Optimal Maintenance of a Spanning Tree*

Baruch Awerbach† Israel Cidon‡ Shay Kutten§

January 13, 2008

Abstract

"Those who cannot remember the past are condemned to repeat it."
(George Santayana)

In this paper, we show that keeping track of history enables significant improvements in the communication complexity of dynamic network protocols. We present a communication optimal maintenance of a spanning tree in a dynamic network. The amortized (on the number of topological changes) message complexity is $O(V)$, where V is the number of nodes in the network. The message size used by the algorithm is $O(\log |ID|)$ where $|ID|$ is the size of the name space of the nodes. Typically, $\log |ID| = O(\log V)$.

Previous algorithms that adapt to dynamic networks involved $\Omega(E)$ messages per topological change—inherently paying for re-computation of the tree from scratch.

Spanning trees are essential components in many distributed algorithms. Some examples include *broadcast* (dissemination of messages to all network nodes), *multicast, reset* (general adaptation of static algorithms to dynamic networks), routing, *termination detection*, and more. Thus, our efficient maintenance of a spanning tree implies the improvement of algorithms for these tasks. Our results are obtained using a novel technique to save communication. A node uses information received in the past in order to deduce present information from the fact that certain messages were NOT sent by the node's neighbor. This technique is one of our main contributions.

*A preliminary version of the material given in this paper appeared in the Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science (FOCS 90), St. Louis, MO, USA, Oct. 1990.

†Johns Hopkins University, Baltimore, MD 21218, baruch@blaze.cs.jhu.edu

‡Faculty of EE, Technion, Haifa 32000, Israel, cidon@ee.technion.ac.il

§Faculty of IE&M, Technion, Haifa 32000, Israel, kutten@ie.technion.ac.il

1 Introduction

1.1 The Model

In this paper, we consider the classical model of *dynamic* asynchronous communication networks. In such networks, communication between nodes is completely asynchronous, and any sequence of network topological changes is possible [AAG87]. Dynamic networks are a good approximation for realistic network models. See, e.g. [Fin79, MRR80, ACG+90] and Subsection 1.3.

The network is represented by a graph $G = (V, E)$ where V is the set of nodes, and E the set of edges (or links). We use n to denote $|V|$ (when no confusion arises we use $O(V)$ instead of $O(|V|)$, etc.). Nodes communicate only by exchanging messages over the edges. We assume that every node has a unique identity, termed *Id*. This is also translated to a unique *weight* of each edge, which is the concatenation of the link's weight and the *Id*s of its endpoints (first the lower *Id* and then the higher one). We assume, w.l.o.g., that each node knows the identities of its neighbors.

Atomicity of events handling: without loss of generality, we use the following common assumption for asynchronous networks: computation (including sending messages) associated with an input event for a node is performed by the node before another event happens. Input events in a node are the following: (1) the failure of an adjacent edge, (2) the recovery of an adjacent edge, and (3) the reception of a message.

Faults: Edges may fail and recover. A faulty edge does not transfer messages, and messages sent over it before it failed, are lost. (However, since this is an asynchronous model, the sender does not know exactly when the edge failed; hence, if it sent some message and after that was notified that the edge failed, it cannot know whether the messages were delivered to the other endpoint or not). Whenever an edge fails, an underlying lower-layer link protocol notifies both endpoints of this edge about the failure, before the edge can recover. This means that an edge that fails, fails in both directions. Similarly, each endpoint is notified of its edge recovery. A message can be received only over an edge that is not faulty. If an edge (u, v) failed at u before some messages sent by v to u have arrived, then these messages will never arrive, even if the edge recovers.

Since there may be times where an edge (u, v) may be faulty at u but not at v, it makes sense to sometimes consider the edge as a pair of two directed edges: (u, v) and (v, u). Nevertheless, when no ambiguity arises, when we refer to a faulty edge, we mean the undirected edge that is faulty at either one of its endpoints (or both). The paper deals with the failure and recovery of edges. As we note at the end of Section 6, the results also apply to the failure and recovery of a node and its memory.

Non-faulty edges: Messages transmitted over a non-faulty edge will either arrive, or the edge will fail (will "become faulty") at both endpoints. Messages that arrive over

any given link arrive according to the FIFO discipline. However, no upper bound is known for the arrival time of a message over a link. Hence, two independent messages sent over two links may arrive in any order.

Complexity: In this paper we are interested in the following complexity measure: A message sent by the algorithm contains $O(\log |ID|)$ bits, where $|ID|$ is the size of the name space of the nodes. Typically, $\log |ID| = O(\log V)$.

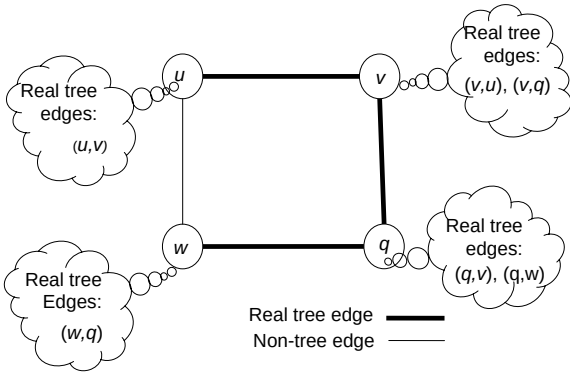
Definition 1.1 *The* amortized communication *is the total number of messages sent by the protocol, divided by the number of input events that occurred during the execution.*

In other words, this is the "incremental" communication cost, in terms of the number of messages caused by a single topological change.

1.2 The Problem Statement

We assume that the network starts with no edges. A node receives, as input, a (possibly infinite) sequence of *local* events called *topological changes*. Each such change is the insertion (recovery, becoming operational) or deletion (failure) of one of its own edges. Note that a node is aware of all its local events, but is completely unaware of the local events happening at other nodes, unless those events are communicated to it by those other nodes. For a deletion event, there exists some time where the edge is deleted at both its endpoints.

Each node maintains in its local memory a subset of the set of all its own emanating edges. We term the edges in this subset *real tree edges*. See Drawing 3. For that purpose, we consider each edge (u, v) to be composed of two directed edges: (u, v), directed from u to v, and (v, u), directed from v to u. Note that (directed) Edge (u, v) becoming a real tree edge is a local decision of its endpoint u. No other node is aware of that, unless this fact is communicated to it.



Drawing 3: a real tree

When no ambiguity arises, we sometimes talk about the undirected real tree edge (u, v). This term refers to an edge that is (1) not deleted at any of its endpoints (that is, non-faulty), and (2) this edge is considered a real tree edge by either one of its endpoints, or by both of them.

Definition 1.2 *The collection of (the undirected) real tree edges is termed the* real forest*.*

A connected component of the real forest is termed a real tree*.*

The task of the tree maintenance algorithm is

(1) to maintain the invariant that the real forest is indeed a forest at all times;

(2) if the topology changes cease, to have the forest converge to a spanning tree.

In practice, there is no need to require that the changes cease permanently; stability for "long enough" periods suffices. This model realistically describes the mode of operation in existing networks, where topological changes occur in bursts.

A spanning tree is an important data structure in a communication network, often used for broadcast of information through the network, routing, and control. Since the communication graph of the network is constantly changing, it is important to update trees in order to adapt communication paths to those changes. A spanning tree protocol was used in networks of DEC. A similar protocol also became an IEEE standard. Both protocols were developed by Radia Perlman [P99]. These protocols were based on the sequential algorithm of Dijkstra (e.g. [Eve79]). Hence, the tree is built from a root that is preconfigured. A distributed maintenance was first proposed for the IBM PARIS experimental network [ACG$^+$90, CGKK95], and later used in the IBM NBBS ATM network [MICG95]. Theoretically, the complexity of this algorithm is unbounded, since it uses counters that are incremented with every topological change. Spanning trees are also used, for example, in CISCO networking products, see e.g. [BS04]. Recently, there are suggestions to use distributed tree maintenance for organizing ad hoc wireless networks. See, e.g. [GRSV03]. There is also interest in exploring the possibility of maintaining a spanning tree to reduce the amount of routing information exchanged by Internet routers, e.g. in OSPF. See, e.g. [IETF].

One can easily see that any dynamic spanning tree protocol requires $\Omega(V)$ in amortized communication complexity. (For example, consider two networks each being a line of $V/2$ nodes, and $V/2-1$ edges; assume that two edges now recover, merging the two lines into one simple cycle; clearly, any distributed algorithm that does not send at least about $V/2$ messages may result in either a cycle or a disconnected structure rather than a spanning tree). In this paper, we show that the amortize communication complexity is also $O(V)$.

1.3 Previous Approaches

The problem of computing on dynamic networks is one of the most well-studied problems in the area of distributed computing. See, e.g. [Gal76, Gal77, MS79, JM82, Gaf87,

AAG87, AS88, Awe88, AAM89, Hum81, Gar89, CRKG89, RF89]. The main motivation for studying dynamic networks rather than static networks is that they are a better model for real networks, which experience failures [FLP85] and addition of new links. There is also a stronger motivation for the efficient implementation of a repetitive task, the typical task in dynamic networks, rather than a one-time task. Various tasks need to be computed in dynamic networks, such as shortest paths, minimum spanning tree, BFS, DFS, maximum flow, minimum cost flow, among others.

Since, in the early days of the field, computing in dynamic networks appeared to be a hard task, many researchers approached the problems in dynamic networks in the following way:

1. Find an efficient "from-scratch" solution in a static asynchronous network.
2. Design a "Reset" procedure, that "blasts away" the existing computation, and restarts the new computation "from scratch".

It is interesting that computing from scratch for many important functions, e.g., a spanning tree, requires $\Omega(E)$ communication [AGPV90] or even $\Omega(E + V \log V)$ [KMZ89]. Thus, the method above is doomed to $\Omega(E)$ amortized communication.

However, an $\Omega(E)$ lower bound does not necessarily apply to the amortized communication complexity of dynamic network protocols. Intuitively, this is because the latter reflects the (incremental) amount of work performed to adapt to a single topological change. Since the work is not performed from scratch, we can benefit from the knowledge gained in the past and thus economize on communication. Our goal here is to prove that this is indeed possible, i.e., adapting to a single change is easier than repeating the whole computation over the newly formed topology. In this paper, we demonstrate this for the maintenance of a spanning tree. Given an algorithm that maintains a spanning tree, the preliminary version of this paper [ACK90] was the first to show that adapting to a single change is easier than computing from scratch, not only for the maintenance of a spanning tree, but also for the many other tasks that require an $\Omega(E)$ communication when computed from scratch.

It is well known that this is the case in sequential computation. A famous example: Frederickson [Fre83] shows how to maintain a dynamic minimum-spanning tree with (amortized cost) $O\left(\sqrt{E}\right)$ computations per input change, while constructing a tree from scratch requires $\Omega(E)$ computations.

Unfortunately, in the distributed computation model, it was far from obvious that it was possible to reduce the incremental cost below the cost of solving the problem from scratch. In fact, it took a long time (from 1976 [Gal76] till 1987 [AAG87]) just to implement the "blast away" itself in $O(E)$ per topology change. One of the by-products of this paper is improving the amortized message complexity of the "reset" task itself further from $O(E)$ to $O(V)$, thereby making it message-optimal.

The best previous solution for the task of maintaining a spanning tree in a dynamic network [AAG87] requires $O(E+V \log V)$ messages per topological change. This matches the performance of protocols that solve the problem "from scratch". Another disadvantage in using [AAG87] for a dynamic spanning tree is that even parts of the tree that do not suffer topological changes may be replaced as a result of a change; this is often undesirable. In [AAG87], bounded message size was assumed (as in the current paper). Previous work that allowed unbounded message size [Gal77, MS79, KM86] does not achieve better performance. The [AAG87] protocol adopts the "classical" [Gal76],[Fin79],[Seg83],[Gaf87], [GA87], *blast away* approach mentioned above for the problem of designing a dynamic protocol. With that approach, one clearly cannot improve over the performance of the static protocol.

There were previous attempts in the literature to save some of the waste associated with the blast-away approach by keeping track of the past computations and using them in the future [MS79, SG89, Gaf87, AS88]. For example, in [SG89], a new principle was introduced to decide which message is relevant and which is obsolete. In [Gaf87], this is generalized to a family of principles. However, [Gaf87] also conjectured that in terms of worst-case complexity, the blast away method is the best algorithm. We disprove this conjecture in the present paper.

Oddly enough, previous methods which interpolate the information from the past actually proved less efficient than the naive "blast away" method of [AAG87] in terms of communication complexity. Intuitively, the reason is that the mixing of the results of the old computation with that of the new one "confuses" the nodes and thus they make wrong decisions. Those decisions cause them to send messages that have to be followed later by correction messages. We overcome these phenomena in this paper.

1.4 Our Results

The main result in this paper is that by keeping track of history, it is possible to construct a communication-optimal protocol for maintaining a spanning tree.

Theorem 1.1 *A spanning tree can be maintained with $O(V)$ amortized communication.*

This proves that amortized communication complexity, i.e., the incremental cost of adapting to a single change, can be *smaller* than the communication complexity of solving the problem from scratch. No similar result has been previously reported.

We introduce a dynamic *tree maintenance algorithm* that maintains a loop-free forest structure at all times, and converges to a spanning tree when the topological changes stop (for a "sufficiently long" time). The amortized message complexity is $O(V)$.

The trees generated by old computations are not discarded when new topological changes disconnect them. Instead, the disconnected parts are "glued" together to form the new tree. Moreover, the "gluing" task uses the old tree for passing its messages, thus achieving the $O(V)$ complexity. Hence the new computation is helped rather than hindered by the results of the old computation.

Several problems arise in implementing the above approach. One is choosing the gluing edges. Note that inherent in the asynchronous model, there are transient cases where some nodes "have heard" of topological changes, while others have not. Since different nodes have different information they may choose different edges, causing either deadlocks or cycles in the "tree" if no special care is taken. Another problem relates to sending updates about changes over the tree. That is, to preserve the $O(V)$ amortized complexity, a node must not receive the same information more than a constant number of times. However, the tree edges are dynamically replaced by other edges concurrently with the distributed process of the updates flowing over the tree. How can we, nevertheless, prevent each node from receiving such an update from several directions, and thus, more than once?

Both of the above problems are consistency problems in the environment of asynchronous and dynamic networks. Our solution is a novel tool, called *tree belief principle*. It enables neighboring nodes to resolve inconsistencies between their views of the forest structure.

Given an efficient algorithm for dynamic tree maintenance, many tasks can be performed efficiently. In [ACK90], it was shown how to use such a tree to perform *topology update* with $O(V)$ amortized communication complexity. Topology update is the task where each node learns the network graph. It is heavily used in communication networks for computing next hop routing tables [MRR80, ACG$^+$90, CGKK95, T02, HS89]. Moreover, when a node has learned the network graph, the node can perform any sequential graph algorithm on that graph. Thus, the computation of any graph problem on the network graph is trivially reduced to topology update. The solution for topology update was generalized in [ACK90] to $O(V)$ amortized communication maintenance of any polynomial size local database. This implies the immediate solution of any problem for which the input is the network graph. Moreover, it also solves any problem for which the input includes the network graph as well as any attribute (of polynomial size) associated with nodes and/or edges.

Another task for which the dynamic tree can be used is the above-mentioned Reset procedure ([Fin79]), improving its amortized communication complexity from $O(E)$ [AAG87] to $O(V)$. This improves (in the dynamic networks model) the amortized complexity of other tasks that relay on the Reset procedure, e.g., the complexity of the end-to-end communication problem [AG91, AAG$^+$, AG88] is improved from $O(E)$ [AAG87] to $O(V)$.

Broadcast is the task of delivering a message to every node in the network. *Multicast* is the task of delivering a message to a specific subset of the nodes. Both tasks are highly necessary in communication networks, and are greatly simplified given an algorithm to maintain a spanning tree [AGKK]. Synchronizers [A85] and clock synchronization are also examples of tasks that benefit from dynamic spanning trees.

1.5 Bounded Counters

We follow the recommendation of [AAG87, AG88, AAG+, AMS89, AAM89, AGH90, AG91, AGR92], in not using unbounded counters. "Unbounded counters" refers to counters which are incremented every time a topological change occurs in the network, theoretically counting to "infinity". However, such counters are frequently used in practice, since a relatively small counter (say, 128 bits per message) suffices to represent a huge number of topological changes.

