, at the same time v needs to see its own pieces, other nodes need to see their own ones.

Additional Related work: The distributed construction of an MST has yielded techniques and insights that were used in the study of many other problems of distributed network protocols. It has also become a standard to check a new paradigm in distributed algorithms theory. The first distributed algorithm was proposed by [15], its complexity was not analyzed. The seminal paper of Gallager, Humblet, and Spira presented a message optimal algorithm that used $O(n \log n)$ time, improved by Awerbuch to $O(n)$ time [21, 5], and later to $O(D + \sqrt{n} \log^* n)$ [22, 33]. This was coupled with an almost matching lower bound of $\Omega(\sqrt{n})$ [35].

Self stabilization [18] deals with algorithms that must cope with faults that are rather severe, though of a type that does occur in reality [27]. The faults may cause the states of different nodes to be inconsistent with each other. For example, the collection of marked edges may not be an MST.

Known transformers can transform any MST construction algorithms to be self stabilizing. The transformer of Katz and Perry [28] also assumes a leader whose memory must hold a snapshot of the whole network. The time of the resulting self stabilizing MST algorithm is $O(n)$ and the memory size is $O(|E|n)$. In [2], the first self stabilizing leader election algorithm was proposed in order to remove the assumptions of [28] that a leader and a spanning tree are given. (The algorithm of [4], presented independently, needed an extra assumption that a bound on n was known, and had a higher time complexity). The combination of [2] and [28]) implied a self stabilizing MST in $O(n^2)$ time. Using unbounded space,

the time of self stabilizing leader election was later improved even to $O(D)$ (the actual diameter) [3, 17]. The bounded memory algorithms of [7] or [1, 16], together with [28] and [5], yield a self stabilizing MST algorithm using $O(n|E|\log n)$ bits per node and time $O(D\log n)$ or $O(n)$.

Gupta and Srimani [24] presented an $O(n\log n)$ bits algorithm. Higham and Lyan [25] improved the core memory requirement to $O(\log n)$, however, the time complexity went up again to $\Omega(n|E|)$. An algorithm with a similar time complexity and a similar memory per node was also presented by [10]. This algorithm assumes the existence of a unique leader in the network and exchanges less bits with neighbors than did the algorithm of [25]. The algorithm of [10] also maintains a tree at all times (after reaching an initial tree). The time complexity of the algorithm in [11] is $O(n^2)$ but the memory usage grows to $O(\log^2 n)$. This may be the first paper using labeling schemes for the design of a self stabilizing MST protocol, as well as the first paper implementing the algorithm by Gallager, Humblet, and Spira in a self stabilizing manner without using a general transformer.

2. PRELIMINARIES

Some General definitions: Since we use rather standard definitions, we allow ourselves to save on space by referring the reader to a more complete version [30] or to the model descriptions in the rich literature on these subjects. In particular, we use the rather standard definitions of self stabilization (including its definition of faults) and of an edge weighted graph $G=(V,E)$ to represent a network. (The weights are polynomial in $n=|V|$). Each node has a unique identity $\texttt{ID}(v)$ encoded using $O(\log n)$ bits. Moreover, the network can store an object such as an MST (Minimum Spanning Tree) by having each node store its *component* of the representation. A component includes a collection of pointers to neighbors, and the collection of the components induces a subgraph $H(G)$. Here, $H(G)$ is supposed to be an MST (and each component is one pointer).

The (rather common) ideal time complexity assumes that a node reads all of its neighbors in at most 1 time unit. See e.g. [10, 11]. Our results translate easily to an alternative, stricter, *contention* time complexity, where a node can access only one neighbor in one time unit. The time cost of such a translation is at most a multiplicative factor of $\triangle$, the (unknown) maximum degree of a node in the graph.

As is commonly assumed in the case of self stabilization, we also assume that each node has only some bounded number (here, $O(\log n)$) of memory bits available to be used. Without *some* bound, the adversary could have started the memory to any value, and the definition of the memory complexity would have become tricky. (Note that the upper bound on n that this memory bound implies is far from being a tight one – it can be any polynomial in n; our algorithms do not make use of this upper bound.)

Proof labeling schemes: This generalization of the schemes in [29, 31] is a framework for maintaining a distributed proof that the network satisfies some given predicate Ψ, e.g., that $H(G)$ is an MST. We are given a predicate Ψ and a graph family $\mathcal{F}$ (if Ψ and $\mathcal{F}$ are omitted, Ψ is MST and $\mathcal{F}$ is all connected undirected weighted graphs $\mathcal{F}(n)$). A *proof labeling scheme* includes the following two components.

- A *marker* algorithm $\mathcal{M}$ that generates a label $\mathcal{M}(v)$ for every node v in every graph $G\in\mathcal{F}$.
- A *verifier* distributed algorithm $\mathcal{V}$, initiated at each node of a *labeled* graph $G\in\mathcal{F}$, i.e., a graph whose nodes v have labels $L(v)$. The verifier at each node is initiated separately, and at an arbitrary time, and runs forever. The verifier may raise an alarm at some node v by outputting "no" at v.

Intuitively, if the verifier at v raises an alarm, then it detected a fault. That is, for any graph $G\in\mathcal{F}$,

- If G satisfies the predicate Ψ and if the label at each node v is $\mathcal{M}(v)$ (i.e., the label assigned to v by the marker algorithm $\mathcal{M}$) then no node raises an alarm. In this case, we say that the verifier *accepts* the labels.
- If G does not satisfy the predicate Ψ, then for *any* assignment of labels to the nodes of G, after some finite time t, there exists a node v that raises an alarm. In this case, we say that the verifier *rejects* the labels.

Note that the first property above concerns only the labels *produced by the marker algorithm* $\mathcal{M}$, while the second must hold even if the labels are assigned by some adversary.

We evaluate $(\mathcal{M},\mathcal{V})$ by the following complexity measures.

- The *memory size*: the maximum number of bits stored in the memory of a single node v, taken over all the nodes v in all graphs $G\in\mathcal{F}(n)$ that satisfy the predicate Ψ (and over all the executions); this includes: (1) the bits used for encoding the identity $\texttt{ID}(v)$, (2) the marker memory: bits used for constructing and encoding the labels, and (3) the verifier memory: the bits used during the operation of the verifier.
- The (ideal) detection *time*: the maximum, taken over all the graphs $G\in\mathcal{F}(n)$ that do *not* satisfy the predicate Ψ and over all the labels given to nodes of G by adversaries (and over all the executions), of the time t required for some node to raise an alarm. (The time is counted from the *starting time*, when the verifier has been initiated at all the nodes).
- The *detection distance*: For a faulty node v, this is the (hop) distance to the closest node u raising an alarm within the detection time after the fault occurs. The detection distance of the scheme is the maximum, taken over all the graphs having at most f faults, and over all the faulty nodes v (and over all the executions), of the detection distance of v.
- The (ideal) *construction-time*: the maximum, taken over all the graphs $G\in\mathcal{F}(n)$ that satisfy the predicate Ψ (and over all the executions), of the time required for the marker $\mathcal{M}$ to assign labels to all nodes of G.