Considering the above, one could question the practical value of attempts to avoid the use of such unbounded counters. This is even more questionable regarding the fact that the results in many previous papers can be trivially obtained (or even improved significantly) if unbounded counters are permitted [AAG87, AG88, AMS89, AAM89, AGH90, AG91, AGR92]. The results of this paper, however, are not trivialized if unbounded counters are permitted. In fact, the results reported here, as well as results based on this paper, improve the results in papers that do use unbounded counters [BGJ+85, Vis83, CCK88, BO99].

There are also practical reasons for avoiding the use of unbounded counters. One of them is to overcome a potentially fatal case of failure: that of a loss of nodes' memory (and consequently, forgetting the highest-used counter values). Another is the difficulty of handling such counters in a specialized hardware switch in fast networks [CGKK95].

1.6 Practical Applications of Our Work

Recall that Requirement (2) in Subsection 1.2 is only eventually required. On the other hand, Requirement (1) (termed *Loop Freedom*) is required to hold at all times. This is also the case with the following additional requirement:

Path-Preservation: `An edge ceases being a "tree-edge" only in the case where it fails`.

Loop-Freedom is essential in the environment of hardware-based fast packet switching [Tur88, CG88, ACG+90, CGK88, CS88]. Path-Preservation is important in a virtual circuit-switching environment. The properties of loop freedom and path-preservation have been studied for a long time [Gal77, MS79, SS81] in the networking literature. Our protocol is the first one that achieves those properties with bounded complexities.

1.7 Structure of the Paper

Sections 2 and 3 explain the tree maintenance algorithm. Section 2 presents the algorithm except for two subroutines FIND and UPDATE. The UPDATE subroutine, which is our main technical contribution in this paper, is presented in Section 3 together with the FIND subroutine. Section 4 explains the distributed implementations of the modules. Section 5 and Section 6 contain the proofs of correctness and the message complexity analysis. The appendices contain some details about known techniques used applied by some out the subroutines, and a pseudo code (a high-level description of that code appears in the body of the paper).

2 Preliminaries

In Subsection 2.1, we describe the main program of the algorithm. It is extremely simple, though it contains a surprising action: one of the subroutines (UPDATE) is called twice in a row. This seemingly redundant operation is crucial for the correction of the algorithm. UPDATE, one of the main contributions of the paper, is described in Section 3, together with the complementing Subroutine FIND.

2.1 Informal Description

For ease of description, we first describe the tasks performed by the algorithm as if the algorithm were not distributed. (The computational complexity of the "non-distributed" algorithm does not interest us.) Recall that the most novel part of the algorithm appears in the next section (3).

2.1.1 Overview

Recall that the *real forest* maintained by the algorithm is composed of locally marked edges. The failure of an edge causes it to become unmarked, and disconnects a *real tree* into two real trees. Multiple topological changes can create a forest of many trees. The algorithm "glues" distinct real trees of this forest into larger real trees, by marking edges that connect them.

Following [GHS83] (the construction of a spanning tree for a *static* network), we adopt a concurrent implementation of Kruskal's algorithm (see, e.g., [Eve79]): Each tree tries to connect to other trees using its *minimum-weight outgoing edge*, namely the minimum-weight edge with exactly one endpoint belonging to that tree. We *attempt* to mark an edge which is the minimum outgoing edge of the real trees of both its endpoints.[1] This operation is repeated as long as there are more than one real tree in a connected component of the network.

In a distributed network, the traditional method of detecting that an edge is indeed outgoing would be (1) to give each real tree a name known to all its nodes, and (2) to have the endpoints of every edge exchange messages to compare the names of the real tree in which they are members [GHS83]. (If they are members of real trees with different names, then this is an outgoing edge.) We cannot use this approach since it clearly leads to a message complexity of $\Omega(E)$.

The primary new technique we use to reduce the complexity from $O(E)$ to $O(V)$ is a method to update dynamic data structures, so that finding an outgoing edge will take only $O(V)$ messages. The data-structure is explained in Subsection 3.1, the update in Subsections 3.3 and 3.4, and finding the minimum outgoing edge in Subsection 3.5. Before describing them, let us summarize the main program.

[1] We say "attempt", since, at the same time that we use the minimum edge to merge two trees, a still lower weight edge may recover. This may render the merger edge non-minimum. If no special care is taken, this may cause not only "non-minimum" but also deadlocks. Overcoming the difficulties this phenomena can cause is discussed in Section 4.2 and the proof of Lemma 5.16.

2.1.2 Main Program

The high-level description of the algorithm outlined above appears in Figure 1. Details about the distributed implementation are deferred to Section 4. At this point, we just state that the algorithm maintains the invariant that every real tree has a root, one of its nodes, that manages the tree operations (similar to [GHS83]). The root invokes the FIND distributed subroutine to find the minimum outgoing edge of a real tree. The role of being a root is passed from node to node until the root is the endpoint of the minimum outgoing edge. Then the root negotiates a merger with the node at the other endpoint.

Subroutine UPDATE is invoked by the root in order to update a data structure, such that the complexity of Subroutine FIND will be reduced. A crucial action of the algorithm is calling UPDATE both before calling FIND, and again after the two roots agree on a merger. The first call has to do with the complexity of the algorithm, while the second call is needed for the correctness of the algorithm. The reason is explained in Subsection 3.3.

3 UPDATE and FIND

In this section, we present the primary new technique used for reducing the message complexity—the subroutines used by the main program described in Section 2. Subsection 3.1 introduces the dynamic data structure used to reduce the complexity of finding an outgoing edge. Subsections 3.2 and 3.3 describes the properties of the subroutines. The UPDATE subroutine is described in Subsection 3.4, and the FIND subroutine in Subsection 3.5.

```
Whenever a marked edge fails
   unmark edge (*at the endpoints*)

Whenever two trees merge or a topological change occurs
   call UPDATE (*correct tree replicas*)
   (*when UPDATE terminates*) call FIND (*choose min outgoing edge*)

Whenever two trees choose the same minimum outgoing edge
   for each of the trees separately, call UPDATE
  (*when UPDATE terminates*) mark the chosen edge at both of its endpoints (*merge*)
```

Figure 1: Main algorithm

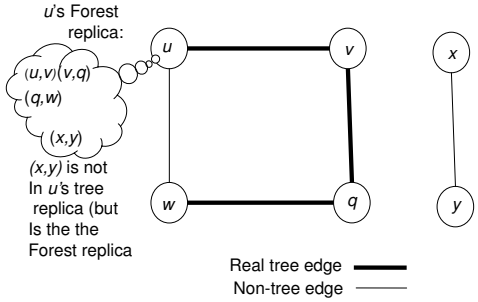
3.1 Basic Data Structures

Our goal is to achieve $O(n)$ amortized message complexity. The approach is similar to that of dynamic data-structures in sequential algorithms: the complexity of "find"

operations is reduced significantly, at the expense of a slight increase in the cost of "update"s. Intuitively, the update is used to let every node "know" the description of the real forest. Thus, a node "knows" which of its neighbors is not in its real tree. Hence, the edge to that neighbor is outgoing. Unfortunately, the actual algorithm is more complex than that because of asynchrony. It may happen that a node will not have the accurate description of its real tree. For example, by the time a node receives a message that some edge left the real tree, that edge may have already rejoined. The algorithm overcomes such cases.

Let us present our dynamic data structure in more detail. The description of the real forest, maintained by a node, may be different than the actual set of marked edges (real forest). Thus, we term it the *forest replica* of the node. However, the real forest and the replica of Node v agree on v's local edges. This is by definition: v's own adjacent edges that appear in v's replica are exactly v's marked edges.

If Node v's forest replica includes several trees, still v itself and all its marked edges belong to exactly one of these trees. We call this tree v's *tree replica*. Intuitively, v's tree replica is an approximation of the real tree to which v belongs. The tree replica of each node is the tool used by Subroutine FIND to identify the outgoing edges. Note that a node's forest replica may include other trees! They are used (by Subroutine UPDATE) to lower the cost of updating the tree replicas when the trees rejoin. A tree replica (as a part of a forest replica) is demonstrated in Drawing 4.



Drawing 4: a forest and a tree replica

The algorithm attempts to keep the tree replicas of all the nodes as "accurate" (i.e., close to the real forest) as possible. To this end, a node that performs a change in the marking of its adjacent edges (unmarking as a result of failure, or marking as a result of "gluing" trees) updates its forest replica, and communicates the change over the marked edges to its whole real tree, using Subroutine UPDATE. (This may not be done immediately, this node first alerts the root to invoke UPDATE if the node is not currently involved in an UPDATE execution.) A node instructs its neighbor to add an edge to the neighbor's tree replica using a list called "Add", carried in a message called "DIFF". In a sense, the contents of such messages are also a part of the distributed "data structure" held in the network. Let us define this distributed data structure more precisely.

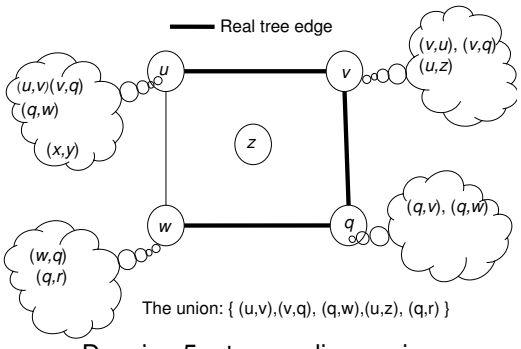
This data structure is distributed over a real tree, and captures the collective view of the nodes of this real tree. The distributed data structure consists of (1) items stored in all the nodes of a real tree, as well as (2) items carried in "Add" lists in DIFF messages.

Definition 3.1 *Given a real tree at any given time during the execution of the algorithm,*

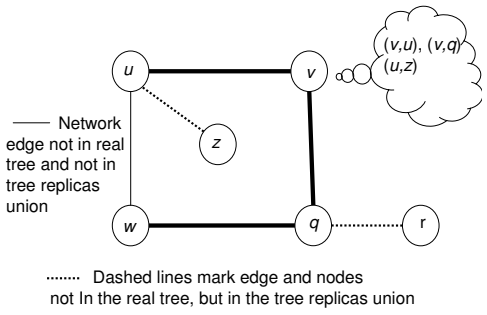
1. *Let U_{nodes} be the union of the tree replicas of the nodes in the given real tree (just the tree replicas, not the other parts of the forest replicas).*
2. *Let U_{msgs} be the union of the "Add" lists of DIFF messages in transit on edges of the given real tree.*

The trees replicas' union *is the union of U_{nodes} and U_{msgs}.*

Note, that the above union contains the real tree itself (since every marked edge appears in the tree replica of its endpoint). However, it may contain additional edges, e.g., edges leading to other real trees. Intuitively, this *tree-replica' union* of a real tree contains every edge that is "believed" by some nodes in the real tree to belong to this real tree (Drawing 5). The union includes an item of an "Add" list because the node that sent this DIFF message "believed" that this edge was in the real tree at the time the node sent the message.



Drawing 5a: tree replicas union



Drawing 5b: tree replicas union

3.2 Properties

The key properties of the subroutines (stated more formally and proven in Section 5) are:

- Subroutine UPDATE is called whenever a topological change occurs, or trees merge (both before and after the trees merge). It updates the tree replicas before the initialization of Subroutine FIND. Its key properties are:

    1. While UPDATE operates in a real tree, no edges can be added to the real tree, though the real tree can shrink as a result of failures of tree edges. (Intuitively, this property is guaranteed by the main program: the real tree does not merge with another while UPDATE is performed; thus, it cannot grow; see the first line of Figure 1.)
    2. The tree replica at each node upon the termination of UPDATE is a subset of the real tree to which the node belonged upon invocation of UPDATE, and a superset of the real tree upon termination of UPDATE. (Intuitively, by the previous property, this means that UPDATE indeed learns the current structure of the real tree, except, possibly, for failing to report some of the failures that may have occurred during its run.)

- Subroutine FIND is called after termination of UPDATE. It finds a minimum-weight outgoing edge of a real tree. (If no edge can be found, then the algorithm stops until a topological change occurs.) The key properties of FIND are similar to those of UPDATE:

    1. During the operation of FIND, no new edges are added to the real tree (although the real tree can still shrink as a result of failures of tree edges).
    2. The edge selected by FIND is a minimum-weight outgoing edge of the real tree at some time during its execution. (More precisely, it is a minimum-weight outgoing edge of a snapshot [CL85] of the real tree.)

3.3 The Strong Loop Freedom Invariant

The invariant defined below plays a major role in the analysis of the algorithm. We present it here in order to give some insight into the structure of the algorithm. (Section 3.4. describes how the invariant is used.)

Definition 3.2 (Strong loop freedom invariant) *We say that a tree replicas' union preserves* strong-sense loop freedom *if for every real tree, its tree-replicas' union does not contain a cycle.*

We show that this is an invariant of the algorithm for every real tree. Intuitively, when the invariant is kept, the tree replicas of nodes of a particular real tree may be different, but they maintain some consistency. This is important, since in distributed operation, one cannot avoid differences between the views of nodes. For example, a node may have learned about a failure of a marked edge but not yet notified other nodes. Thus, the tree replica of this node is different than those of the other nodes

in the same real tree. In this example, the strong loop freedom invariant implies that this node does not choose a replacement edge before the other nodes are notified of the failure. This is because until the other nodes in the same real tree are notified of this failure, the failed edge is still in the tree replicas' union; so, choosing a replacement edge would have introduced a cycle into that union.

The following action of the algorithm may look redundant. In fact, it is a crucial action required to maintain the above invariant: UPDATE is performed first separately, in two real trees that chose to merge. UPDATE is then performed again in the merged tree. The sole change in the merging real trees (aside from possible edge failures) between the two executions, is the marking of the edge that connects the real trees. Nevertheless, the correctness of the algorithm is based on this seemingly redundant repetition. Let us now explain the intuition behind this. (For a more formal analysis, see Section 5.)

Given a particular real tree for which the invariant holds, it is relatively easy to see that the operations do not violate the invariant as long as the real tree does not merge. However, when two real trees ("left tree" and "right tree") merge, the invariant can be violated, even if it holds for each of these real trees separately. For example, some node v may appear both in the union of tree replicas of "left tree", as well as in the union of tree replicas of "right" real tree. (Intuitively, there are nodes in the left real tree that "believe" that v is in the left real tree, and nodes in the right real tree that believe that v is on the right.)

Note that v actually belongs to, at most, one of these real trees. Thus, at least in one of those replicas, some edge e leading to Node v appears erroneously, and should be removed.

- Node v may have belonged to, say, real tree "left";
- later, Node v could be disconnected when some Edge e failed, see Property (1) of UPDATE.
- later, v merges into real tree "right" before "right" chose to merge with "left".

Thus, before UPDATE is executed, v may appear both in tree replicas in the "right" real tree, and in (some) tree replicas in the "left" real tree.

In the example above, the purpose of executing UPDATE in each of these real trees before the merge is to remove the said erroneous edge—see Property (2) of UPDATE. This preserves the invariant.

The execution of UPDATE again in the merged tree, is intended to let the nodes learn about nodes that were added to their tree by the merge, so that FIND can be performed correctly.

3.4 Subroutine UPDATE

Subroutine UPDATE attempts to make the tree replicas of all nodes on a real tree identical to the real tree. For each marked edge, it calls procedure LOCAL-UPDATE which makes the tree replicas of the edge's endpoints identical. (Note that other trees of the forest replicas can remain different at the two endpoints.) The pseudo-code of LOCAL-UPDATE appears in Figure 2.1 using functions defined in Figure 2.2. The subscript of a variable designates the node in which the variable resides. Thus, Forest_v is a variable of Node v. $\text{Forest}_v(k)$ is the estimate in Node v of the forest replica of its neighbor k. $\text{Forest}_v(v)$ is the forest replica of Node v.

The (distributed) subroutine UPDATE terminates after all the LOCAL-UPDATE procedures have terminated. (The distributed implementation of UPDATE is explained in Section 4.) Consider some sample marked edge, over which we execute procedure LOCAL-UPDATE. Let us call the endpoints of the sample edge Node "left" and Node "right".