In our terms, the definitions of [29, 31] allowed only detection time complexity 1. Because of that, the verifier of [29, 31] at a node v, could only consult the neighbors of v. Whenever we use such a scheme, we refer to it as a 1-proof labeling scheme, to emphasis its running time. Also, in [29, 31], if f faults occurred, then the detection distance was f. Intuitively, a short detection distance and a small detection time may be helpful for the design of local correction and for fault containment algorithms [8, 23].

Generalizing the complexities to a computation: Above, we defined the memory size, detection time and the detection distance complexities of a *verification* algorithm. When

considering a (self stabilizing) computation algorithm, we extend the notion of the memory size to include also the bits needed for encoding the component $c(v)$ at each node. (This was excluded from the definition of memory size for verification because, there, the designer of the verification scheme has no control over the nodes' components.)

The notions of detection time and the detection distance can be carried to the very common class of self stabilizing *computation* algorithms that use fault detection. Examples are algorithms that have silent stabilization [19]. Informally, algorithms in this class first compute an output. After that, all the nodes are required to stay in some *output state* where they (1) output the computation result forever (unless a fault occurs); and (2) check repeatedly until they discover a fault. In such a case, they recompute and enter an output state again. Here, informally, the detection time is the time from a fault (occurring after stabilization) until the detection. (There is a small delicate point, however; if, at this point, a node is not in the output state, this is considered a detection of a fault, since in the stabilized case, all the nodes are supposed to be and remain in an output state.) The detection distance is the distance from a node where a fault occurred to a node that detected a fault. A more formal definition appears in [30].

Trees, hierarchies and candidate functions: From now on, fix a spanning tree $T = (V(G), E(T))$ of a graph $G = (V(G), E(G))$, rooted at some node $r(T)$. Following [21], a *fragment* F is a subtree of T. Given a fragment F, an edge $(v, u) \in E(G)$ whose one endpoint v is in F, while the other endpoint u is not, is called *outgoing* from F. Such an edge of minimum weight is called a *minimum outgoing* edge from F. A fragment containing a single node is called *singleton*.

Below, we define a hierarchy and a candidate function, two notions that play a major role later. Hence, let us first explain them informally. The proof labeling scheme proves that T could have been computed by an algorithm that is similar to that of GHS, the algorithm of [21]. GHS starts when each node is a fragment by itself. Fragments merge over their minimum outgoing edges to form larger fragments. That is, each node belongs to one fragment F_1, then to a larger fragment F_2 that contains F_1, etc. This is repeated until one fragment spans the network. A tree constructed that way is an MST [21]. In GHS, each fragment has a *level*; in the case of v above, F_2's level is higher than that of F_1. The hierarchy $\mathcal{H}$ of fragments we define below follows that structure. Fragment F_1 is a descendant in $\mathcal{H}$ of F_2 if F_2 contains F_1. GHS managed to ensure that each node belongs to at most one fragment at each level, and that the number of levels is $O(\log n)$. Hence, $\mathcal{H}$ has a small depth.

The marker algorithm in the proof labeling scheme presented here performs, in some sense, a reverse operation. If T is an MST, the marker "slices" it back into fragments. Then, the proof labeling scheme gives each node v (1) the (unique) name of each of the fragments F_j that v belongs to, (2) the level of F_j, and (3) the weight of F_j's minimum outgoing edge. This (for $O(\log n)$ fragments) is really too much information to store in one node, so we shall see later how the scheme brings this information to the node without violating the memory size bound. For now, it suffices to know that given this information, the nodes can verify that T could have been constructed by an algorithm similar to GHS, and, hence, T is an MST. Finally, the "candidate" defined below for each fragment F_j is really what is supposed to be F_j's minimum outgoing edge. We say a "candidate", and "supposed to be", to stress the point that this is, really, what the proof labeling scheme is supposed to check.

Formally, we define a *hierarchy* $\mathcal{H}$ for T as a collection of fragments of T satisfying the following: (1) $T \in \mathcal{H}$, and $\{v\} \in \mathcal{H}$ for every $v \in V(G)$, and (2) for any two fragments F and F' in $\mathcal{H}$, if $F \cap F' \neq \emptyset$ then either $F \subseteq F'$ or $F' \subseteq F$. (The collection of fragments is a laminar family).

For a fragment $F \in \mathcal{H}$, let $\mathcal{H}(F)$ denote the collection of fragments in $\mathcal{H}$ *strictly* contained in F. A child of a non-singleton fragment $F \in \mathcal{H}$ is a fragment $F' \in \mathcal{H}(F)$ such that no other fragment $F'' \in \mathcal{H}(F)$ satisfies $F'' \supset F'$. Note that the rooted tree induced by a hierarchy is unique (if the children are unordered). To avoid confusion with tree T, we refer to the tree induced by a hierarchy as a *fragment-tree*. We associate a *level* with each fragment $F \in \mathcal{H}$. It is defined as the height of the node corresponding to F in the fragment-tree induced by $\mathcal{H}$. In particular, the level of a singleton fragment is 1. The level of the fragment T is called the *height* of the hierarchy, and is denoted by ℓ.

Given a hierarchy $\mathcal{H}$ for T, a function $\chi : \mathcal{H} \backslash \{T\} \longrightarrow E(T)$ is called a *candidate function* of $\mathcal{H}$ if it satisfies $E(F) = \bigcup_{F' \in \mathcal{H}(F)} \chi(F')$ for every $F \in \mathcal{H}$, namely, F is precisely the union of candidates of all fragments of $\mathcal{H}$ strictly contained in it. The proof of the following lemma is similar, e.g., to the proof of [21]. See [30].

LEMMA 1. *Let T be a spanning tree of a graph G. If there exists a candidate function χ for a hierarchy $\mathcal{H}$ for T, such that for every $F \in \mathcal{H}$, $\chi(F)$ is a minimum outgoing edge from F, then T is an MST of G.*

3. FAST AND COMPACT PROOF SCHEME

In this section, we describe the labels used by the proof labeling scheme. The distributed implementation of the marker that actually *assigns* these labels to the nodes is deferred to the full paper. We begin with a few simplifying assumptions, called the *data structure assumptions*. The formal description of how to remove these assumptions is deferred to the full paper. (For the implementation and the removal, see also [30]). First, we assume that $H(G) \equiv T$ is a spanning tree of G rooted at some node r. Thus, our goal is to verify that T is in fact, minimal. Second, we assume that each node knows n. Third, we assume the existence of a hierarchy $\mathcal{H}$ for T of height $\ell \leqslant 1 + \lceil \log n \rceil$ and a candidate function χ for $\mathcal{H}$. which are represented distributively in the nodes of T as follows. Each node v is equipped with a data-structure marking exactly those levels $1 \leqslant j \leqslant \ell$ for which v belongs to a fragment $F_j(v)$ of level j in $\mathcal{H}$. For each such a level j, the data-structure at v also indicates (1) whether v is the root of $F_j(v)$ (in particular, whether v is the root r of T), and (2) whether v is an endpoint of the (unique) candidate edge of $F_j(v)$, and if so, which of the edges adjacent to v is the candidate edge. Furthermore, given the data-structures of two nodes u and v which are neighbors in G, one can find out whether they are neighbors in T as well, and if so: (1) whether u is v's child in T, and (2) for each $1 \leqslant j \leqslant \ell$, whether u belongs to $F_j(v)$.

Informally, to remove the third assumption, given an MST T, the marker $\mathcal{M}$ constructs a hierarchy, a candidate function, and data-structures as required by the above assumptions. We couple these constructions with a 1-proof labeling scheme that verifies that these are as required, using ideas similar

to the ones described in [31]. The first two assumptions can be even more directly removed using simple known 1-proof labeling schemes from [31]. The resulted data-structure, together with the labels needed to verify it, can be constructed in $O(n)$ time using memory of at most $O(\log n)$ bits at each node. Hence, the removal of the assumptions does not violate the memory size nor the time requirements of the final proof labeling scheme.

Intuitively, the scheme below allows the nodes of each fragment F_j to detect whether the lowest weight edge connecting F_j to the rest of the graph is included in T. By Lemma 1, if this holds for every fragment, then T is an MST.

The pieces each node uses for verification (see the intuition paragraph, in the introduction): A crucial point in the scheme is letting each node v know, for each edge $(v,u) \in E$ and for each level j, whether u and v share the same level j fragment. (In the special case where u is also a neighbor of v in T, this information can be extracted by v using u's data-structure.) We assign each fragment a unique identifier, and v compares the identifier of its own level j fragment to the identifier of u's level j fragment.

Consider the number of bits required to represent the identifiers of all the fragments that a node v participates in. There exists a method to assign unique identifiers such that this total number is only $O(\log n)$ [20, 32]. Unfortunately, we did not manage to use that method here. Indeed, our marker can assign identifiers according to that method. However, we could not find a low space and short time method for the verifier to verify that given identifiers of the fragments Where indeed assigned that way (in particular, we could not verify that the given identifiers are indeed unique).

Hence, we assign identifiers according to another method that appears more memory wasteful: the identifier of a fragment F is $\mathtt{ID}(F) = (\mathtt{ID}(r(F)), j_F)$, where $\mathtt{ID}(r(F))$ is the unique identity of the root $r(F)$ of F, and j_F is F's level. We also need each node v to know the weight $\omega(F)$ of the minimum outgoing edge of each fragments containing v. To summarize, the *piece of information* $\mathtt{I}(F)$ required in each node v per fragment F containing v is $\mathtt{ID}(F) \circ \omega(F)$. Thus, $\mathtt{I}(F)$ can be encoded using $O(\log n)$ bits. Since a node may participate in $\ell = O(\log n)$ fragments, the total number of bits used for storing all the $\mathtt{I}(F)$ for all fragments F containing v would thus be $\Theta(\log^2 n)$. Had no additional steps been taken, this would have violated the $O(\log n)$ memory constraint. To save on memory, our scheme distributes the above information, while guaranteeing that each node u holds $\mathtt{I}(F)$ for at most constant number of fragments F.

To allow some node v to check whether its neighbor u belongs to v's level j fragment $F_j(v)$ for some level j, the verifier at v needs first to reconstruct $\mathtt{I}(F_j(v))$. Intuitively, we had to distribute the information, so that $\mathtt{I}(F)$ is placed "not too far" from every node in F. To compare $\mathtt{I}(F_j(v))$ with a neighbor u, Node v must also obtain $\mathtt{I}(F_j(u))$ from u. This requires some mechanism to synchronize the reconstructions in neighboring nodes. Furthermore, the verifier must be able to overcome difficulties resulting from faults, that can corrupt the information stored, as well as the reconstruction and the synchronization mechanisms.

The above distribution of the $\mathtt{I}$'s is described in the next subsection. The distributed algorithm for the "fragment by fragment" reconstruction (and synchronization) is described in Subsection 3.2. The required verifications for validating the $\mathtt{I}$'s and comparing the information of neighboring nodes are described in Subsection 3.3.

3.1 Distributing the information

At a very high level description, each node v stores permanently $\mathtt{I}(F)$ for a constant number of fragments F. Using that, $\mathtt{I}(F)$ is "rotated" so that each node in F "sees" $\mathtt{I}(F)$ in $O(\log n)$ time. We term the mechanism that performs this rotation a *train*. A first idea would have been to have a separate train for each fragment F that would "carry" the piece $\mathtt{I}(F)$ and would allow all nodes in F to see it. However, we did not manage to do that efficiently in terms of time and of space. That is, one train passing a node could delay the others. Since neighboring nodes may share only a subset of their fragments, it is not clear how to pipeline the trains. Hence, those delays could accumulate. Moreover, as detailed below, each train utilizes some (often more than constant) memory per node. Hence, a train per fragment would have prevented us from obtaining an $O(\log n)$ memory solution.

A more refined idea would have been to partition the tree into connected parts, such that each part P intersects $O(|P|)$ fragments. Had we managed such a partition, we could have allocated the $O(|P|)$ pieces (of these $O(|P|)$ fragments), so that each node of P would have been assigned only a constant number of such pieces, costing $O(\log n)$ bits per node. Moreover, just one train per part P could have sufficed to rotate those pieces among the nodes of P. Each node in P would have seen all the pieces $\mathtt{I}(F)$ for fragments F containing it in $O(|P|)$ time. Hence, this would have been time efficient too, had P been small.

Unfortunately, we did not manage to construct the above partition. However, we managed to obtain a similar construction: We constructed *two* partitions of T, called `Top` and `Bottom`. We also partitioned the fragments into two kinds: "top" and "bottom" fragments. Now, each part P of Partition `Top` intersect with $O(|P|)$ "top" fragments. Each part P of Partition `Bottom` intersects with $O(|P|)$ "bottom" fragments. Hence, it is enough for each node to participate in two trains only, one for each partition.

The two partitions: The marker performs the partitioning in the preliminary stage, while assigning the labels using $O(\log n)$ memory and $O(n)$ time. This is deferred to the full paper (see also [30]). We now describe the desired partitions `Top` and `Bottom`, and the classification of fragments to "top" and "bottom" ones. First, define the "top" fragments to be precisely those fragments which contain at least $\log n$ nodes. Observe that the "top" fragments correspond to a subtree $T_{\mathtt{Top}}$ of the hierarchy tree $\mathcal{H}$. Let us describe Partition `Top`. A leaf fragment in subtree $T_{\mathtt{Top}}$ is colored red. A fragment not in $T_{\mathtt{Top}}$ which is a sibling in $\mathcal{H}_{\mathcal{M}}$ of a fragment in $T_{\mathtt{Top}}$ is colored blue. We say that a red fragment F_{red} and a blue fragment F_{blue} are *relatives* if F_{red} is a descendant of a sibling of F_{blue} in $\mathcal{H}$. Observe that the collection of red and blue fragments forms a partition $\mathcal{P}'$ of the nodes of T. Also, each fragment F_{large} that strictly contains blue fragments is composed of at least one red fragment F_{red} as well as one or more blue ones, and does not contain any additional nodes. Since F_{large} is connected, it is possible to partition F_{large} to connected components such that each component P'' contains precisely one red fragment and possibly several blue ones, and no additional *nodes*. (Of course, P'' may contain also the *edges* connecting the fragments P''contains.)