Whenever LOCAL-UPDATE is invoked or after processing a DIFF message
    $\forall$ k s.t. (v, k) is marked and $\text{Forest}_v(k) \neq$ undefined
    (*$\text{Forest}_v(k)$: local mirror at v of (remote) forest of k*) do
        send message DIFF(diff(k)) to k
        (*foreseeing the way k will change its Forest_k replica:*)
        $\text{Forest}_v(k) := \text{Forest}_v(k) \oplus \text{diff}\ (k)$

Whenever (*receiving message*) DIFF(Add, Del) from k
    $\text{Forest}_v(v) := \text{Forest}_v(\text{v}) \oplus$ (Add, Del) (*updating forest replica*)
    $\text{Forest}_v(k) := \text{Forest}_v(k) - \text{tree}^k\ (\text{Forest}_v(k) - \{(v, k)\}) \cup \text{tree}^k\ (\text{Forest}_v(v) - \{(v, k)\})$
    (*deducing forest replica update performed in k*)

Figure 2.1: Procedure LOCAL-UPDATE, code in Node v (see definitions in Figure 2.2)

Definition: Let A, B, C be a set of edges.

$A \oplus (B, C)$ is defined as $(A - C) \cup B$.

A proj B is defined as $\{(x, y)$ s.t. $((x, y) \in A$ and $\exists\ k$ s.t. $(k, y) \in B)\}$

function diff(k: neighbor; returns ((Add, Del): pair of lists of edges)

(* Add: list of edges to be added to Forest$_v(k)$ since they are in Forest(v), and are on *)

(* v's side of the tree. Del: edges that k should delete since they don't appear in v's *)

(* Forest although they are on v's side of the tree (even according to k's Forest) *)

Add := ((Forest$_v(v)$−Forest$_v(k)$) proj my-side(k)

Del := ((Forest$_v(k)$−Forest$_v(v)$) proj my-side(k)

return(Add, Del)

Functions My-side(k : neighbor; returns a tree)

(* returns v's side of tree$_v^v$ relative *)

(* to its edge from k *)

return (tree^v(Forest$_v(v) - \{$ Edge $(v, k)\}$))

Function tree^i (f: forest; returns a tree)

(* returns the tree in f that includes Node i *)

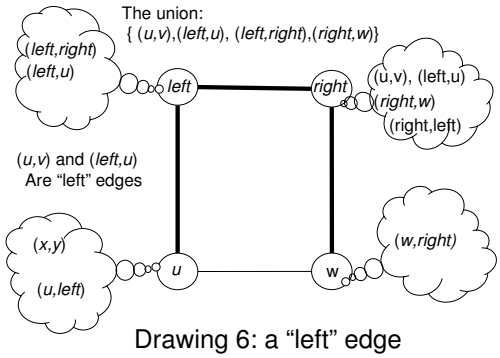
(* convention: Tree, or $tree_v^v$, is tree$^v(Forest_v(v))$*)

Figure 2.2: Functions used by Procedure LOCAL-UPDATE in Node v

The main difficulty is the following: In case the tree replicas of the endpoints of an edge disagree (regarding the status of an edge), it is not obvious how such an "agreement" can be reached, or which one of the endpoints should be determined to be "more correct". (Intuitively, "left" is "more correct" than "right" regarding some edge, if "left" "knows" the marking (or unmarking) as it existed in the real forest at a later time than the one "known" to "right".) However, assuming the above strong loop freedom invariant, procedure LOCAL-UPDATE makes tree replicas identical in an unambiguous and "correct" way.

Consider again a sample edge in a real tree. Using the invariant, for each edge (u, v) in the tree-replicas' union, there is a unique, undirected path in the union that

starts with Edge (u, v) and ends with the sample edge. If this path enters the sample edge through the left (right) endpoint, then we call Edge (u, v) a "left" ("right") edge (Drawing 6). Sometimes we call this edge a "left"-sided edge. Similarly, nodes u and v are "left"-sided nodes.



Drawing 6: a "left" edge

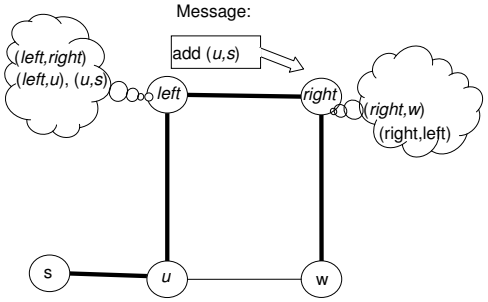
If the replicas of "left" and "right" differ about the status of Edge (u, v), and this edge is a left edge, then endpoint "right" corrects its replica to agree with the replica of "left" regarding Edge (u, v) (see Figure 2.2). It will be shown that the tree replica of "left" is "more correct" regarding "left" edges. See (Drawing 7a). Note that if it is a "left" edge according to the replica of one of the endpoints (either of "left" or of "right") then it cannot be a "right" edge in the replica of the other endpoint (because of the strong loop freedom invariance).

In particular, "left" edges that do appear in the replica of "left" but not in the replica of "right" are added to the latter. On the other hand, a left edge that does not appear in the replica of the left endpoint is removed from the replica of the right endpoint (Drawing 8). Similarly, the right endpoint replica is considered "more correct" regarding right edges. We use the name *tree belief principle* for this method of deciding which is "more correct" about an edge according to which is closer on this edge over the tree. We note here that Spinelli and Gallager were the ones to introduce a belief principle for the same purpose [SG89]. However, the specific belief principle in [SG89] leads to an exponential message complexity, while the belief principle used here (together with the strong loop freedom invariant) leads to a linear complexity.

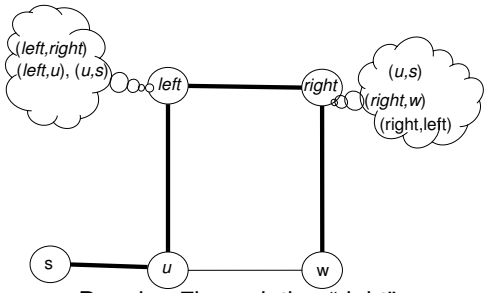
3.4.1 Neighbor Knowledge

Intuitively, UPDATE saves communication by sending a node only information this node does not already "know". To do that, for each real tree Edge $(left, right)$, each endpoint, e.g., *left*, keeps a mirror $\text{Forest}_{left}(right)$, which is an approximation of the forest replica of the other endpoint $right$.

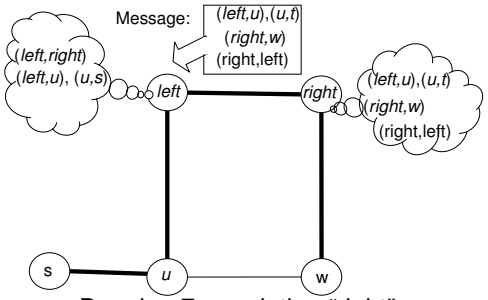
Whenever a new edge $(left, right)$ is marked, each of its endpoints, e.g. $left$, simply transmits its whole forest replica over the newly marked edge to the other



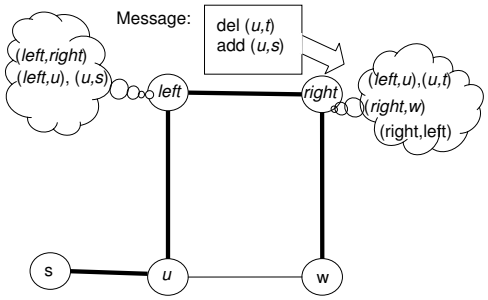
Drawing 7a: updating "right": inserting "left" edges



Drawing 7b: updating "right": inserting "left" edges

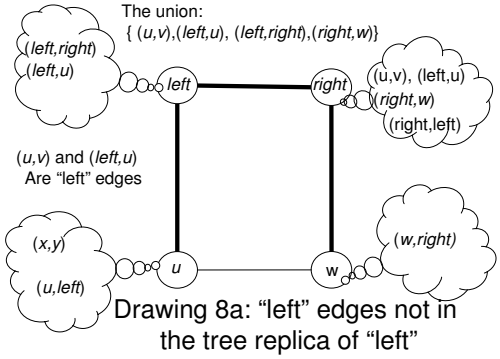


Drawing 7c: updating "right": learning "right"'s views

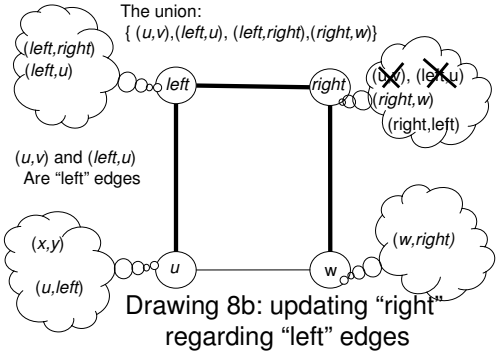


Drawing 7d: updating "right": inserting and deleting "left" edges

---



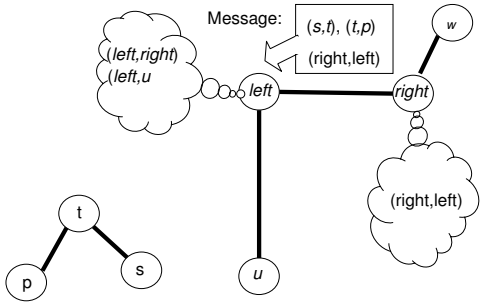
Drawing 8a: "left" edges not in the tree replica of "left"



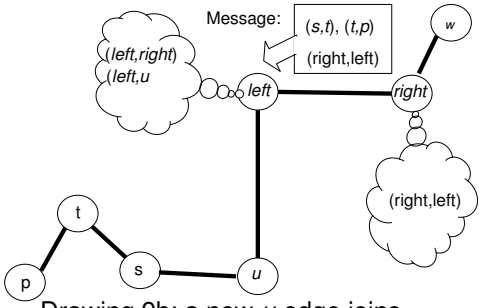
Drawing 8b: updating "right" regarding "left" edges

endpoint $right$. Thus, in later activations of LOCAL-UPDATE, endpoint $left$ already has a mirror $\text{Forest}_{left}(right)$ of the replica of $right$. This mirror is accurate at least regarding left edges, since any change in the replica of $right$ regarding $left$ edges must result from an update message coming from $left$ to $right$. Thus, it suffices that the left endpoint transmits only corrections to this mirror, rather than transmit the entire replica of the left node (Drawings 7c and 7d). The resulting reduction in the message complexity is demonstrated in Example 3.1.

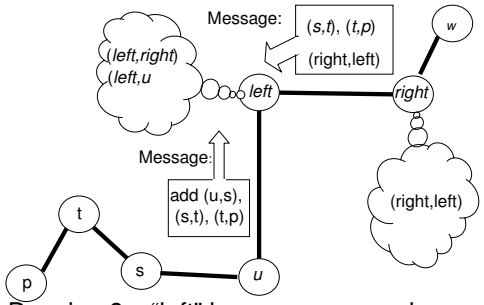
Example 3.1 *Consider the case that "left" and "right" are in one real tree T, and "left" has learned about a newly marked "left" edge that connected real tree T to another*



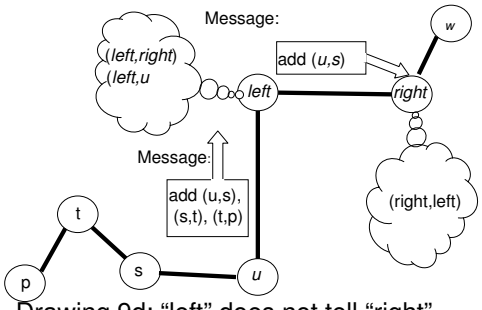
Drawing 9a: "left" learns "right"'s forest replica



Drawing 9b: a new *u*-edge joins the real tree



Drawing 9c: "left" learns a new *u*-edge (a "left" edge relative to edge (*left, right*))



Drawing 9d: "left" does not tell "right" about edges that "right" already knows

real tree T_1 (Drawing 9). Assume further that "left" sees in its mirror of "right" that the forest replica of "right" already contains the description of real tree T_1. Now, "left" must send "right" only the information about the newly marked left edge, rather than the whole description of real tree T_1 that now became a part of T.

3.5 Subroutine FIND

To determine which of its edges is outgoing, a node simply considers its tree replica. If the node has an edge to a neighbor that is not in this tree replica, then the node deduces that this is an outgoing edge. This still leaves us with the task of comparing the outgoing edges known to different nodes of a real tree, in order to find the minimum. We use a method that is taken from [GHS83] and is, by now, considered standard in distributed network algorithms. Since this is a very well known method, we describe it in Appendix A. Details regarding its (rather straightforward) adaptation to the environment of topological changes are discussed in Section 4.

4 Distributed Implementation of the Tree Maintenance

We elaborate on the high-level overview given in Sections 2 and 3. We first describe the implementation of the main program (Figure 1) assuming the existence of subroutines (UPDATE and FIND) that tell each node which of its edges is outgoing. (The subroutines, described in Subsections 4.1.2–4.1.4, are the distributed version of Subsections 3.4 and 3.5.) For the sake of clarity, we first describe the actions the algorithm takes from the point that no additional topological changes occur (Subsection 4.1). (Note, though, that it is never known whether additional topological changes will occur.) Later (Subsection 4.2), we describe the additional steps taken when topological changes occur during execution.

4.1 Operations When No Additional Topological Changes Occur

Distributed tree representation: First, recall that each node maintains a subset REAL-TREE-EDGES of its adjacent edges. The union (over all the nodes) of these subsets is the real forest. Second, each node maintains a *parent* pointer that can point either to nil or to one of its real tree edges. A node whose parent pointer is nil is called a *root*.

4.1.1 Main Program

Recall that the high-level description of the main program appears in Figure 1. In the distributed implementation, each real tree has a unique root which is one of its nodes. The root coordinates the connection of this tree to others. The first task of a root is to activate Subroutines UPDATE and FIND (described in Section 4.1.2). The subroutines eventually terminate, and their terminations are detected at the root. At that time, each node has a pointer to the next node on the route to the minimum outgoing edge. The root then transfers the rootship to the next node on that route. See Section 4.1.3 for more details. The rootship continues to migrate until it reaches the node adjacent to the minimum outgoing edge.

When this node u becomes the root, it tries to reach an agreement with the other endpoint of the minimum outgoing edge (u, v) to merge the two real trees (this is described in Section 4.1.4). Consider the case that the other endpoint v is also the root of its own real tree. If Edge (u, v) is also the minimum outgoing edge of v's real tree, then eventually u and v agree to merge their trees by marking (u, v). Recall that this subsection deals with the case when no additional topological changes occur. Such an agreement is not guaranteed in the case of topological changes (e.g. in the case of

an edge recovery), unless we take additional steps, as described in Subsections 4.1.4, 4.2.5, and 4.2.6.

The higher *Id* endpoint (e.g. v) is chosen to be the unique root of the united tree. The new root v now invokes Subroutine UPDATE separately in each of the two trees. When these invocations terminate, these terminations are detected at the new root. The new root v then sends its forest replica to the other endpoint u of the merging edge. The other endpoint u marks the edge, initializes $\text{Forest}_u(v)$, and sends its own forest replica to the root. The root then marks the edge too, and initializes $\text{Forest}_v(u)$. Next, the root starts looking for a new merge candidate by invoking UPDATE ***again*** and then FIND.

This process is repeated until a single tree spans the connected component of the network.

On the other hand, when a real tree edge (u, v) fails, its endpoint u unmarks the edge and removes the $\text{Forest}_u(v)$ associated with it (i.e., sets it to "uninitialized"). The endpoint then notifies the root that a topology change has occurred (the ALERT notification method is detailed in Section 4.2.4 below).

4.1.2 Subroutines: Search for the Minimum Outgoing Edge

Recall that a node discovers (Subsection 3.5) which of its adjacent edges is outgoing by consulting its tree replica to see which of its neighbors is not in its own tree. We now explain the distributed implementation of UPDATE, which maintains this replica. We also must explain how the nodes in a real tree compare their adjacent minimum outgoing edges to find the global minimum.