The collection of such connected components P'' forms

a partition $\mathcal{P}''$ which is a coarsening of partition $\mathcal{P}'$. Since each part contains a red fragment, we get that each part is of size at least $\log n$. (This is the reason why we attach blue fragments to one of their relative red fragments, that is, a blue fragment cannot be a part by itself since it may not contain enough nodes to store all the pieces of information regarding all of its many ancestors.) Observe also that if $F_1, F_2, \cdots F_t$ are the ancestors of a blue fragment F in $\mathcal{H}$ then those $F_1, F_2, \cdots F_t$ fragments are also ancestors in $\mathcal{H}$ of each red fragment which is a relative of F. Hence, the "top" fragments intersecting a part P are precisely those fragments which are the ancestors in $\mathcal{H}$ of the red fragment in P. In particular, we get that each part $P \in \mathcal{P}''$ intersects at most one level j "top" fragment, for every j. (This is the reason for not putting more than one red fragment in a part.)

We would like to pass a train in each part P of $\mathcal{P}''$. Unfortunately, the diameter of P may be too large. In such a case, we partition P further to *neighborhoods*, such that each neighborhood is a subtree of T of size at least $\log n$ and of diameter $O(\log n)$, establishing the following lemma.

LEMMA 2. *There exists a partition* `Top` *of* $V(T)$ *such that* $\forall P \in$ `Top`, $|P| \geqslant \log n$ *and* $D(P) = O(\log n)$. *Moreover,* P *intersects at most one level* j *"top" fragment, for every* j *(in particular, it intersects at most* $\log n$ *"top" fragments).*

The "bottom" fragments are precisely those with less than $\log n$ nodes. The parts of the second partition `Bottom` are the following: (1) the blue fragments, and (2) the children in $\mathcal{H}_{\mathcal{M}}$ of the leaves of $T_{\texttt{Top}}$. Observe that each part of `Bottom` is a "bottom" fragment. Thus, the size, and hence the diameter, of each part P of `Bottom` is less than $\log n$. Observe also that P contains all of P's descendant fragments in $\mathcal{H}$ (recall, P is a fragment), and does not intersect other "bottom" fragments. Hence, P intersects at most $2|P| < 2\log n$ "bottom" fragments.

Consider either Partition `Top` or `Bottom`. The parts of the partition and their corresponding roots are represented using 1 bit at each node v. The bit indicates whether v is the root $r(P)$ of a part P (the highest node of P) or not. Thus, by consulting the data-structure of a tree neighbor u, Node v can detect whether u and v belong to the same part.

A delicate and interesting point is that the verifier does not need to verify directly that the partitions were constructed as explained here. This is explained in Section 3.3.

Initializing the trains: Given Partition `Top` (respectively, `Bottom`), we construct a train for each of its parts P. Recall that P is a subtree of T rooted at $r(P)$. Let $\{F_i\}_{i\in[1,k]}$ be the "top" (resp., "bottom") fragments intersecting P, for some integer k. Recall that $k \leqslant \min\{2|P|, 2\log n\}$. Assume w.l.o.g., that the indices are such that the level of F_i is at least the level of F_{i-1}, for each $1 < i \leqslant k$. Let $\texttt{I}(P) = \texttt{I}(F_1) \circ \texttt{I}(F_2) \circ, \cdots, \circ \texttt{I}(F_k)$. We break $\texttt{I}(P)$ into $|P|$ pairs of pieces. Specifically, for $1 \leqslant i \leqslant \lceil k/2 \rceil$, the i'th pair, termed $\texttt{PcP}(i)$, contains $\texttt{I}(F_{2i-1}) \circ \texttt{I}(F_{2i})$ (for odd k, $\texttt{PcP}(\lceil k/2 \rceil) = \texttt{I}(F_{2i-1})$). The subtree P stores distributively $\texttt{I}(P)$, as follows. Consider a DFS traversal over P that starts at $r(P)$ and let $\texttt{dfs}(i)$ denote the the i'th node visited in this traversal. For each i, $1 \leqslant i \leqslant \lceil k/2 \rceil$, $\texttt{dfs}(i)$ stores permanently the i'th pair of $\texttt{I}(P)$, i.e., $\texttt{PcP}(i)$.

3.2 Viewing distributed informaion

Recall that $\texttt{I}(F_j(v))$ should reside permanently in some node of a part P to which v belongs. To allow v to compare $\texttt{I}(F_j(v))$ to $\texttt{I}(F_j(u))$ for a neighbor u, both these pieces must somehow be "brought" to v. The process handling this task contains several components. The first is called a "train" and is responsible for moving the pieces' pairs $\texttt{PcP}(i)$ through P's nodes, such that each node does not hold more than $O(\log n)$ bits at a time, and such that in time $O(\log n)$, each node in P "sees" all pieces, and in their correct order. Unfortunately, this is not enough, since $\texttt{I}(F_j(v))$ may arrive at v at a different time than $\texttt{I}(F_j(u))$ arrives at u. Further complications arise from the fact that the neighbors of a node v may belong to different parts, so different trains pass there. Note that v may have many neighbors, and we would not want to synchronize so many trains. Moreover, had we delayed the train at v for synchronization, the delay would have accumulated, or even would have caused deadlocks. Hence, we do not delay these trains. Instead, v repeatedly samples a piece from its train, and synchronizes the comparison of this piece with pieces sampled by its neighbors, while both trains advance without waiting. Perhaps not surprisingly, this synchronization turns out to be easier in synchronous networks, than in asynchronous ones.

This process is presented below assuming that no fault occurs. The detection of faults is described later.

The trains: For simplicity, we split the task of a train into two subtasks, each performed repeatedly- the first, *convergecast*, moves the pieces one at a time *pipelined* from their permanent locations to $r(P)$ according to the DFS order. (Recall, $\texttt{dfs}(i)$ stores permanently the i'th piece of $\texttt{I}(P)$.) Call each consecutive delivery of the k pieces pair $\texttt{PcP}(1), \texttt{PcP}(2), \cdots, \texttt{PcP}(k)$ to $r(P)$ a *cycle*. Since we are concerned with at most $k \leqslant 2\log n$ pieces's pairs, each cycle can be performed in $O(\log n)$ time. The second subtask, *broadcast*, broadcasts each piece from $r(P)$ to all other nodes in P (pipelined). This subtask can be performed in $D(P) = O(\log n)$ time, where $D(P)$ is the diameter of P. These two subtasks (and their stabilization) are rather straightforward, hence their description is omitted.

Consider now the case that a piece containing $\texttt{I}(F)$ carried by the broadcast wave arrives at some node v. Abusing notations, we refer to this event by saying that Fragment F *arrives* at v. Recall that v does not have enough memory to remember the identifiers of all the fragments containing it. Thus, a mechanism for letting v know whether the arriving fragment F contains v must be employed. Note that the level j of F can be extracted from $\texttt{I}(F)$, and recall that it is assumed that v knows whether it is contained in some level j fragment. Obviously, if v is not contained in a level j fragment then $v \notin F$. If $F_j(v)$ does exist, we now explain how to let v know whether $F = F_j(v)$.