4.1.2.1 Let us start with Subroutine UPDATE. The root broadcasts an instruction to the real tree nodes to perform procedure LOCAL-UPDATE, i.e., each node which receives the broadcast forwards the received broadcast to its children on the tree. Next, the node performs procedure LOCAL-UPDATE. See Subsection 3.4, and the pseudo-code in Figures 2.1 and 2.2. The root detects that all the activations of LOCAL-UPDATE terminate, using the termination detection algorithm of [DS80]. Since the termination detection algorithm is quite known, we describe it in Appendix A.

4.1.2.2 As mentioned above, the tool used to find the minimum outgoing edge is exactly the one used in [GHS83]. Additional details can be found in Appendices A.2 and B. The adaptation to additional topology changes is described in Subsection 4.2. After FIND terminates at the root, the root calls Subroutine Root Migration (Section 4.1.3 below) to move the role of the root from itself to the endpoint of the minimum outgoing edge.

4.1.3 Subroutines: Root Migration

This distributed subroutine is initiated by a root of the real tree after the FIND it invokes finds an edge that is outgoing from the tree. It transfers the role of the root to the endpoint (in the real tree) of this outgoing edge. Note that the other endpoint of that edge is in another real tree, which may be busy transferring its own rootship to that other endpoint. (There are some other possible cases, e.g. that the rootship of the other real tree may already be waiting in the other endpoint; yet another case is that the other real tree is still busy merging with some other tree). Since, again, this migration is a known technique, it is described in Appendix A. When this subroutine terminates, the tree is directed towards the endpoint of the minimum outgoing edge, unless some additional topology changes occurred. The adaptation to additional topology changes is described in Subsection 4.2.

4.1.4 Agreement between Two Real Trees

The stage when two real trees agree to merge is different than the one used in [GHS83]. Had we used the mechanism of [GHS83] in a network with topology changes, the algorithm could have deadlocked. In the current algorithm, out of the two endpoints of the minimum outgoing Edge (u, v), only the one with the lower identity u is responsible for offering a connection. Let u be the lower endpoint of the minimum outgoing edge (u, v). As long as u is not a root, it does nothing. If u becomes a root as a result of the actions in 4.1.3.2, then it sends a REQUEST message over the minimum outgoing edge. When the REQUEST message is accepted at v, Node v may or may not be the root of its own real tree. The reception of the REQUEST over edge (v, u) is recorded at v even if it is not currently a root. When the tree of the higher endpoint v has chosen this edge too and transferred the rootship to its endpoint v, v then waits to receive such a REQUEST message from u, unless such a REQUEST has already been received. Consider the time that all the following 3 conditions hold at v: (1) v is a root, (2) a REQUEST message from u is recorded to have been received, and (3) (u, v) is the current selection of the minimum outgoing edge of the tree rooted at v. Root v then sends u an ACCEPT message agreeing to the merge offer. (It also erases the record that a REQUEST has been received.) A case where the connection will not take place even if a REQUEST message has been sent is described in Subsection 4.2 that deals with additional actions taken when topological changes occur.

Node v becomes the root of the merged tree. It first invokes a separate UPDATE invocation in each of the two merging trees (see 4.1.1). When these UPDATE invocations terminate at v, Node v marks Edge (u, v) as a real tree edge, and instructs u to do the same.

4.2 Additional Steps Taken When Topological Changes Occur

4.2.1 When the edge of a node to its parent fails, it sets its parent pointer to nil, and thus becomes a root (by definition). Thus, there is always a root. Like the previous

root, the new root starts by searching for an outgoing edge. If a search for a minimum edge is ongoing (including Subroutine UPDATE), then the new root waits until it detects the termination of this search. Then the root restarts the search. (For the termination detection, see Section 4.2.2.)

4.2.2 In a Broadcast and Echo (used in FIND and UPDATE), if a real tree edge between a parent and a child fails, then that edge leaves the tree. Thus, the child stops being a child, and the parent will not wait for its Echo before sending an Echo to its own parent (or before terminating, if this node is the root and thus has no parent). The child that lost its parent will not send an Echo message.

No additional step for FIND is needed for edge recovery, nor for the case where a non-real-tree edge fails. If an edge recovers during the execution of the Broadcast and Echo, then it is still not a part of the real tree, and thus does not participate in the search at all. As for a failed non- tree edge, such an edge is not used for the Broadcast and Echo mechanism. Hence, the failure cannot prevent the termination of FIND. The only bad case is the one when the failed edge is a minimum edge reported by a child (or an edge on the route to that minimum). We do not change FIND to take care of such a case. Thus, it may happen that the minimum edge reported by FIND has meanwhile failed. Recovering from such a case is done not by FIND, but rather by the ALERT messages mechanism, described in 4.2.4, 4.2.5.

4.2.3 If the minimum outgoing edge fails, then the root repeats the operation of finding (another) minimum outgoing edge. This also happens if the root, while migrating to the minimum outgoing edge, finds that the next edge on its route has failed.

4.2.4 If any node on the real tree notices any topology change, then it sends an ALERT message to the root. This enables the root to "know" that the minimum edge may have changed. The root needs to "know" that, since an agreement to merge is guaranteed to be reached only if the chosen edge is the minimum outgoing edge. An offer to merge with another real tree over a non-minimum edge may never be agreed upon by the other endpoint. However, thanks to the ALERT, if no agreement was reached, then a topological change is detected and the real tree then searches again for the new minimum outgoing edge.

4.2.5 When the root receives the ALERT message, it must first check whether an agreement with another (a higher root *Id*) real tree is being negotiated. This is the case when (1) the root has sent a REQUEST message offering a merge to another tree, and (2) it has not received an ACCEPT message from that other tree. If no such offer was made, then the root restarts a search for the minimum outgoing edge. That is, the root activates UPDATE again, then FIND, and, finally, migrates the root to the endpoint of the minimum outgoing edge reported by FIND (if such an edge exists).

Note that the case that the root both offered a merge and received an agreement is impossible, since in this case, it became the child of the other root and is no longer

a root. If this happens after the agreement, but before this previous root marked the merging edge, then this ALERT is ignored. (Anyhow, after the edge is marked the new root will invoke UPDATE and find).

4.2.6 However, if the root u did send a REQUEST to another (a higher root $Id\,v$) real tree and has *not* yet received an ACCEPT, then the root tries to check the status of the offer. It asks (using a CANCEL message) the higher Id endpoint (of the minimum outgoing edge) whether the latter has already sent an ACCEPT, thus agreeing to offer. If the higher Id endpoint did agree, by sending an ACCEPT then the higher endpoint need not answer the CANCEL question. In this case, the ACCEPT sent by the higher endpoint will eventually arrive (unless the edge fails, canceling the merge). If such an ACCEPT arrives at the lower Id endpoint, then the merge takes place, and the ALERT message is discarded. (Anyhow the new root is going to invoke another UPDATE.) Otherwise (when no ACCEPT message has been sent by the higher endpoint, e.g. because it is not a root), the higher endpoint erases the record that a REQUEST message has been received, and sends a reply (CANCELLED). The meaning of Message CANCELLED is a promise to the lower Id root u that u's last REQUEST is forgotten, and that u will not act upon it. The reception of the CANCELLED message frees the offering root u to restart the search for the current minimum outgoing edge.

We comment that this canceling mechanism is needed in order to prevent a deadlock. Since a topology change occurred, Edge (u, v) may no longer be the minimum outgoing one. Thus, v's tree may never choose it for merging. Therefore, u cannot afford to keep its own selection unchanged, since this would have caused a deadlock. On the other hand, it is unsafe for u to change its selection without consulting v. Had u canceled the merge offer over (u, v) without first consulting v, it may have happened that v already agreed to the merge offer. This problem is not limited to our particular edge agreement sub-protocol. It seems that any edge agreement protocol will have to deal with the fact that an edge must be either marked by both sides, or by none. This task is complicated by the topology changes that may require its abort. Details for the method by which edge agreement is performed are not given explicitly in [CCK88]. Hence, the agreement method described in this subsection (4.2.6) may be useful for other protocols, such as the one of [CCK88].

5 Correctness

We first prove several properties, assuming the strong loop freedom invariant. Later, we use these properties to prove the invariant itself through an induction on the sequence of events. From section 4.1, we can observe that the algorithm is constructed to work in an orderly fashion. It works in "iterations" (for each root) where each iteration conducts a single tree link merge.

In principle, an "iteration" is composed of an ordered sequence of subroutines, UPDATE, FIND, root-migration, tree-merging and UPDATE again. Link failures may

be accommodated during the "iteration" itself while the handling of new link recoveries are delayed until the end of the "iteration".

Our proof of correctness follows the natural flow of the iteration. We prove that if the strong loop-freedom invariant holds at the beginning of an iteration it continues to hold after the execution of each subroutine, until the end of the iteration, where subtrees merge. Link changes may impact the operation of the algorithm and are taken into account in the proof steps. Consequently, the sequence of lemmas tracks the flow of an algorithmic iteration. In order to facilitate the description, we also group the lemmas according to this logic.

Lemmas 5.1 and 5.2 show that the real forest is indeed a forest, consisting of rooted trees. Lemma 5.5 shows that a real tree in this forest has periods of stability (in some sense) even if changes continue to occur. This serves later to show that the root can organize the (stable) tree to take action by way of a Broadcast and Echo over the tree. This is the way UPDATE and FIND function.

Lemma 5.4 shows that Subroutine FIND indeed terminates, and that the termination is detected by the root. The next group of lemmas deals with the properties of UPDATE that are more complex than those of FIND. The easiest property to show (Lemma 5.6 and Corollary 5.7) is that the strong loop freedom invariant is not violated by the actions of Subroutine UPDATE. Lemma 5.18 (appearing much later) generalizes this claim to all the events, excluding edge marking events. To prove that edge markings do not violate the invariant (Lemma 5.19) we need to first establish several facts regarding Subroutine UPDATE.

Lemma 5.8 shows that the corrections regarding the state of an edge (x, y) flow from x and y toward the other nodes and not vice versa. Intuitively, this makes this information "more accurate", as is shown in later lemmas. This is also used to show the termination of UPDATE as follows. Lemma 5.9 shows that this "flow" is not deadlocked, since a node "knows" which information needs correction at its neighbor. The progress shown by that lemma is also used in Lemma 5.10 to prove that the number of messages sent by UPDATE is bounded. This facilitates the proof that UPDATE terminates, and moreover, that this termination is detected by the root. See Lemma 5.11.

Lemmas 5.12, 5.13, and 5.14 show that when UPDATE terminates, the tree replicas at all the nodes of the real tree are equal to the real tree, except, possibly, for missing "knowledge" of recent edge failures.

Lemmas 5.15, 5.16, and 5.17 show that the above correct collection of information leads to the merging of trees. Merging neither creates cycles in the forest nor breaks the invariant. Lemma 5.19 then sums up that strong loop freedom in every tree replicas' union is indeed an invariant of the algorithm. This leads to Theorem 5.20 that states that eventually a spanning tree spans the connected component of the network.

5.1 Additional Invariants

The following two lemmas establish that the "iteration" description is well defined. That is, at all times, there exist a forest, a unique tree for each node, and a root of each tree. This is important since the root directs the advances of the "iteration".

Lemma 5.1 *As long as the strong loop freedom invariant holds, the real forest is indeed a forest.*

Proof. By definition, each real tree edge appears in the tree replica of its endpoints, and thus, in the tree-replicas' union. Thus, as long as there is no cycle in this union (since the strong loop freedom invariant holds), there is no cycle in any real tree. □

Definition 5.1 *We say that Node u and Node v* agree on the direction *of their parent pointer if the following two conditions hold: (1) if the parent pointer of Node u points at its neighbor v, then the parent pointer of Node v does not point at u, and (2) if the parent pointer of Node v does not point at u and Edge (u, v) is marked as a real tree edge at u, then the parent pointer of u does point at v.*

Definition 5.2 *A* Root Transfer State *for a real tree: a global state where (1) this real tree has no root, (2) there is exactly one edge (u, v) in the real tree for which the parent pointer of Node u points at v, the parent pointer of Node v points at u, and there is a message BE-ROOT from u to v, and (3) the endpoints of every other edge in the real tree agree on the direction of their parent pointers.*

Lemma 5.2 *As long as the strong loop freedom invariance holds, the only global states in which a real tree does not have exactly one root are root transfer states, which eventually end.*

Proof. We prove by induction on the order of events that

1. Both endpoints of every edge of a real tree agree on the direction of the parent pointer, except during a root transfer state.
2. During a root transfer state, the endpoints of every real tree edge agree on the direction of the parent pointer, except for the edge carrying the message BE-ROOT.
3. There is never more than one root in a real tree.

4. There is no root in the real tree if and only if the real tree is not in a root transfer state.

5. A message BE-ROOT is carried only by one edge in a real tree, and if and only if the real tree is in a root transfer state.

The basis of the induction is the case that every node is a real tree by itself, and this claim holds trivially. Consider any event.

1. Consider all the cases that a node may assign a value to its parent pointer. The first case is when the edge to the parent fails and the node unmarks the edge, and sets the parent pointer to nil. Clearly, the induction claim continues to hold. (Recall, that a failed edge is not considered a real tree edge, even if one of its endpoint has not learned yet about the failure). The second case is when trees merge. Only the lower endpoint assigns a value to its parent pointer, and that pointer is made to point at the higher endpoint. Moreover, the assignment of the parent pointer at the lower endpoint is made at the same time the lower endpoint marks the edge (Note, that the lower endpoints marks the edges before the higher endpoint does). See Section 4.1.4. Hence, when the edge is marked, the endpoints agree on the direction of the parent pointers. Finally, the value of the parent pointer is changed in a root transfer (Section 4.1.3.2). By the induction hypothesis, before the root transfer action is taken, all the endpoints of the edges in the real tree agreed about the parent pointer direction, and there was exactly one root. Hence, by 4.1.3.2 this enters the real tree into a root transfer state, and the claim holds.

2. This part follows directly from the definition. (It also follows directly from Part 1, when recalling the only place in the algorithm that sends a BE-ROOT message.)

3. By Part 3 of the induction hypothesis, before the event there was at most one root. If the event was a failure of a tree edge, then consider first the case when the real tree was not in a root transfer state. By Part 1, there is exactly one node (in one of the resulting new real trees) that lost its parent. Since the other nodes did not loose their parents, they do not become roots. Now consider the case that the edge failed during a root transfer state. In each of the resulting new trees there is at most one node that lost its parent, by Part 2. Hence, there is at most one root in any of the resulting trees, by Part 4. The only other event that turns a non-root node into a root is the reception of a BE-ROOT message. This part now follows from Parts 5 and 4 of the induction claim.

4. This follows from Part 1.

5. Consider the time that a message BE-ROOT is sent. By Section 4.1.3.2 and by the induction hypothesis, the real tree enters a root transfer state. In this state there is no root. Only a root can send an additional BE-ROOT message. However, for a new root to be created, either the first BE-ROOT message must

have arrived first, or this edge fails (and the first BE-ROOT message is lost), or the new root is a node that lost its parent. In the first two cases, the first BE-ROOT message is no longer in transit when the second is sent. In the last case, by the induction hypothesis (Part 2) this new root is no longer in the same real tree where the first BE-ROOT message is.

□

5.2 Actions of FIND

Lemma 5.3 *From the time a root invokes Subroutine FIND in a real tree, and until the time the root detects the subroutine's termination, the only possible changes to the tree are the loss of edges (and nodes).*

Proof. Only a root marks edges as real tree edges, and the root does no markings from the time it activates an FIND until FIND terminates at the root (if it does). In Figure 1, the last line is the only case of edge marking. □

Lemma 5.4 *As long as the strong loop freedom invariant holds, Subroutine FIND always terminates, and the termination is detected by the real tree root.*

Proof. FIND sends messages only over real tree edges. By Lemma 5.2, these edges form a forest. We prove by induction (starting from the leaves and ending at the root) for every node u in this tree (at the time FIND is invoked) that either u leaves the tree, or the following holds:

- If u is not the root, then u receives the Echoes of the FIND from all its children, and then sends its Echo to its parent.
- If u is the root, then u receives the Echoes of the FIND from all its children. At that time, the root decides that this FIND terminates in its real tree.
- At the time u receives the Echoes of all its children, all its offspring in the real tree already sent their Echo messages, and no message of FIND is in transit in this subtree.

The basis of the induction (a leaf) is trivial. The induction step is the same as the proof for a static tree (see e.g. [Seg83]), because u cannot acquire new children during its execution of FIND, by Lemma 5.3. Specifically, Node u sends an Echo when receiving the Echoes of all its children. For those Echoes, the induction hypothesis holds, and thus, it holds for u too. □

Note that topology changes do not change this proof of the induction step: A recovery of an edge does not change the proof since this edge does not join the tree, by Lemma 5.5. This is also the case with the failure of an edge that does not belong to the real tree. Finally, if an edge connecting a parent u and its child v fails, then it stops being a real tree edge. Thus, u will not wait for v's Echo before u sends its own Echo, so the above proof of the inductive step still holds. For the sake of completeness, let us consider the case of a node v that lost its parent before it Echoed FIND. In this case, v itself becomes a root, and the induction proof holds for v's new real tree. That is, eventually, v receives the Echoes of all its children, detects the termination of FIND in its own real tree, and, indeed, at that time, FIND in v's current real tree has terminated.

5.3 Actions of UPDATE

Lemma 5.5 *From the time a root invokes Subroutine UPDATE in a real tree, and until the time the root detects the subroutine's termination, the only possible changes to the tree are the loss of edges (and nodes).*

Proof. The proof is the same as the proof of Lemma 5.3 □

The following lemma is crucial in proving the correctness of the algorithm. It demonstrates that adding edges to replicas during the execution of UPDATE cannot create cycles in the replica union and therefore, cannot violate the strong loop freedom invariant.

Lemma 5.6 *Consider an event t that is a part of an execution of UPDATE in a real tree T. Every edge that belongs to the tree-replicas' union of T just after t, also belongs to the tree-replicas' union of T just before t.*

Proof. We need to consider only addition of edges. An edge e is added by Subroutine UPDATE to the tree replica of a node v only by inserting an edge from the "Add" list of a DIFF message v receives from a neighbor u in the same real tree (see Figure 2.1). Note that this edge was already in the union by virtue of being in the "Add" list. Similarly, the edge is included by u in an "Add" list it sends only if it is already in u's tree replica. See Figures 1 and 2.1. Thus, this edge is in the tree-replicas' union of v's tree. Hence, the union of the tree replicas can change only by the addition of nodes to the real tree (implying the addition of replicas to the union, or of a marking of an edge by the root). This does not happen during the execution of UPDATE by Lemma 5.5. □

Corollary 5.7 *The strong loop freedom invariant is not violated by actions of Subroutine UPDATE.*

Note that an edge recovery does not change the replica of any node (until some later time when this edge may get marked and become a real tree edge as a result of additional actions of UPDATE, FIND, etc.)

Definition 5.3 *A u-sided edge according to a tree replica $tree_w^w$ (of Node w) relative to a pair of neighbors u and v: this is any edge (x, y) such that the route from x to v over $tree_w^w$ passes via u.*

Lemma 5.8 *As long as the strong loop freedom invariant holds, the following holds:*

- *Whenever a node u instructs its neighbor v to add an edge to v's tree replica, this indeed is a u-sided edge according to $tree_u^u$, and is not a v-sided edge according to $tree_v^v$.*
- *Whenever a node u instructs its neighbor v to delete an edge from v's tree replica, this is a u-sided edge according to $tree_v^v$ and is not a v-sided edge according to $tree_u^u$.*

Proof. Follows directly from the invariant, and from the computation of the content of DIFF in Figures 2.1, 2.2.

Lemma 5.9 *For any edge (u, v) of the real tree, as long as the strong loop freedom invariant holds, consider any non- faulty real tree edge (i, j). If (i, j) is u-sided then the following holds:*

- *Edge (i, j) appears in $tree_u^v(Forest_u(v))$ but not in $tree_v^v$, if and only if there is a message on the way from u to v instructing u to add this edge to $tree_v^v$.*
- *After $Forest_u(v)$ is initialized, Edge (i, j) does not appear in $tree_u^v(Forest_u(v))$ but appears in $tree_v^v$, if and only if there is a message on the way from u to v instructing u to remove this edge from $tree_v^v$.*

Proof. We prove by induction on the events during the run of the algorithm. The induction basis is the case when every node is alone in its own real tree and there are no edges in the real forest. The proof in this case is trivial.

First, consider all the possible changes in $\text{tree}_u^v(\text{Forest}_u(v))$. Recall that when an edge is being marked as a real edge, its endpoints exchange their entire forest replicas. Moreover, there is no old DIFF message on the edge. This is because DIFF messages are sent over tree edges only, and this edge has not been a tree edge until the marking. Indeed, it may have been a tree edge in the past. However, the only way to get unmarked for an edge is to fail. At that time, all the messages over it are lost. So

no DIFF message exists over it when the edge is marked again. Thus, at the time u becomes a real tree neighbor of v, the claim holds.

Later, u sends a DIFF to v if and only if Node u also updates $\text{tree}_u^v(\text{Forest}_u(v))$, see the computation of the DIFF messages in Figures 2.1, 2.2. Thus, the claim continues to hold after such an update.

The second relevant case is when v changes tree_v^v. If this is the result of a DIFF message from u, then the proof follows from the induction hypothesis for the previous case. If this change in tree_v^v results from a message DIFF from another neighbor of v, then the change involves a non-u-sided edge, by Lemma 5.8. This is also the case if v marks, or unmarks, some other Edge (v, w) as a real tree edge. Thus, the claim is not violated. □

Note that, by the code of LOCAL-UPDATE (see Figure 2.1), Node v always follows the update instruction.

Lemma 5.10 *As long as the strong loop freedom invariant holds, during the execution of UPDATE in a real tree, an Edge (u, v) may be added at most once to the tree replica of each node.*

Proof. Assume the contrary. Let s be the first node in which some edge (u, v) is added twice to its tree replica during the same run of UPDATE. Let t_1 be the first time when this happened, and let l be the node that instructed s (by a DIFF message) to perform this addition. Let $t_3 > t_1$ be the second time when node s adds (u, v) to its tree replica, and let p be the node that instructed s to perform the addition this time. Thus, there exists some time t_2, such that $t_3 > t_2 > t_1$, when s deletes Edge (u, v) from its tree replica.

First, we prove that $p \neq l$. Assume the contrary, and note that $p = l$ had (u, v) in p's tree replica from the time it sent M_1, the DIFF message received at s at t_1, to the time p sent M_3, the DIFF message received at s at time t_3. Otherwise, this would have contradicted the definition of s as the first node that performed two such additions during that invocation of UPDATE. For that, recall that in order to instruct s to add (u, v), Node p must have (u, v) in tree_p^p and not have it in $tree_p^s(Forest_P(s)$; see Figures 2.1 and 2.2.

Moreover, when p sent M_1, it also inserted Edge (u, v) into $\text{tree}_p^s(\text{Forest}_p(s))$. (See Figure 2.1, the last line of each of the *Whenever* statements.) Also, p does not delete (u, v) from $\text{tree}_p^s(\text{Forest}_p(s))$, by Lemma 5.9 and since we proved that p did not remove (u, v) from p's tree replica.

Hence, Edge (u, v) is in $tree_p^s$ from t_1 to t_3. By the computation of the returned value Add (see figure 3) (and by Lemma 5.9), Edge (u, v) will not be included in an Add list p may generate in the time interval $[t_1, t_2]$. However, by Figure 2.1 (the first

"Whenever" clause), such an Add list is the only case that p sends instructions to s to add edges to s's replica. Hence, p could not have sent M_3. This contradicts the assumption that $p = l$.

Now, consider the remaining case that $p \neq l$. By the Tree Belief Principle (the computation of Add in Figures 2.1 and 2.2), when p instructed s to add Edge (u, v), p was, according to its tree replica, closer (on the tree) than s is to u. This also holds for l at time t_3. By Lemma 5.6, this means that there exist two routes over the tree-replicas' union from u to s at time t_1 (one via p, and the other via l). This contradicts the assumption that the strong loop freedom invariant holds. □

Lemma 5.11 *As long as the strong loop freedom invariant holds, Subroutine UPDATE always terminates, and the termination is detected by the real tree root.*

Proof. A node that receives the UPDATE broadcast starts the LOCAL-UPDATE process, in which (1) it instructs neighbors to add edges to their tree replicas or to remove edges from them, and (2) it gets such instructions from its neighbors. Assuming that this process ends, the proof that UPDATE terminates and its termination detection succeeds is the same as the proof of Lemma 5.4. It is left to show that the LOCAL-UPDATEs terminate, but this follows from Lemma 5.10. □

Recall that a tree replicas' union of a real tree may include edges (say (x, y)) that are not in the real tree. This happens, for example, when (1) some edge (even (x, y) itself) connecting x to the real tree failed, and (2) the failure has not yet been reported to some node in the real tree. Note that the node that is closest to (x, y) on the tree replicas' union no longer has (x, y) in its tree replica, but it may take some more time until the edge disappears from the replicas' union. This gives rise to the following definition.

Definition 5.4 *Consider a real tree and its tree replicas' union. Consider a maximal connected component of this union, such that no edge of the component is in the real tree. We say that this component is* trimmed *from the real tree at Node u and Edge (u, w) if Edge (u, w) is in this component, but u is in the real tree. When we refer to the component, we call it "the subtree trimmed at edge (u, v)".*

The following lemma states that UPDATE makes the tree replicas identical (in nodes in the same real tree) with the exception of trimming. Intuitively, trimming is an exception since it could have occurred after UPDATE terminated at a node.

Lemma 5.12 *As long as the strong loop freedom invariant holds, whenever UPDATE terminates in a real tree, for every two neighbors u, v in the real tree the following holds:*

If an edge (x, y) appears in $tree_v^v$, and does not appear in $tree_u^u$ then Edge (x, y) belongs to a subtree trimmed at (u, w) for some $w \neq v$. Moreover, the trimming happened after UPDATE terminated at u.

Proof. It is easy to see that the termination of UPDATE is not detected at the root of the real tree before every node in the real tree receives the UPDATE's broadcast and starts its LOCAL-UPDATE. By Lemma 5.11, UPDATE does not terminate before all the LOCAL-UPDATEs in the real tree terminate. Thus, LOCAL-UPDATE is performed at every node in the real tree before UPDATE terminates. It is left to prove that the LOCAL-UPDATEs reach a state for which the lemma holds.

By the strong loop freedom invariant, every edge with at least one endpoint in the subgraph $\text{tree}_v^v \bigcup \text{tree}_u^u$ is either a u-sided edge or a v-sided edge, but not both. First, consider a u-sided Edge (x, y) that is in tree_u^u and not in tree_v^v. By the invariant, this edge is not a v-sided one, and v does not instruct u to remove it (see the calculation of the list Del in Figure 2.2). Moreover, it is not a k-sided edge relative to v and any of its neighbors $k \neq u$. Thus, v does not get any instructions from any such neighbor k regarding Edge (x, y) (see, again, the calculation of the list Del in Figure 2.2). By Lemma 5.9, as long as UPDATE has not terminated in u, $\text{tree}_u^v(\text{Forest}_u(v))$ includes (x, y) if and only if it is also added to tree_v^v. This shows that UPDATE causes tree_u^u and tree_v^v to agree on every u-sided edge in tree_u^u.

The proof for a u-sided edge not in $tree_u^u$ is similar, as long as UPDATE has not terminated yet at u. The only change to a replica that can take place after UPDATE has terminated in Node u, is for an edge (u, w) of u itself that was in the real tree and now fails. (Node u does not use UPDATE yet; instead it sends an ALERT message to the root asking it to start UPDATE; see Section 4.2.4). In this case, u removes the edge from tree_u^u, but does not update v. Note that this may lead to a discrepancy between u and v only regarding edges in the subtree trimmed at Node u and Edge (u, w), so the lemma continues to hold.

□

Lemma 5.13 *As long as the strong loop freedom invariant holds, the tree replica at each node u when UPDATE terminates, is a subset of the real tree to which u belonged when this invocation of UPDATE started.*

Proof. Assume, by way of contradiction, that when UPDATE terminates, there exists an edge (x, y) in u's tree replica such that (x, y) did not belong to the real tree of u when UPDATE was invoked. Let $M = \{u = u_0, u_1, u_2, ...u_m = x, u_{m+1} = y\}$ be the route (in u's tree replica at the termination of UPDATE) that starts in u, passes via x, and ends in y. Let (x', y') be the first edge on M that does not belong to the real tree at the time UPDATE was invoked. (Note that x' is its endpoint in u's real tree.) Clearly, the assumption that (x, y) exists implies the existence of such an Edge (x', y'). We show that such an edge (x', y') does not exist.

Let $x'' = u_q$ be the closest node to x' on M, that is still in $u's$ real tree when UPDATE terminates (possibly, $x' = x''$). We prove by induction on $q - j$, the distance of a node from x'' on M, that when Node u_{q-j} terminates its part in UPDATE, the following holds:

(*) Edge (x', y') does not appear in $\text{tree}_{u_{q-j}}^{u_{q-j}}$.

Note that by Lemma 5.12 the following holds:

(**) all the edges on M between $x'' = u_q$ and u_{q-j} do appear in $\text{tree}_{u_{q-j}}^{u_{q-j}}$.

The induction hypothesis holds for $j = 0$ by the definitions of x'' and (x', y') and by the assumption that the strong loop freedom invariant holds. Assume that the hypothesis holds for $j - 1$. Hence, there is a time during the execution of UPDATE from which (and up to the termination of UPDATE, inclusive) (*) holds for $x'' = u_q, u_{q-1}, ...u_{q-j-1}$. Consider that time interval.

By Lemma 5.9, if any edge on M between $x = u_q$ and $u_{q-(j-1)}$ does not appear in $\text{tree}_{u_{q-j}}^{u_{q-j}}$, it (eventually) does not appear in $\text{tree}_{q-(j-1)}(\text{Forest}_{q-(j-1)}(u_{q-j})$. By the definition of M, the assumption that the strong loop freedom invariant hold, and (**), such an edge is a $u_{q-(j-1)}$ sided edge relative to u_{q-j}. Thus, $u_{q-(j-1)}$ instructs u_{q-j} to add it (see the computation of the list Add in figure 3). Node u_{q-j} does indeed add it (see figure 2). Similarly, Node $u_{q-(j-1)}$ also instructs u_{q-j} to remove (x'', y'') if it appears in the tree replica of u_{q-j} (see the computation of the list Del in Figure 2.2). This is because this edge is $u_{j-(1-1)}$- sided (by (**) and by the strong loop freedom invariant). This edge is indeed removed (see Figure 2.1). By similar arguments, Edge (x'', y'') is not added again to the tree replica of u_{q-j} in the same invocation of UPDATE. Thus, neither Edge (x', y') nor Edge (x'', y'') are in tree_u^u when UPDATE terminates. □

Note that Edge (x', y') in the proof of the above lemma may still appear in the forest replica of u when UPDATE terminates, though not in u's tree replica. Edge (x'', y'') was removed from the forest replica too.

Lemma 5.14 *As long as the strong loop freedom invariant holds, the tree replica at each node u when UPDATE terminates is a superset of the real tree to which u belonged at that time.*

Proof. Follows from Lemma 5.12. □

In the following lemma we show that trees merge, if there is a "sufficiently long" interval of time during which no topology changes occur. The main point is to show that there is no deadlock. The algorithm prevents a case in which one tree insists on merging over one edge, while the other insists on merging over the other. Such a problem does not arise in static networks, see [GHS83]. However, this could have happened in a dynamic network, had we not taken steps against it.

5.4 Actions of Tree Merging

Lemma 5.15 *Assume the topology changes stop, and the strong loop freedom invariant holds. Consider the minimum edge (u, v) that connects two real trees. Eventually, one of these real trees merges with another real tree.*

Proof. We first outline the proof: (1) Edge (u, v) is not chosen by its real tree only if this real tree chose another outgoing Edge (x, y), and (2) in this case, either the real tree merges over Edge (x, y), and the lemma holds, or the real tree performs another UPDATE and FIND, in which it selects (u, v). This happens in both the real tree of u and the real tree of v. Thus, these two trees merge together, unless one of them merges with yet another tree.