Consider first a train in a part $P \in$ `Top`. Here, P intersects at most one level j "top" fragment, for each level j (see Lemma 2). Thus, this train carries at most one level j fragment F_j. Hence, $F_j = F_j(v)$ iff $F_j(v)$ exists.

Now consider a train in a part $P \in$ `Bottom`. Unfortunately, in this case, Part P may intersect several "bottom" fragments of the same level. To allow a node v to detect whether a fragment F_j arriving at v corresponds to Fragment $F_j(v)$, we refine the above mentioned train broadcast mechanism as follows. During the broadcast wave, we attach a flag to each $\texttt{I}(F)$, which can be either "on" or "off", where initially, the flag is "off". Recall that $\texttt{I}(F)$ contains the identity $\texttt{ID}(r(F))$ of the root $r(F)$ of F. When the broadcast wave reaches this root $r(F)$ (or, when it starts in $r(F)$ in the case that

$r(F) = r(P)$), it changes the flag to "on". In contrast, before transmitting the broadcast wave from a leaf u of F to u's children in T (that do not belong to F), Node u sets the flag to "off". That way, a fragment F arriving at a node v contains v if and only if the corresponding flag is set to "on". (Recall that the data structure lets each node know whether it is a leaf of a level j fragment). Hence, node v can detect whether $F = F_j(v)$.

To avoid delaying the train beyond a constant time, each node multiplexes the two trains passing via it. That is, it passes one piece of one train, then one piece of the other.

Sampling and synchronizing: Node u maintains two variables: $\texttt{Show}(u)$ and $\texttt{Ask}(u)$, each for holding one piece $\texttt{I}(F)$. In $\texttt{Ask}(u)$, node u keeps $\texttt{I}((F_j(u))$ for some j, until u compares the piece $\texttt{I}(F_j(u))$ to the piece $\texttt{I}(F_j(v))$, for each of its neighbors v. Let $\mathcal{E}(u, v, j)$ denote the event that Node u holds $\texttt{I}(F_j(u))$ in $\texttt{Ask}(u)$ and sees $\texttt{I}(F_j(v))$ in $\texttt{Show}(v)$. (For simplicity of presentation, we consider here the case that both u and v *do* belong to some fragments of level j; otherwise, storing and comparing the information for a non-existing fragments is trivial.) For any point in time t, let $C(t)$ denote the minimal time interval $C(t) = [t, x(t)]$ in which every event of the type $\mathcal{E}(u, v, j)$ occurred. For the scheme to function, it is crucial that $C(t)$ exists for any time t. Moreover, to have a fast scheme, we must ensure that $\max_t |C(t)|$ is small.

Recall that the train brings the pieces $\texttt{I}(F)$ in a cyclic order. When u is done comparing $\texttt{I}((F_j(u))$ (to $\texttt{I}(F_j(v))$ for each of its neighbors v), node u waits until it receives (by the train) the first piece $\texttt{I}(F)$ following $\texttt{I}((F_j(u))$ in the cyclic order, such that F contains u (recall that u can identify this F). Let us denote the level of this next fragment F by j', i.e., $F = F_{j'}(u)$. Node u then removes $\texttt{I}(F_j(u))$ from $\texttt{Ask}(u)$ and stores $\texttt{I}(F_{j'}(u))$ there instead, and so forth.

Let us explain the comparing mechanism. Each node u also stores some piece $\texttt{I}(F_i(u))$ at $\texttt{Show}(u)$ to be seen by its neighbors. Fix a node v and one of its neighbors u. In a synchronous network, Node v sees $\texttt{Show}(u)$ in every pulse. Then, u waits until u receives (by the train) $\texttt{I}(F_{i'}(u))$, then u stores it in $\texttt{Show}(u)$, etc. Hence, if v waits some $O(\log n)$ time (while $\texttt{I}(F_j(v))$ is in $\texttt{Ask}(v)$), Node v sees $\texttt{I}(F_j(u))$ in $\texttt{Show}(u)$ and event $\mathcal{E}(v, u, j)$ occurs. The time for $\log n + 1$ such events to occur (one event per level j) is $O(\log^2 n)$.

The above result for synchronous networks is already enough to ensure that our asynchronous self stabilizing MST construction algorithm has $O(n)$ stabilization time. Intuitively, this is because the latter uses a self stabilizing synchronizer, see Section 4. However, since the synchronizer itself has stabilization time $O(n)$, the result is not enough to ensure a short detection time complexity of the self stabilizing MST construction. To ensure that, we also establish an upper bound of $O(\triangle \log^2 n)$ for the detection time of a verification algorithm in asynchronous networks. Because of lack of space, this is deferred to the full paper.

LEMMA 3. *If (1) two partitions are indeed represented, such that each part of each partition is of diameter $O(\log n)$, and the number of pieces in a part is $O(\log n)$, and (2) the trains have self stabilized, then the following holds.*

- *In a synchronous network,* $\max_t |C(t)| = O(\log^2 n)$.
- *In an asynchronous network,* $\max_t |C(t)| = O(\Delta \log^2 n)$.

3.3 Local verifications

In this section, we describe the measures taken in order to make the verifier self stabilizing. That is, the train process and also, the pieces of information carried by the train may be corrupted by an adversary. To stress this point and avoid confusion, a piece of information of the form $z \circ j \circ \omega$, carried by a train, is termed the *claimed* information $\hat{\texttt{I}}(F)$ of a fragment F whose root $\texttt{ID}$ is z, whose level is j, and whose minimum outgoing edge is ω. Note that such a fragment F may not even exist, if the information is corrupted. Conversely, the adversary may also erase some (or even all) of such pieces corresponding to existing fragments. Finally, even correct pieces that correspond to existing fragments may not arrive at a node in the case that the adversary corrupted the partitions or the train mechanism. Below we explain how does the verifier detect such undesirable phenomenas, if they occur. Note that for a verifying, the ability to detect assuming any initial configuration means that the verifier is self stabilizing, since the sole purpose of the verifier is to detect. We show, in this section, that if an MST is not represented in the network, this is detected. (Since the detection time (the stabilization time) is sublinear, we still consider this detection as local, though some of the locality was traded for improving the memory size (when compared to the results of [31, 29]).

Verifying that *some* two partitions exist is easy, given the data structure assumptions (see the top of Section 3). It is sufficient to (1) let each node verify that its label contains the two bits corresponding to the two partitions; and (2) to have the root $r(T)$ of the tree verify that the value of each of its own two bits is 1. (Observe that if these two conditions hold then (1) $r(T)$ is a root of one part in each of the two partitions; and (2) for a node $v \neq r(T)$, if one of these two bits in v is zero, then v belongs to the same part in the corresponding partition as its parent.) Note that this module of the algorithm self stabilizes in constant time.

It is difficult to verify that the given partitions are as described in Section 3.1, rather than two arbitrary partitions generated by an adversary. Fortunately, this verification turns out to be unnecessary. First, after we verify that *some* partitions are indeed represented, it is a known art to self stabilize the train process, see, e.g. [12, 13]. After the trains stabilize, what we really want to ensure is (a) that the set of pieces stored in a part (and delivered by the train) includes all the (possibly corrupted) pieces of the form $\texttt{I}(F_j(v))$, for every v in the part and for every j such that v belongs to a level j fragment, and (b) that each node obtains all the the pieces it needs quickly, i.e., in $O(\log n)$ time.