Detailed proof: By the code (restarting whenever topology changes occur, see Figure 1, the second *Whenever* statement, and Section 4.2.4 in the distributed implementation) and by Lemmas 5.4 and 5.11, as well as Lemmas 5.13 and 5.14, the real tree to which u belongs does find an outgoing edge if such exists. This can be either Edge (u, v) or some other edge (x, y) whose weight must be higher (since (u, v) is minimal). Moreover, by Lemma 5.4 and the fact that FIND compares values on a tree, the later case can happen only if (u, v) recovered after u already sent the Echo of FIND to its parent. In this case, the recovery of (u, v) causes an ALERT message to be sent to the root, prompting it to start a new round of UPDATE and FIND (See 4.2.4). Thus, either the real tree abandons the selection of (x, y) and selects (u, v) as its minimum outgoing edge, or it already is involved in a merger.

That is, the root will insist on the selection of (x, y) only if it is x and it is in the following state: It already sent a REQUEST message to y, and has not received an ACCEPT message. See 4.2.6, 4.2.3, 4.2.1. The other cases are easier, but let us mention them first for the sake of completeness: (1) Had x received an ACCEPT message, it is either about to complete a merge, and the lemma holds, or is no longer a root, which contradicts the claim that the root received the ALERT. (2) Alternatively, had the root received a REQUEST message and already sent an ACCEPT message, it would have completed the merger by now, and then found Edge (u, v) in the next iteration of the algorithm. Recall that a new root runs UPDATE again. See Figure 2.1.

We now show that no deadlock occurs also in the remaining case that a root x has already sent a REQUEST and is waiting for an ACCEPT when it receives an ALERT. Root x sends a CANCEL message to y. See 4.2.6. Recall that we assume here that the topology changes stopped; thus this message arrives. (No deadlock occurs here even if Edge (x, y) did fail, since then x would have performed another round of UPDATE and FIND; see 4.2.4, 4.2.3, 4.2.1). If this message arrives when y has not yet sent an ACCEPT to x, then y sends a CANCELLED message (see 4.2.6), and root x is free to perform another round of UPDATE and FIND (see 4.2.4, 4.2.3), in which it will find Edge (u, v) or the other edge that is the minimum outgoing at that time. (If no such edge exists, then the lemma holds trivially.) If, on the other hand, the CANCEL message of x arrives at y when y already sent an ACCEPT message to x, then the two trees merge, and the lemma holds. See 4.2.6. □

Observation 5.16 *Assume that the strong loop freedom invariant holds up to the point that two real trees merge. Then, no cycle is created in the real forest by the merge.*

Proof. By Lemmas 5.1 and 5.2, the real forest is indeed a forest of rooted trees until the merge. Moreover, real trees do not add any edges except by merging.

For two trees to merge, the root of one of them sends a REQUEST message, and the root of the other agrees and sends an ACCEPT message. See 4.1.4. Each of these trees has only one root (Lemma 5.2). The sender of the REQUEST message sends neither an ACCEPT nor any other message before either one of the following 2 events occur:

1. The REQUEST message is answered: in this case the sender of this REQUEST stops being a root, and this real tree will not merge with any other real tree. Thus, a cycle is not created. See Section 4.1.4.

2. The sender of the REQUEST sends a CANCEL message and gets a reply that approves the cancellation of the merger (CANCELLED message). In the second case, the merger is canceled, and cannot lead to a cycle. See 4.2.6.

□

We are now ready to establish the fact that strong loop freedom is an invariant of the algorithm. The proof is based heavily on the remarkable action of the algorithm, namely, the fact that it executes UPDATE twice: before a merge (without FIND) and after a merge (with FIND), for two different reasons. The proof of the following lemma explains the reason for executing UPDATE before a merge.

Lemma 5.17 *The merging of trees does not violate the strong loop freedom invariant.*

Proof. By Lemma 5.1, the set of edges of two merging trees is disjointed at the time they choose an edge to merge. By Observation 5.16, no cycle is created in the merged real tree. Hence, these two real trees do not share any node (before adding the merging edge). Recall (Figure 1) that after choosing the merging edge, but before marking it, UPDATE is performed in each of the two merging trees separately. By Lemma 5.13, after these UPDATEs, the tree replica of any node v in either one of the two merging real trees is a subset of v's real tree. It is left to show that this replica does not contain nodes from the other real tree. However, this is clear from the fact that this tree replica is a subset of a set (the real tree) that does not contain nodes from the other real tree.
□

5.5 Theorem

Lemma 5.18 *For each event during the execution of the algorithm, except for a "mark" event (see the last code line, Figure 1), the following holds: If the strong loop freedom invariant holds before the event, then it holds after the event as well.*

Proof. The only events of the algorithm that change tree replicas at all are as follows:

- An edge failure may remove edges from tree replicas (if this is a real edge). This, however, cannot create cycles.
- A "mark" event adds a real tree edge to the tree replicas of both its endpoints. Note (from the text of the lemma) that this lemma does not make any claim about this event.
- The reception of a DIFF in UPDATE may insert (or delete) edges to the tree replica of the recipient. This cannot create cycles by Corollary 5.7. □

Lemma 5.19 *Strong loop freedom is an invariant for the algorithm.*

Proof. Follows from Lemmas 5.18 and 5.17. □

Theorem 5.20 *If the topology changes stop, then, eventually, every connected component is spanned by one real tree.*

Proof. Follows from lemmas 5.15 and 5.19. □

6 Complexity

In this section, we analyze the *message complexity* of the algorithm. Recall that the number of bits per message is logarithmic in $|ID|$, which is the size of the name space of the nodes. Typically, $\log |ID| = O(\log V)$. This message complexity is one of the main contributions of this paper. We do not analyze here the time complexity which does not compete with those of [AAG87] or [KP99] (though it is still polynomial).

Intuitively, we show that every node is notified about every topological change a bounded number of times (actually once, but this is somewhat harder to show). We rely on the structure of the forest, analyzed in the previous section. Recall that each real tree is rather stable (except for, possibly, loosing edges) for the period it performs UPDATE. Thus, the notification of the changes flow over trees, and, thus, cannot cycle. We also make sure this flow is completed before the merge. This prevents the possibility that an old update "in transit" will arrive at a Node u from the side of its old tree, while the same information will arrive from the side of the other tree with which u's tree has now merged. Hence, and because of the strong loop freedom invariant, when two trees merge, the information about each edge flows only in the correct direction, *from* the edge *to* other nodes. This ensures not only the correction of the information, but also that the information flow does not cycle. We also keep the invariant that a node "knows" what its tree neighbor "thinks". Hence, a node is not told what it already "knows". This ensures that the amortized cost of notification about a topology update is $O(V)$. Finally, we show how to attribute the work performed for each merge

to a recent topological change. Since we manage to keep the overhead per change to $O(V)$, the cost of merging is also $O(V)$ per topological change.

Lemmas 5.8 and 5.9 show the cases that DIFF messages are sent. Lemma 6.1 shows that the tree replicas indeed stabilize before a merge, in the sense that each node already "heard" of every edge insertion to its real tree. Recall that edge failures can occur at any time. Thus a node may not be aware of deletions that occurred recently. Lemmas 6.2, 6.3, and 6.5 show that such deletions of which the node is unaware are indeed recent. The above lemmas are used to show in Lemma 6.6 that every node updates its tree replica a constant number of times per item. To conclude the analysis it is shown (in Theorem 6.7) that every merge can be attributed to a recent topological change. The proof of the theorem includes also a count of the rest of the messages sent.

Definition 6.1 *We say that* a merging edge is agreed upon between two trees *when (1) one of them has sent a REQUEST message over it, (2) the other has responded with an ACCEPT message, and (3) the edge has not failed in the interval between the two events.*

Definition 6.2 *Let a* pre-merge update *be a run of UPDATE invoked when a merging edge was already agreed upon between two trees, and UPDATE is executed in each of them separately (see Figure 1, the third* Whenever *statement, first line). Let a* post-merge update *be the other kind of UPDATE (see Figure 1, the second* Whenever *statement).*

If a topology change occurs during UPDATE, causing UPDATE to be executed again, we refer to the combined execution as one execution.

When talking of a post-merge update, we would like to be able to talk separately about the nodes that were in each of the merging real trees before this merge.

Definition 6.3 *Let u be a node that participates in an invocation of UPDATE. Let T_u be the set of nodes that participated with u both in its previous UPDATE invocation and the current one. Let $\bar{T}_u$ be the set of nodes that participate with u in its current UPDATE invocation, but not in the previous one (note that this is possible only in a post-merge update).*

Lemma 6.1

- *No node adds an edge to its tree replica during a pre-merge UPDATE.*
- *If some node x is added to a tree replica of a node u during a post-merge UPDATE, then x does not belong to T_u.*

Proof. Every node that belongs to the real tree of some node u during any pre-merge update, already belonged to u's real tree in the previous UPDATE. The lemma now follows from Lemma 5.12. Similarly, every node that belongs to T_u during a post-merge UPDATE, already belonged to u's real tree in its previous UPDATE, and the lemma follows from Lemma 5.14. □

The following lemma is used to show that changes in the data structures can be attributed to topological changes that occurred recently.

Lemma 6.2 *If Edge (u, v) is deleted from the forest replica of a node x during an UPDATE, and it was in the tree replica of x when this UPDATE was invoked, then Edge (u, v) failed after the beginning of the previous invocation of UPDATE in the real tree to which x belongs.*

Proof. Either this is a post-merge UPDATE and Edge (u, v) is the merging edge, or Edge (u, v) must have been in x's tree replica when the previous invocation of UPDATE terminated at x. This is because there are only two ways in the algorithm for an edge to join x's tree replica – either (1) if $x = u$, by having x mark Edge (x, v), or (2) by an UPDATE. Obviously, the lemma holds in the first case. Let us consider the second case.

By Lemma 5.13, Edge (u, v) was in x's real tree when that previous UPDATE was invoked. We show that it indeed failed later. Let $w \in T_x$ be the first node in T_x to either delete Edge (u, v) from w's tree replica, or send a DIFF message instructing another node to delete that edge. By Lemma 5.12, either (a) Node w had Edge (u, v) in its tree replica at the beginning of the current UPDATE, or (b) (u, v) belongs to a subtree that was trimmed at w before the beginning of the current UPDATE.

Consider case (a). By the strong loop freedom invariant, Node w cannot receive a DIFF message from a node not in T_x, instructing w to delete (u, v) from w's tree replica. In addition, w did not receive a DIFF message from another node in T_w instructing it to delete (u, v) by the choice of w. The only other case in the algorithm when w may delete the edge is that $w = u$, and w unmarked Edge $(u, v) = (w, v)$ after the termination at w of the previous invocation of UPDATE. Note that an edge is unmarked only when it fails (Figure 1).

Now consider case (b). Similarly to case (a), Node w did not delete Edge (u, v) as a result of a DIFF message. In addition, note that w may send a DIFF message that includes an edge in a subtree trimmed at w only if $w = u$ (see figures 2 and 3). Hence, this edge was unmarked, which implies that it failed.

We showed that in both cases the edge was in the real tree when the previous UPDATE was invoked, and failed after that. □

Lemma 6.2 showed that a deleted edge failed recently, but only under a certain condition that is eased in the next lemma.

Lemma 6.3 *If Edge (u, v) is deleted from the tree replica of a Node x during an UPDATE, and Node u was in the tree replica of x just before this UPDATE was invoked, then Edge (u, v) failed after the beginning of the previous invocation of UPDATE in the real tree to which x belonged.*

Proof. If Edge (u, v) was in x's tree replica when the current UPDATE was invoked, then the lemma follows from Lemma 6.2. Otherwise, since u was in x's tree replica at that time, Edge (u, v) was not in the forest replica of x (otherwise it would have belonged to the tree replica too). Consider a scenario that caused the *insertion* of Edge (u, v) to x's tree replica in the current UPDATE, so that it can be later deleted. If $u = x$, recall that an edge of u joins u's tree replica only as a result of Merge (the marking of an edge, the last line in Figure 1). An edge of u cannot be inserted to u's tree replica during UPDATE, since this edge is u-sided, relative to every neighbor of u. (See the way the list Add is computed in Figure 2.2.) This means that Edge (u, v) was in the tree replicas' union at the beginning of the UPDATE, and the lemma follows from lemma 6.2.

We are left with the case that $u \neq x$. In this case, since (u, v) is not in the tree replica of x at the beginning of the current UPDATE, it was not in the replica of x at the termination of the previous UPDATE. (Outside of the execution of an UPDATE, an edge can leave a tree replica when it belongs to a trimmed subtree, and then leaves only the replica of the node at which the subtree is trimmed; see lemma 5.12.)

Hence, by Lemma 5.14, Edge (u, v) was not in the real tree of x at the termination of the previous UPDATE in x's real tree.

Moreover, by Lemma 5.12 (and since $x \neq u$), Edge (u, v) was not in the tree replica of any nodes in x's real tree when the previous UPDATE (at x's tree) terminated. Consider $w \in T_x$ that is the first of them to either insert Edge (u, v) or send an instruction to insert Edge (u, v) or both. See Figures 2.1 and 2.2. By Lemma 6.1, there are only two cases:

Case (1) (a) Node w first received such an instruction from a node not in T_x, and (b) this is a post-merge UPDATE.

Case (2) (a) $w = u$, and (b) this is a post-merge UPDATE, and (c) Edge (u, v) is the edge used for the merge.

In case (1), first we claim that u is in T_x (and thus in T_w). Note that when a post-merge UPDATE is invoked, no tree replica in T_x includes a node in $\bar{T}_u$, by Lemma 5.17, except for the replica of the root. The root inserts the merging edge to its replica at the same operation it also invokes UPDATE (See Figure 1, and the atomicity of events handling in the model, Section 1.1). Hence, at the time stated in the lemma (just before this operation), the tree replicas of nodes in T_x do not contain nodes in $\bar{T}_x$.

Hence, u is in not in $\bar{T}_x$. However, this means that case (1) contradicts the strong loop freedom invariant (which is guaranteed by Lemma 5.19). Hence, case (1) is impossible.

In the second case, note that Edge (u, v) was non-faulty when this UPDATE was started by Root u. This means that Edge (u, v) was not faulty after the previous invocation of UPDATE. The deletion time of Edge (u, v) is even later than that. Recall that in case (2) we assume that (u, v) is a real tree edge. Since a real tree edge is deleted by its endpoint only if it fails, it is left to prove that its endpoint did delete it during the current UPDATE in order to show that Edge (u, v) failed during the current UPDATE. The lemma now follows from the lemma's assumption that the edge was deleted from the replica of Node x during the current UPDATE, and from Lemma 5.12 (since $x \neq u$). □

Let us comment on the first case in the proof. We used the fact that the root inserts the merging edge to its replica in the same atomic operation of initiating the post-merge UPDATE. This is used just for the convenience of the proof. Without it, we would have needed to consider an additional case, where u is the root of the other merging tree. Had we needed to prove for this case too, the proof would have been very similar to the proof of the second case.

The next lemma is similar to the previous two, except that the condition is eased further. Before the lemma, we need an observation.

Observation 6.4 *Consider an Edge (u, v) that is deleted from the* forest replica *of a node x during some UPDATE invocation U in x's real tree. Let U_2 be the last UPDATE in x's real tree during which Edge (u, v) belonged also to x's* tree replica *(in addition to belonging to x's forest replica). Possibly $U_2 = U$. We claim that such a U_2 exists.*

Proof. Recall that the tree replica of Node x contains initially only x itself. Moreover, it is not difficult to see from LOCAL-UPDATE that an edge is never added to the forest replica without having been added to the tree replica. □

Lemma 6.5 *Consider an Edge (u, v) that is deleted from the* forest replica *of a node x during some UPDATE invocation U in x's real tree. Let U_2 be the last UPDATE in x's real tree during which Edge (u, v) belonged also to x's* tree replica *(in addition to belonging to x's forest replica). Possibly $U_2 = U$. Then, Edge (u, v) failed after the beginning of the UPDATE preceding U_2 in x's real tree.*

Proof. Assume, by way of contradiction, that the lemma does not hold. We prove that in this case, one of the endpoints, either u or v appears in the tree replica of x when U is invoked, implying the lemma by Lemma 6.3.

Assume, further, that neither u nor v are not in the tree replica of x just before U is invoked (otherwise, again, the lemma follows from Lemma 6.3). In particular, $x \neq u$,

$x \neq v$. So, the only place in the algorithm when x may delete (u, v) is when x receives a DIFF message that includes (u, v) in the Del list (see Figures 2.1 and 2.2). We prove that x does not receive such a message during U under the above assumptions.