Addressing (a) above, we shall show that the verifier at each node rejects if it does not obtain all the required pieces eventually, whether the partitions are correct or not. Informally, this is done as follows. Recall (the data structure assumptions) that each node v knows the set $J(v)$ of levels j for which there exists a fragment of level j containing it, namely, $F_j(v)$. Using a delimiter (stored at v), we partition $J(v)$ to $J_{\texttt{Top}}(v)$ and $J_{\texttt{Bottom}}(v)$; where $J_{\texttt{Top}}(v)$ (respectively, $J_{\texttt{Bottom}}(v)$) is the set of levels $j \in J(v)$ such that $F_j(v)$ is "top" (resp., "bottom"). Node v "expects" to receive the claimed information $\hat{\texttt{I}}(F_j(v))$ for $j \in J_{\texttt{Top}}(v)$ (respectively, $j \in J_{\texttt{Bottom}}(v)$) from the train of the part in $\texttt{Top}$ (respectively, $\texttt{Bottom}$) it belongs to. Let us now consider the part $P_{\texttt{Top}} \in \texttt{Top}$ containing v. In correct instances, by the way the train operate, it follows that the levels of fragments arriving at v should

arrive in a strictly *increasing order* and in a *cyclical* manner, that is, $j_1 < j_2 < j_3 < \cdots < j_a, j_1 < j_2 < \cdots j_a, j_1 \cdots$ (observe that $j_a = \ell$). Consider the case that the verifier at v receives two consecutive pieces $z_1 \circ j_1 \circ \omega_1$ and $z_2 \circ j_2 \circ \omega_2$ such that $j_2 \leqslant j_1$. The verifier at v then "assumes" that the event S of the arrival of the second piece $z_2 \circ j_2 \circ \omega_2$ starts a new cycle of the train. Let the set of pieces arriving at v between two consecutive such S events be named a *cycle set*. To be "accepted" by the verifier, the set of levels of the fragments arriving at v in each cycle set must contain $J_{\texttt{Top}}(v)$. It is trivial to verify this in two cycles after the faults cease, given the fact that v knows $J_{\texttt{Top}}(v)$ (by the data structure assumptions) and the fact that the levels in each cycle set arrive in strictly increasing order. (The discussion above is based implicitly on the assumption that each node receives pieces infinitely often; this is guaranteed by the correctness of the train mechanism, assuming that at least one piece is indeed stored permanently in $P_{\texttt{Top}}$; Verifying this assumption is done easily by the root $r(P_{\texttt{Top}})$ of $P_{\texttt{Top}}$, simply by verifying that $r(P_{\texttt{Top}})$ itself does contain a piece.)

Verifying the reception of all the pieces in a part in `Bottom` is handled very similarly, and is thus omitted. This completes the informal description of how to guarantee (a) above.

We still need to show that the partitions are such that (b) above is also accomplished. It is a known art to show that the sub- tasks of the train stabilize in Given the $O(D(|P| + |\texttt{PcP}(P)|))$ where $|\texttt{PcP}(P)|$ is the number of pieces stored in the nodes of P, see [13, 12], as well as some more details in [30]. Hence, it is sufficient to verify that each part P of each of the two claimed partitions has diameter $D(P) = O(\log n)$, and that the number of pieces stored in each part is $O(\log n)$. Verifying this is done easily using a 1-proof labeling scheme that uses $O(\log n)$ bits, using the methods of [31] (see, also [30]). The stabilization time of (b) above is, thus, constant.

Hence, we can sum up the above discussion as follows: if the verifier accepts then each node v receives $\hat{\texttt{I}}(F_j(v))$, for every level $j \in J(v)$ (in the time stated in Lemma 3), and conversely, if a node does not receive $\hat{\texttt{I}}(F_j(v))$ (in the time stated in Lemma 3) then the verifier has rejects.

Let $p(v)$ denote the parent of v in T. In event $\mathcal{E}(v, p(v), j)$, Node v compares $\hat{\texttt{I}}(F_j(v))$ with $\hat{\texttt{I}}(F_j(p(v)))$. Consider the case that such a comparison finds these pieces are equal for every such pair of node $v, p(v)$ in a fragment. Intuitively, this means the nodes of the fragments agree (by transitivity) on the fragment's $\hat{\texttt{I}}$. Hence, in the full paper we show that the following is verified in in $O(\log n)$ time:

- The claimed identifiers of the fragments are compatible with the given hierarchy $\mathcal{H}$. (I.e., for every $F \in \mathcal{H}$, $\hat{\texttt{I}}(F)$ is of the form $z \circ j \circ \omega$, and we verify that the identifier of F's root is z.) In particular, this guarantees that the identifiers of fragments are indeed unique.
- For every $F \in \mathcal{H}$, all the nodes in F agree on the *claimed* weight of the minimum outgoing edge of Fragment F, denoted $\hat{\omega}(F)$.

So far, we have shown that each node does receive the necessary information needed for the verifier. Now, finally, we show how to use this information to detect whether this is an MST. Basically, we verify that $\hat{\omega}(F)$ is indeed the minimum outgoing edge $\omega(F)$ of F and that this minimum is indeed the candidate edge of F, for every $F \in \mathcal{H}$. Consider a time when $\mathcal{E}(v, u, j)$ occurs. Node v proceeds and outputs "no" if any of the checks below is not valid.

- **C1:** If v is the endpoint of the candidate edge $e = (v, u)$ of $F_j(v)$ then v checks that u does not belong to $F_j(v)$, i.e., that $\hat{\texttt{ID}}(F_j(v)) \neq \hat{\texttt{ID}}(F_j(u))$, and that $\hat{\omega}(F_j(v)) = \omega(e)$ (recall, the data structure assumptions ensure that v knows whether it is an endpoint, and if so, which of its edges is the candidate);
- **C2:** If $\hat{\texttt{ID}}(F_j(v)) \neq \hat{\texttt{ID}}(F_j(u))$ then v verifies that $\hat{\texttt{ID}}(F_j(v)) \neq \hat{\texttt{ID}}(F_j(u)) \Rightarrow \hat{\omega}(F_j(v)) \leqslant \omega(v, u)$.

The following now follows from C1, C2 and Lemma 1.

LEMMA 4.
- *If by some time t, the events $\mathcal{E}(v, u, j)$ occurred for each node v and each neighbor u of v in G and for each level j, and the verifier did not reject, then T is an MST of G.*
- *If T is not an MST, then in the time stated in Lemma 3 after the fault cease, the verifier rejects.*

The following theorem now follows (the detection distance and more detailed analysis are deferred to the full; additional details also appears in [30]).