The proof proceeds by a case study according to the type of U: either a pre-merge UPDATE or a post- merge UPDATE. Each case is further subdivided into two: the subcase that either u or v appears in the tree replica of other nodes in T_x, and the subcase that no endpoint of (u, v) appears in the tree replica of any node in T_x.

- U is a pre-merge UPDATE:

– **No endpoint of (u, v) appears in the tree replica of any node in T_x:**
By Lemma 5.6 this remains the case throughout U. By the algorithm (Figures 2.1 and 2.2), no node sends a DIFF message that includes an edge of u.

– **Of of the endpoints (say, u) appears in the tree replica of some node $y \in T_x$:**
By Lemma 5.12, Node u belongs to a subtree that was trimmed in x after the termination in x of the UPDATE that preceded U. By Lemma 5.19, at that time, for every real- tree- neighbor k of x, node u is not k-sided with respect to Edge (k, x). This remains the status until U, by Lemma 5.17. Moreover, u cannot become k-sided during U, by Lemma 5.6. Hence, no node sends a DIFF message to x that includes an edge of u.

- U is a post-merge UPDATE:

– **Nodes u and v do not appear in the tree replica of any node in T_x:**

We prove that neither x, nor any other node in T_x receive a DIFF message that includes (u, v) in the Del list. Assume the contrary and let x' be the node closest to $\bar{T}_x$ among the nodes of T_x that received such a DIFF message.

Let $p \in \bar{T}_x$ be the closest node to x' on the merged real tree such that an endpoint of (u, v) appears in the tree replica of p at the beginning of U. Such a node p exists, otherwise no node sends a DIFF message that includes an edge of either u or v (by Lemma 5.6, see also the computation of DIFF in Figures 2.1 and 2.2).

By Lemma 5.12, either

- (1) (u, v) belongs to a subtree that was trimmed at the endpoint b of the merging edge after the UPDATE that preceded U terminated in b;
or
- (2) not case (1) and $p \in \bar{T}_x$ is an endpoint, b, of the merging edge.

In case (1), the endpoint of (u, v) appearing in p's replica is b-sided with respect to every neighbor k of b. By Lemma 5.6 and the strong loop freedom invariant, no such k sends DIFF messages including an edge of u to b. This means that b does not send such a message either if b is not the trimming point.

In the subcase that b is the trimming point, this trimming could not have happened at Edge (u, v) itself, since otherwise, the lemma holds. Hence, in this case too, b never sends any DIFF messages that include any edge of u.

Recall that the only node in $\bar{T_x}$ that can send messages to nodes in T_x is Node b. Hence, Nodes u and v cannot enter the tree replica of any node of T_x. Recall that these nodes are also not in the tree replica of any node in T_x at the beginning of U. This contradicts the assumption that Node x' receives a DIFF message that includes an edge of u.

The remaining case is case (2) where it is assumed that $p \in \bar{T_x}$ is the endpoint of the merging edge, and at least one of the endpoints of (u, v) (say Node u) appears in the tree replica of p at the beginning of U. This means that u was in the real tree of p at the time that the UPDATE preceding U was invoked in p's real tree, by Lemma 5.13. If Edge (u, v) does not appear in the tree replica of p at the end of the UPDATE preceding U, then the lemma holds by Lemma 5.14. tree replica of p at the beginning of U.

Consider the path $M = \{p = l_0, l_1, l_2, ...l_{q+1} = x'\}$ connecting p to x' over the real tree at the beginning of U. At each time during U, this path can be partitioned into a prefix (initially, containing only node $l_0 = p$) and a suffix (at the beginning of U, containing all the rest of the nodes on the path). The idea is that every node l_i of the prefix has a tree replica that includes (u, v), as well as all the path up to l_i. Moreover, the prefix of M ending with l_i is the suffix of the path (over l_i's tree replica) to u. No node in the suffix of M contains u in its tree replica.

We show by induction, that such a path $M(t)$ exists at any time t throughout the execution of U. Consider any event that may affect $M(t)$, and consider the real tree to which x' belongs after the event. If the event is a failure of an edge between nodes in $M(t)$, this results with a path $M(t^+)$ with the same properties (but, possibly, either the prefix or the suffix is empty). The only other events that may influence $M(t)$ are the receptions of DIFF messages by nodes on the path. Moreover, only a DIFF message from l_i may affect the claimed properties of the path at l_{i+1}, for $i \geq 0$, by Lemmas 5.6, 5.19 and by the code (Figures 2.1 and 2.2). If this is a DIFF message from a node in the suffix to another node in the suffix, the claimed properties of the path are not affected, since the sender does not have u in its tree replica. If it is a message from a node in the prefix to another node in the prefix, it does not affect the claimed properties of the path, since the tree replica of the sender includes (u, v), as well as the route to it. Finally, if there is a message from a node l_i in the prefix to a node l_{i+1} then l_{i+1} joins the prefix by Lemma 5.9 and Figures 2.1,2.2.

Hence, the only way that a path $M(t)$ looses its claimed properties is that (u, v) ceases to be in p's tree replica. A deletion of (u, v) by p cannot be the event that causes this to happen, since in this case, the lemma follows from Lemma 6.3. Hence, u leaves p's tree replica because it belongs to a trimmed subtree. Either this trimming happens at p (but $p \neq u$, otherwise the lemma holds), or the trimming is reported to p in a DIFF message, causing u to leave p's tree replica. In either way, p no longer can send DIFF messages that delete an edge

of u. By Lemma 5.19, Lemma 5.6 and the calculation of DIFF in Figures 2.1 and 2.2, no other node on the path can send such a DIFF message either. A contradiction.

– **Node u appears in the tree replica of some node $y \in T_x$:**
This subcase is the same as the second subcase for a pre- merge UPDATE. □

Let a *change* in an edge (u, v) be either a failure, or a marking as a tree edge. An *update* in a tree replica is either the insertion to the tree replica, or a deletion from it.

Lemma 6.6 *Every node x updates its tree replica at most a constant number of times per change in edge (u, v)*

Proof. By Lemma 6.5, every deletion corresponds to a failure that happened since the last invocation of UPDATE in x's real tree that updated (u, v) in x's tree replica. By Lemma 5.10, this can happen only a constant number of times per such UPDATE. An edge cannot be reinserted unless deleted first. Hence, the number of insertions is bounded by the number of deletions. The lemma follows. □

Now consider an execution of UPDATE where (u, v) already exists in x's replica (and is not deleted again). We claim that no additional insert (ADD) messages containing (u, v) are sent to x. This follows from Lemma 5.9.

Actually, Edge (u, v) is updated at most once in x's tree replica per topology change. Since we only counted the order of magnitude of the message complexity, the above lemma suffices.

Theorem 6.7 *If k topological changes occurred, the number of messages (of $O(\log Id)$ bits each) is $O(kn)$.*

Proof. Let us first count the number of *Id*s exchanged between the two endpoints of an edge that is being marked as a real tree edge. By Lemmas 5.1 and 5.19, the real forest is indeed a forest at all times. Thus, the marking of a new tree edge by the root at its endpoints joins together two trees. Thus, for k topology changes, at most k tree mergers can occur. In each merger, each endpoint sends the other endpoint its forest replica, this is $O(n)$ *Id*s.

Next, one ACCEPT message is sent per merger, to the total of k messages. A REQUEST message may be followed by an ACCEPT message and lead to a merger. The number of such REQUEST messages is also $O(k)$. One CANCEL message may also be sent before the merger actually takes place. In this case, the number of such CANCEL messages is bounded by the number of REQUEST messages, and this is $O(k)$.

Alternatively, a REQUEST may be followed by a CANCEL message, and a CANCELLED reply, leading to the cancellation of the planned merger. This happens when the root of the real tree receives an ALERT message. By the algorithm (see 4.2.4), such a message is initiated by a node, in the real tree, which noticed a topology change. By Lemmas 5.1, 5.2 and 5.19, by the FIFO discipline on the links, and by the fact that every merger is followed by several Broadcasts and Echoes (of FIND, and of UPDATE), such ALERT messages are forwarded on a tree, and hence do not cycle. Thus, such a cancellation corresponds to a topology change. Moreover, by the same arguments, this topology change occurred after the last FIND invocation started in the real tree. Since each time a merger is canceled, the root performs a new round of UPDATE and FIND, each cancellation corresponds to a different topology change. (We count a change in an edge twice—once per endpoint.) Thus, the number of CANCEL and CANCELLED messages is $O(k)$. Similarly, the number of ALERT messages is $O(kn)$.

As shown in the previous paragraphs, FIND is performed on a tree, once per topology change. Thus, the number of messages is $O(kn)$. The argument for BE-ROOT messages is similar. This is also the case with UPDATE messages, except for the messages used by LOCAL-UPDATE. The theorem now follows from Lemma 6.6. □

Theorem 1.1 now follows from Theorems 5.20 and 6.7.

We analyzed above the complexity per edge- topological- change. Note that a node failure or recovery can be modeled as a failure (or a recovery) of all of its edges. In addition, a failing node may loose its memory (besides loosing its edges). Hence, when it recovers, it may receive updates that insert previously known edges to its Forest. Since Forest contains $O(|V|)$ entries, this may add an $O(|V|)$ to the complexity of the recovery of a node. This can be charged to the recovery of the first recovered edge of that node. Note, that as long as the recovering node has no edges, it is not connected to the rest of the nodes, and we do not consider it recovered.

7 Conclusion

Let us highlight the main new technique we used in order to save communication. Consider the case that a real tree is broken into two real trees. Should the nodes in one of them forget the structure of the other? If they do, and the two real trees reunite, then each of the $O(n)$ nodes in one tree must be informed of the $O(n)$ edges of the other real tree. This may lead to $O(n^2)$ communication. On the other hand, while T_1 is disconnected from the other real tree T_2, the latter may change, or even be erased completely. What should the nodes of T_1 tell a node w that joins T_1 about T_2? Anything they tell may be incorrect. In fact, the information w has may be more up to date than the information possessed by nodes in T_1. Thus, it seems that they should forget this information. Indeed, as explained in the introduction, one of the reasons for the high communication complexity of previous solutions is sending wrong information, that has to be corrected later.

The major saving in communication is obtained mostly by the new technique of sending implied information. That is, a node learns of the existence of an edge by the fact it was not instructed to remove it. To see that, let us continue Example 3.1. There, Node "left" sends Node "right" only the corrections to the tree replica of "right", rather than the whole tree replica of "left" (See Drawing 9d). Since "left" did not instruct "right" to remove Edges (s,t), (t,p), Node "right" can "deduce" that these edges are in the tree. Based on this deduction, Node "right" now instructs its neighbor w to add these edges.

The edge used here to merge the tree is edge (u,s), and it is now the responsibility of u to inform its side of the tree (Nodes "left", "right" and w) of the structure s's tree. This information eventually reaches all of u's tree, though not every piece of it travels explicitly over every edge. Only the identity of Edge (u,s) traveled to "left", and then to "right". Then the identities of Edges $(u,v),(s,t)$, and (t,p) traveled to Node w. In general, u's tree may be much larger, and every node of it may have a different view of the real tree of Node s. Every node of it learns every edge in the tree of s. However, different nodes in u may learn only implicitly about different edges of the real tree of s.

The preliminary version of this paper inspired follow-up work. In particular, [ACKMP98, AS97] dealt with the issue above, namely, broadcasting when different intended recipients may have different parts of the broadcast information. The methods suggested in these papers (for a small part of the task of Subroutine UPDATE) are not as good as UPDATE in terms of message efficiency, but are better in terms of time complexity. This gives rise to the hope that the time complexity of our algorithm can be improved significantly. A method to improve the time complexity to some degree was suggested in [KP99], but in general the issue of time improvement is still open. Another contribution of [KP99] is a proposal for defining the "current" size of the network, so that the complexity can be defined in terms of the "current" size rather than some maximum size, or a predefined size. Note, that in an asynchronous network, it is not clear what is a realistic definition of the current size for that purpose. For example, events that happen far from some node v (or even at other connected components) may change the size, but can have no impact on v's behavior (at least, until some time later). At this point, it is not clear that the definition in [KP99] is the "best" (for example, it is not clear that no distributed algorithm can estimate the size better). We chose not to adopt that definition in this journal version.

In [BO99] a topology maintenance algorithm is suggested that is similar to that of [ACK90] in the sense that the broadcast trees are built using the topology information that is being broadcast over these trees. However, the paper of [BO99] uses multiple broadcast trees (one per source). Moreover, these are Reverse Path Forwarding trees, as opposed to our tree that is arbitrary. Such trees are close to minimum hop trees, and have the potential of reducing the broadcast time. It is claimed in [BO99], based on simulation, that the protocol presented there reduces the number of topology update messages compared to those of flooding based protocols, especially in common cases when most recipients are leaves. It seems clear that in a worst case, the number of messages sent by the protocol of [BO99] is much higher than $O(V)$ per change,

as opposed to the current paper that obtains $O(V)$ message complexity per change. Other algorithms that are more scalable than flooding were proposed later for topology update, e.g. in [Gar93, JJ94, OTL]. Their complexity seem to be higher (in the worst case) than the complexity of the algorithm given here). Algorithms were proposed also for various related dynamic tasks such as updating shortest paths, e.g. [CSFN03, RV92, Hum91, IR98, Ita91, BBL04]. Some of the above algorithms were for different models (e.g., the sequential model, or a distributed algorithm for an environment where no edge may fail during the execution). Others had a higher message complexity (but provided more information to the nodes, e.g., the whole topology).

The idea of tree maintenance was utilized in a later implementation of a fast link-state based topology update algorithm used for fast connection reservation and QoS supported routing in [CHKPRS99]. This implementation disseminates to all nodes hardware forwarded messages over a multicast tree providing a unified up to date link state and link utilization information.

A major body of later work on dynamic networks was performed (after the publication of the conference version of this paper) in the context of self stabilization. For example, in [DIM93] the maintenance of a tree when a leader is known is described, and [AKY90] and several later papers describe tree maintenance even when no leader is known (similarly to the current paper). Similarly, many self stabilizing reset protocols were suggested, starting from [AKY90, AG94, APV91]. The message complexity of the current algorithm cannot be compared to self stabilizing network algorithms since the latter need to send messages indefinitely, even when no events occur. Algorithms for maintaining other structures besides trees in a distributed environment (also without translating the algorithm from one for static environments) were also suggested, see, e.g., [E07].

Another contribution of this paper is the method to avoid deadlock when joining trees together. Note that the method used by [GHS83] for static networks, may fail in dynamic networks. In [GHS83] it is assured that some two (real) trees will choose the minimum edge connecting them. Here, at different times, different edges may be the minimum. Because of the asynchronous nature of the network, one real tree may be already aware of a new minimum, while the other real tree may commit to an old minimum, causing a deadlock. Note, that at some point, a real tree must commit, since an edge must either be marked at both sides, or not marked at all.

We note that if our algorithm and the MST algorithm of [GHS83] are both used to construct a spanning tree in a static network where initially every edge is up but no edge is yet marked as a tree edge, the complexity of algorithm of [GHS83] is $O(E + VlogV)$ messages, while for the algorithm presented we only proved $O(V^2)$ messages. Intuitively, this is because of the "levels" mechanism of the algorithm of [GHS83]. Each level uses up $O(V)$ messages to add multiple edges to the tree (at least $V/2$ for the first level). Our algorithm spends $\Omega(V)$ messages for every tree edge. We do that because, in a dynamic case, we must respond on-line to topological change as we do not know when these changes cease. This is a different case than the static one of [GHS83] where all the edges are known to their endpoints in advance.

Finally, one main motivation for the maintenance of a spanning tree is to use it for broadcasting topology updates, whenever there is a topology change. Such a mechanism (that does not use unbounded counters) was presented in [ACK90].

Acknowledgments

We thank Avner Porat and the anonymous reviewers for their helpful comments.

References

[A85] Baruch Awerbuch. Complexity of network synchronization. Journal of the ACM (JACM), Volume 32, Issue 4 (October 1985), pages: 804 - 823, 1985.

[AAG87] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. Proceedings of the Twenty-Eighth IEEE FOCS Conference, pages 358–370, October 1987.