THEOREM 1. *The scheme described in this section is a correct proof labeling scheme for MST. Its memory complexity is $O(\log n)$ bits. Its detection time complexity is $O(\log^2 n)$ in synchronous networks and $O(\triangle \log^2 n)$ in asynchronous ones. Its detection distance is $O(f \log n)$ if f faults occurred. Its construction time is $O(n)$.*

4. THE SELF STABILIZING ALGORITHM

We use a transformer that inputs a non- stabilizing algorithm and outputs a self stabilizing one. For simplicity, we first explain how to use the transformer proposed in the seminal paper of [9] (which utilizes the transformer of its companion paper [8]) as a black box. This already yields an $O(n)$ time and $O(\log n)$ memory per node. Later, we refine that transformer somewhat to add the property that the verification time is of $O(\log^2 n)$ time in a synchronous network, or $O(\min\{\triangle \log^2 n, n\})$ in an asynchronous one. We then also establish the property that if f faults occur, then each fault is detected within its $O(f \log n)$ neighborhood.

The Resynchronizer of [9] gets as an input a non-stabilizing synchronous input/output algorithm Π whose running time and memory size are some T_Π and S_Π, respectively, and yields a self stabilizing version whose memory size is $O(S_\Pi + \log n)$ and whose time complexity is $O(T_\Pi + \hat{D})$ where $\hat{D}$ is an *upper bound* on the actual diameter D of the network. (An input/output algorithm is one whose correctness requirement can be specified as a relation between its input and its output). To have the Resynchronizer yield our desired result, we first need to come up with such a bound $\hat{D}$. Second, the result of the Resynchronizer is a synchronous algorithm, while we want an asynchronous one. Third, we need such a synchronous MST construction algorithm Π whose memory size is $O(\log n)$ bits per node and whose time complexity is $O(n)$. Let us describe briefly how we bridge these gaps. The detailed description is deferred to the full paper.

We use known self stabilizing protocols [1, 16] to compute D, the diameter of the network, in time $O(n)$, using $O(\log n)$ bits. We use this computed D as the desired $\hat{D}$. Note that at the time that [9] was written, the only algorithm

for computing a good bound (of n) on the diameter with a bounded memory had time complexity $\Theta(n^2)$ [2].

To bridge the second gap, of converting the resulting self stabilizing algorithm for an *asynchronous* network, we use a *self stabilizing synchronizer* that transforms algorithms designed for synchronous networks to function correctly in asynchronous ones. Such a synchronizer was not known at the time of [9], but several are available now. The synchronizer of [6, 7] was first described as if it needs unbounded memory. However, as is stated in [7], this synchronizer is meant to be coupled with a reset protocol to bound the memory. To have a memory size of $O(\log n)$ and time $O(n)$, it is sufficient to use a reset protocol with these complexities. We use the reset protocols of [8]. Similarly, this reset protocol is meant to be coupled with a self stabilizing spanning tree algorithm. The complexities of resulting protocol are dominated by those of the spanning tree construction. We plug in some spanning tree algorithm with the desired properties (such as [1, 16]) whose memory size and time complexities are the desired $O(\log n)$ and $O(n)$ in asynchronous networks, respectively. (It is easy to improve the time to $O(D)$ in synchronous networks). We, thus, obtain the desired reset protocol, and, hence, the desired synchronizer protocol. (An alternative synchronizer can be based on the one of [14], again, coupled with some additional known components, such as a module to compute n). Let us sum up the treatment of the first two gaps: thanks to some new modules developed after [9], one can now use the following version of the main result of [9].

THEOREM 2. **Enhanced Awerbuch-Varghese Theorem, (EAV):** *Assume we are given a distributed algorithm* Π *to compute an input/output relation* R*. Whether* Π *is synchronous or asynchronous, let* T_Π *and* S_Π *denote* Π*'s time complexity and memory size, respectively, when executed in synchronous networks. The enhanced Resynchronizer compiler produces an asynchronous (respectively, synchronous) self stabilizing algorithm whose memory size is* $O(S_\Pi + \log n)$ *and whose time complexity is* $O(T_\Pi + n)$ *(resp.,* $O(T_\Pi + D)$*).*

The EAV theorem differs from the result in [9] by (1) addressing also asynchronous algorithms, and (2) basing the time complexity on the actual values of n and D of the network rather than on an a-priori bound $\hat{D}$ that may be arbitrarily larger than D or n. Plugging in the algorithm of Awerbuch [5] as Π, the EAV theorem already yields a self stabilizing MST construction with $O(n)$ time and $O(\log^2 n)$ bits per node. This is faster than the best known result presented recently [11], with the same memory size complexity. Still, below, we improve that further to $O(n)$ time and $O(\log n)$ bits. This is obtained by plugging a different Π, with a better complexity, in the EAV theorem. It turns out that it is rather easy to implement the algorithm of [21] in a shared memory model such that it will use $O(\log n)$ bits per node, without increasing its time complexity T_{GHS}. However, T_{GHS} is somewhat too high, i.e., $\Theta(n \log n)$. Fortunately, it is easy to improve that time complexity to $O(n)$ provided that we implement that algorithm as a *synchronous algorithm*. (Recall that a synchronous algorithm suffices as an input for the Resynchronizer.) This $O(n)$ time implementation uses a very simplified version of the ideas of Awerbuch in [5]. (Awerbuch needed a more complex adaptation of the algorithms of [21] because he needed the algorithm to perform well in asynchronous networks). The description of Π is deferred to the full paper.

4.1 Obtaining fast verification

The Resynchronizer compiler performs iterations forever. Essentially, the first iteration is used to compute the result of Π, by executing Π (plus some additional staff needed for the self stabilization). Each of the later iterations is used to check that the above result is correct. For that, the Resynchronizer executes a checker. If the result is not correct, then the checker in at least one node "raises an alarm". This, in effect, signals that Resynchronizer to drop back to the first iteration. Let us term such a node a *detecting node*. Our refinement just replaces the checker, interfacing with the original Resynchronizer by supplying such a detection node.

We should mention that the original design in [9] is already modular in allowing such a replacement of a checker. In fact, two alternative such checkers are studied in [9]. The first kind of a checker is Π itself. That is, if Π is deterministic, then, if executed again, it must compute the same results again (this is adjusted later in [9] to accommodate randomized protocols). This checker functions by comparing the result computed by Π in each "non-first" iteration to the result it has computed before. If they differ, then a fault is detected. The second kind of a checker is a local checker of the kinds studied in [2, 8] or even one that can be derived from local proofs [31, 29]. That is, a checker whose time complexity is exactly 1. When using this kind of a checker, the Resynchronizer uses one iteration to execute Π, then the Resychronizer executes the checker repeatedly until a fault is detected. The door was left open in [9] for additional checkers. It was in this context that they posed the open problem whether MST has a checker which is faster than MST computation, and still uses small memory. (This paper answers the open problem in the affirmative).

We use a self stabilizing verifier (of a proof labeling scheme) as a checker. That is, if a fault occurs, then the checker detects it, at least in one node, regardless of the initial configuration. Such nodes where the fault is detected serve as the detecting nodes used above by the Resychrnonizer. The following theorem differs from the EAV theorem by stating that the final protocol (resulting from the transformation), also enjoys the good properties of the self stabilizing verifier. I.e., if the self stabilizing verifier has a good detection time and good detection distance, then, the detection time and distance of the resulting protocols are good too.

THEOREM 3. *Suppose we are given the following:*

- *A distributed algorithm* Π *to compute an input/output relation* R*. Whether* Π *is synchronous or asynchronous, let* T_Π *and* S_Π *denote* Π*'s time complexity and memory size, when executed in* synchronous *networks.*
- *An asynchronous (respectively, synchronous) proof labeling scheme* Π' *for verifying* R *with memory size* $S_{\Pi'}$*, whose verifier self stabilizes with verification time and detection distance* $t_{\Pi'}$ *and* $d_{\Pi'}$*, and whose construction time (of the marker) is* $T_{\Pi'}$*.*

Then, the enhanced Resynchronizer produces an asynchronous (resp., synchronous) self stabilizing algorithm whose memory and time complexities are $O(S_\Pi + S_{\Pi'} + \log n)$ *and* $O(T_\Pi + T_{\Pi'} + t_{\Pi'} + n)$ *(resp.,* $O(T_\Pi + T_{\Pi'} + t_{\Pi'} + D)$*), and whose verification time and detection distance are* $t_{\Pi'}$ *and* $d_{\Pi'}$*.*

To obtain the desired proof labeling scheme Π', we implemented two tasks. The first, is constructing a non-stabilizing

distributed marker. The description is deferred to the full paper (See [30]). Note that the marker algorithm does not have to be self stabilizing, since it is transformed to be self stabilizing using the Resynchronizer compiler. (That is, the input of the Resynchronizer includes a combination of the marker and of the MST construction algorithm). Now, two possible checkers are the verifiers of the schemes of [29, 31]. Since their detection time is 1, they stabilize trivially. (Their distributed markers are simplified versions of the marker of the proof labeling scheme given of the current paper). Plugging either one of them (together with the above $O(n)$ time Π) yields the following.

COROLLARY 1. *There exists a self stabilizing MST algorithm with $O(\log^2 n)$ memory size and $O(n)$ time. Moreover, its detection time is* 1*and its detection distance is $f+1$.*

Finally, by plugging the self stabilizing verifier described in Section 3 in the Resychronizer, we obtain The following.

COROLLARY 2. *There exists a self stabilizing MST algorithm that uses optimal $O(\log n)$ memory size and $O(n)$ time. Moreover, its detection time complexity is $O(\log^2 n)$ in synchronous networks and $O(\triangle \log^2 n)$ in asynchronous ones. Furthermore, its detection distance is $O(f \log n)$.*

5. REFERENCES

[1] Y. Afek and A. Bremler-Barr. Self-Stabilizing Unidirectional Network Algorithms by Power Supply. Chicago J. Theor. Comput. Sci. 1998: (1998).

[2] Y. Afek, S. Kutten, and M. Yung. The Local Detection Paradigm and Its Application to Self-Stabilization. TCS 186(1-2): 199-229 (1997).

[3] S. Aggarwal and S. Kutten. Time Optimal Self Stabilizing Spanning Tree Algorithms. FSTTCS 1993.

[4] A. Arora, M. G. Gouda. Distributed Reset. IEEE Trans. Computers (TC) 43(9):1026-1038 (1994).

[5] B. Awerbuch. Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems. STOC 1987: 230-240.

[6] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. A Time-Optimal Self-Stabilizing Synchronizer Using A Phase Clock. IEEE Trans. Dependable Sec. Comput. 4(3): 180-190 (2007).

[7] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. STOC 1993: 652-661.

[8] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-Stabilization By Local Checking and Correction. FOCS 1991, pp. 268-277.

[9] B. Awerbuch and G. Varghese. Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols. FOCS 1991: 258-267.

[10] L. Blin, M. Potop-Butucaru, S. Rovedakis and S. Tixeuil. A New Self-stabilizing minimum spanning tree construction with loop-Free property. DISC 2009.

[11] L. Blin, S. Dolev, M. Gradinariu Potop-Butucaru, and S. Rovedakis. Fast Self-Stabilizing Minimum Spanning Tree Construction. DISC 2010: 312-327.

[12] A. Bui, A. K. Datta, F. Petit, V. Villain. Space optimal PIF algorithm: self-stabilized with no extra space. IPCCC 1999:20-26.

[13] Z. Collin and S. Dolev. Self-stabilizing depth-first search. IPL, 49(6): 297-301 (1994).

[14] C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. PODC 2004.

[15] Y. K. Dalal. A Distributed Algorithm for Constructing Minimal Spanning Trees. IEEE Trans. Software Eng. 13(3): 398-405 (1987).

[16] A. K. Datta, L. L. Larmore, and P. Vemula. Self Stabilizing Leader Election in Optimal Space. SSS 2008: 109-123.

[17] A. K. Datta, L. L. Larmore, H. Piniganti. Self stabilizing Leader Election in Dynamic Networks. SSS 2010: 35-49.

[18] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. CACM, 17(11), 643–644, 1974.

[19] S. Dolev, M. Gouda, and M. Schneider. Requirements for silent stabilization. *Acta Informatica*, 36(6), 447-462, 1999.

[20] P. Fraigniaud and C. Gavoille. Routing in trees. ICALP 2001, pp. 757–772.

[21] R. G. Gallager, P. A. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. ACM TOPLAS, 5(1): 66-77, 1983.

[22] J. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm. FOCS, 1993.

[23] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-Containing Self-Stabilizing Algorithms. PODC 1996.

[24] S.K.S Gupta, P.K. Srimani. Self-stabilizing multicast protocols for ad hoc networks. JPDC, 63(1): 87-96.

[25] L. Higham, and Z. Liang. Self-stabilizing minimum spanning tree construction on message passing networks. DISC 2001: 194-208.

[26] D. Peleg. Distributed computing: a locality-sensitive approach. SIAM, 2000, ISBN:0-89871-464-8.

[27] G.M. Jayaram and G. Varghese. The fault span of crash failures. JACM, 47(2): 244-293 (2000).

[28] S. Katz, K. J. Perry. Self-Stabilizing Extensions for Message-Passing Systems. DC, 7(1): 17-26 (1993).

[29] A. Korman and S. Kutten. Distributed verification of minimum spanning trees. DC, 20(4):253-266 (2007).

[30] A. Korman, S. Kutten, and T. Masuzawa. Fast and Compact Self-Stabilizing Verification, Computation, and Fault Detection of an MST, an extended version of the PODC paper. http://iew3.technion.ac.il/~kutten/KKM2011-ext.pdf.

[31] A. Korman, S. Kutten, and D Peleg. Proof labeling schemes. DC, 22:215–233, 2010.

[32] A. Korman and D. Peleg. Compact Separator Decomposition for Dynamic Trees and Applications. DC, (2008), 21(2): 141-161.

[33] S. Kutten and D. Peleg. Fast Distributed Construction of k-Dominating Sets and Applications. PODC 1995.

[34] Z. Lotker, B. Patt-Shamir, and D. Peleg. Distributed MST for constant diameter graphs. DC, 18(6), 2006.

[35] D. Peleg, V. Rubinovich: A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum Weight Spanning Tree Construction. SICOMP 30(5).

[36] R. E. Tarjan. Applications of path compression on balanced trees. *JACM* 26 (1979), pp. 690–715.