[AAM89] Yehuda Afek, Baruch Awerbuch, and Hezi Moriel. Overhead of resetting a communication protocol is independent of the size of the network. Unpublished manuscript, May 1989.

[ACK90] Baruch Awerbuch, Israel Cidon, Shay Kutten. Communication-Optimal Maintenance of Replicated Information. Proceedings of IEEE FOCS 1990: 492-502.

[ACG+90] Baruch Awerbuch, Israel Cidon, Inder Gopal, Marc Kaplan, and Shay Kutten. Distributed control for PARIS. Proceedings of the ninth annual ACM symposium on Principles of distributed computing Quebec City, Quebec, Canada, August 1990, pages 145–160, 1990.

[AGKK] Joshua Auerbach , Madan Gopal , Marc Kaplan , Shay Kutten. Multicast group membership management. IEEE/ACM Transactions on Networking (TON), Vol.11 No.1, p.166-175, February 2003.

[AG88] Yehuda Afek and Eli Gafni. End-to-end communication in unreliable networks. Proceedings of ACM PODC pages 131–148, 1988.

[AAG+] Yehuda Afek, Baruch Awerbuch, Eli Gafni, Yishay Mansour, and Adi Rosen, Nir Shavit. Slide-The Key to Polynomial End-to-End Communication. J. Algorithms 22(1): 158-186 (1997)

[AG94] Anish Arora and Mohamed G. Gouda. Distributed Reset. IEEE Trans. Computers 43(9): 1026-1038 (1994).

[AG91] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization. Proceedings of PODC 1991, pages 295–307, 1991.

[AGH90] Baruch Awerbuch, Oded Goldreich, and Amir Herzberg. A quantitative approach to dynamic networks. Proceedings of ACM PODC, 1990, pages 189-203.

[AGPV90] Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A tradeoff between information and communication in broadcast protocols. J. ACM 37(2): 238-256 (1990).

[AGR92] Yehuda Afek, Eli Gafni, Adi Rosen. The Slide Mechanism with Applications in Dynamic Networks. Proceedings of ACM PODC 1992, pp. 35-46.

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-Efficient Self Stabilizing Protocols for General Networks. Proceedings of WDAG 1990:15-28.

[AM86] Baruch Awerbuch and Silvio Micali. Dynamic deadlock resolution protocols. Proceedings of IEEE FOCS 1986: 196-207.

[AMS89] Baruch Awerbuch, Yishay Mansour, and Nir Shavit. End-to-end communication with polynomial overhead. Proceedings of IEEE FOCS 1989, pages 358–363, 1989.

[APV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-Stabilization By Local Checking and Correction FOCS 1991: 268-277.

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. Proceedings of FOCS 1988, pages 206–220, October 1988.

[Awe88] Baruch Awerbuch. On the effects of feedback in dynamic network protocols. J. Algorithms 11(3): 342-373 (1990).

[BO99] Bhargav Bellur and Richard G. Ogier. A Reliable, Efficient Topology Broadcast Protocol for Dynamic Networks. Proceedings of IEEE INFOCOM 1999.

[BGJ+85] A. E. Baratz, J. P. Gray, P. E. Green Jr., J. M. Jaffe, and D.P. Pozefski. SNA networks of small systems. *IEEE Journal on Selected Areas in Communications*, SAC-3(3):416–426, May 1985.

[AS97] Baruch Awerbuch and, Leonard J. Schulman. The maintenance of common data in a distributed system. J. ACM 44(1): 86-103 (1997)

[ACKMP98] Baruch Awerbuch, Israel Cidon, Shay Kutten, Yishay Mansour, and David Peleg: Optimal Broadcast with Partial Knowledge. SIAM J. Comput. 28(2): 511-524 (1998)

[BBL04] Marc Bui, Franck Butelle, and Christian Lavault. A distributed algorithm for constructing a minimum diameter spanning tree. J. Parallel Distrib. Comput. 64(5): 571-577 (2004).

[BS04] David Barnes and Basir Sakandar. Cisco LAN Switching Fundamentals. Cisco Press, ISBN: 1587050897; Published: Jul 15, 2004.

[CCK88] C. Cheng, I.A. Cimet, and Srikanta P.R. Kumar. A Protocol to Maintain a Minimum Spanning Tree in Dynamic Topology. Proceedings of ACM SIGCOMM 1988 Symposium, Stanford, California, United States August 16 - 18, 1988, pp. 330 - 337.

[CHKPRS99] I. Cidon, T. Hsiao, A. Khamisy, A. Parekh, R. Rom, and M. Sidi. OPENET: an open and efficient control platform for ATM networks Journal of High Speed Networks, 8(3): 195-210 (1999)

[CSFN03] S. Cicerone, G. D. Stefano, D. Frigione, and U. Nanni. A fully dynamic algorithm for distributed shortest paths. Theoretical Computer Science, 297:83?102, 2003.

[CG88] I. Cidon and I. S. Gopal. PARIS: An approach to integrated high-speed private networks. *International Journal of Digital & Analog Cabled Systems*, 1(2):77–86, April-June 1988.

[CGK88] Israel Cidon, Inder Gopal, and Shay Kutten. New models and algorithms for future networks. New models and algorithms for future networks. IEEE Transactions on Information Theory, 41(3):769 - 780, May 1995.

[CGKK95] I. Cidon, I. Gopal, M. Kaplan, and S. Kutten. A Distributed Control Architecture of High-Speed Networks. IEEE Transactions on Communications, vol. 43, no. 5, May 1995, pp. 1950 - 1960.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. 3(1): 63-75 (1985).

[CRKG89] Chunhsiang Cheng, Ralph Riley, Srikanta P.R. Kumar, and Jose J. Garcia-Luna-Aceves. A loop-free extended Bellman-Ford routing protocol without bouncing effect. ACM SIGCOMM '89, pages 224–237, September 1989.

[CS88] Reuven Cohen and Adrian Segall. A distributed query protocol for high-speed networks. In *Proceedings of the 9th International Conference on Computer Communication, ICCC88*, Tel Aviv, October-November 1988, pages 299-302.

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. Distributed Computing 7(1): 3-16 (1993)

[DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[E07] Michael Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, Portland, Oregon: 185 - 194, 2007.

[Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[Fin79] Steven G. Finn. Resynch procedures and a fail-safe network protocol. COM-27(6):840–845, June 1979.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one family faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Fre83] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees. SIAM J. comput. 14:781–798, 1985.

[GA87] Eli Gafni and Yehuda Afek. Local fail-safe resynch procedure. unpublished manuscript, April 1987.

[Gaf87] Eli Gafni. Topology resynchronization: A new paradigm for fault tolerance in distributed algorithms. In Proceedings of the 2nd Workshop on Distributed Algorithms (WDAG'87), Amsterdam, The Netherlands, July 8-10, 1987, July 1987, Proceedings: Lecture Notes in Computer Science 312, Springer 1988.

[Gal76] Robert G. Gallager. A shortest path routing algorithm with automatic resynch. Technical report, MIT LIDS, March 1976.

[Gal77] Robert G. Gallager. A minimum delay routing algorithm using distributed computation. IEEE Transactions on Communications, Vol 25, Issue 1: 73-85, 1997.

[Gar89] Jose J. Garcia-Luna-Aceves. A unified approach to loop-free routing using distance vectors or link states. ACM SIGCOMM Computer Communication Review archive Vol. 19, Issue 4, pages: 212 - 223.

[Gar93] J.J. Garcia-Lunes-Aceves . Loop-free routing using diffusing computations. IEEE/ACM Transactions on Networking, Vol. 1, No. 1, Feb 1993: 130 - 141.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 5 , Issue 1 (January 1983), Pages: 66 - 77.

[HS89] P. Humbet and S. Soloway. Topology Broadcast Algorithms. Computer Networks and ISDN Systems, North-Holland, 16 (1988/89), pp. 179-186.

[IETF] http://www.ietf.org/proceedings/02nov/179.htm.

[KP99] Shay Kutten and Avner Porat. Maintenance of a Spanning Tree in Dynamic Networks. Proceedings of the 13th International Symposium on Distributed Computing (DISC'99), Bratislava, Slovak Republic, September 1999, pp. 342-355.

[GRSV03] R. Guerin, J. Rank, S. Sarkar, and E. Vergetis. Forming Connected Topologies in Bluetooth Adhoc Networks. Proceedings of ITC'18, Berlin, September 2003.

[Hum81] Pierre A. Humblet. An adaptive distributed Dijkstra shortest path algorithm. Technical Report CICS-P-60, Center for Intelligent Control Systems, MIT, May 1981.

[Hum91] P. Humblet. Another adaptive distributed shortest path algorithm. IEEE Transactions on Communications, 39(6):995?1003, 1991.

[Ita91] Giuseppe F. Italiano. Distributed Algorithms for Updating Shortest Paths. Proceedings of the 5th International Workshop on Distributed Algorithms: 200 - 211, 1991.

[IR98] G.F. Italiano and R. Ramaswami. Maintaining Spanning Trees of Small Diameter. Algorithmica 22 no.3: 275-304.

[JJ94] Jochen Behrens, J. J. Garcia-Luna-Aceves. Distributed, scalable routing based on link-state vectors. Proceedings of the conference on Communications architectures, London, United Kingdom: 136 - 147, 1994.

[JM82] Jeff Jaffe and Frank Moss. A responsive distributed routing protocol. COM-30(7, Part II):1758–1762, July 1982.

[KM86] Ephraim Korach and M. Markovitz. Algorithms for distributed spanning tree construction in dynamic networks. Technical Report 401, Dept of CS, Technion, Haifa, Israel, February 1986.

[KMZ89] Ephraim Korach, Shlomo Moran, Shmuel Zaks. Optimal Lower Bounds for Some Distributed Algorithms for a Complete Network of Processors. Theor. Comput. Sci. 64(1): 125-132 (1989).

[MICG95] G.A. Marin, C.P. Immanuel, P.F. Chimento, and I.S. Gopal. Overview of the NBBS architecture - Networking BroadBand Services. IBM Systems Journal, Volume 34, Number 4, Page 564, Dec, 1995.

[MRR80] John McQuillan, Ira Richer, and Eric Rosen. The new routing algorithm for the arpanet. IEEE Transactions on Communications 28 (5):711–719, May 1980.

[MS79] P. Merlin and A. Segall. Failsafe distributed routing protocol. IEEE Trans. Comm. 27:1280-1287, September 1979.

[OTL] R. Ogier, F. Templin, and M. Lewis. RFC3684: Topology Dissemination Based on Reverse-Path Forwarding (TBRPF). an Internet RFC: http://portal.acm.org/citation.cfm?id=RFC3684.

[P99] Radia Perlman. Interconnections: Bridges, Routers, Switches, and Internetworking Protocols, 2nd Edition. Addison Wesley Professional, 1999.

[RF89] Balasubramanian Rajagopalan and Michael Faiman. A new responsive distributed shortest path routing algorithm. pages Proceedings of the ACM Symposium on Communications architectures & protocols, Austin, Texas, pages 237–246, September 1989.

[RV92] K. V. S. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. Journal of Algorithms, 13:235 ? 257, 1992.

[Seg83] Adrian Segall. Distributed network protocols. IEEE Transactions on Information Systems, Vol IT-29(1):23–35, January 1983. Some details in technical report of same name, MIT Lab. for Info. and Decision Syst., LIDS-P-1015; Technion Dept. EE, Publ. 414, July 1981.

[SG89] John M. Spinelli and Robert G. Gallager. Broadcasting topology information in computer networks. IEEE Trans. on Commun., May 1989.

[SS81] A. Segall and M. Sidi. A failsafe distributed protocol for minimum delay routing. IEEE Trans. on Commun., Vol COM-29(5):689–695, May 1981.

[T02] Andrew Tanenbaum. Computer Networks, 4th Edition. Prentice Hall, 2002.

[Tur88] J. S. Turner. Design of a broadcast packet switching network. IEEE Transactions on Communications, vol. 36, no. 6, pp. 734-743, June 1988.

[Vis83] U. Vishkin. A distributed orientation algorithm. IEEE Trans. on Information Theory IT-29, 4 (July 1983), 624-629.

A Appendix: details about subroutines that use known techniques

A.1 Appendix: details about the distributed termination detection (Section 4.1.2.1)

Let us describe that termination detection algorithm briefly. That algorithm was designed for a computation that starts in a single node, which joins the computation following some signal from the outside. Every other node joins the computation after receiving a message from a node that has already joined the computation. In our case, the root of the real tree is "signaled" to start the computation of UPDATE when the main algorithm of Figure 1 at the root reaches the point when the root is to start the execution of UPDATE. Every non-root node v in the real tree joins the computation (of UPDATE) upon receiving the broadcast message of UPDATE from some other node u already in the computation (of UPDATE), that instructs them to join the computation. (We view the messages of LOCAL-UPDATE as a part of the messages of UPDATE.) Let us term u the *predecessor* of v.

The main idea is that of a Broadcast and Echo search that appears also in e.g. [BGJ$^+$85, Seg83, AM86, GHS83]. That is, a node u that receives a message (of that computation) from a node v "owes" v an acknowledgement message (not a part of the underlying computation, this is an "extra" messages added for the sake of the termination detection). A node pays what it owes immediately, except for one of the messages it owes to its predecessor. (That is, if v owes its predecessor more than one acknowledgement, it sends all these acknowledgements but one.) Node v pays the last acknowledgement to its predecessor only when nobody owes v anything. At that time, v leaves the computation. It is then possible that v joins the computation again if it receives another message of the same computation (in a general application of [DS80] it is possible that at that time, v has a predecessor that is not the same as the predecessor v had when it joined the same computation for the first time.) The root detects the termination when nobody owes it any acknowledgement. For more details, a formal description and proofs please see [DS80].

A.2 Appendix: Details about subroutine FIND (Section 4.1.2.2)

As mentioned above, the tool used to find the minimum outgoing edge is exactly the one used in [GHS83]. There it uses messages called "find" and "found". This tool also became a standard one. It is called, sometimes, Broadcast and Echo search, e.g. [BGJ$^+$85, DS80, Seg83, AM86]. Note that the usage of this tool is based on the assumption that each node already "knows" which of its adjacent edges is outgoing. The method for realizing this assumption in [GHS83] is different than the one used here, described above in Section 4.1.2.1. However, given the assumption, finding the

minimum outgoing edge of the entire real tree is exactly the same. For the sake of completeness we describe this subtask in a little more detail. The Broadcast part is similar to the one explained for Subroutine UPDATE. In the Echo part of this subroutine, each node waits for the reports from all its children. (A leaf does not need to wait.) Next, the node compares the minimum weight reported by its tree children, and the weight of the lightest of its own outgoing edges (if such exist). The minimum between the two is reported (in an Echo message) back to its parent. The search terminates when the root receives an Echo from all of its children, and compares their minimum to its own lightest outgoing edge. The minimum edge reported to the root is selected to be the minimum outgoing edge of the real tree.

A.3 Appendix: Details about subroutine ROOT-MIGRATION (Section 4.1.3)

4.1.3.1 The following technique too is taken from [GHS83]. Consider again the search for the minimum outgoing edge (see 4.1.2.2). The following is used in order to establish a route from the root to the endpoint of the minimum outgoing edge. Consider a node that is going to send an Echo report to its parent with the weight of the minimum outgoing edge among those that are either adjacent to it or were reported by its children. If this minimum was reported by a child, then the node also remembers a pointer to this child. The collection of these pointers is a route from the root to the endpoint of the minimum outgoing edge. Thus, the root can migrate along these pointers to the endpoint of the minimum edge.

4.1.3.2 This subsection completes that parts copied from [GHS83]. Recall that a root r is a node with no parent. Thus, to transfer the rootship to one of its children, u, the root (1) sets its parent pointer to point at u, and (2) sends a message BE-ROOT to u over Edge (r, u) telling u to become a root. When u receives this message, it sets its own parent point to nil, thus becoming a root. This process is repeated until the root is the endpoint of the minimum outgoing edge, or until the root is made aware of a topological change (see Section 4.2).

B Appendix: Code for node v

The algorithm of node v

A signal is always given to self and received from self.
If there are more than one event to handle, then a topology change has highest priority and an ALERT message is second in priority. The priority of the other events is lower.

Initially, Parent = Chosen-Edge = nil, Tree = Forest = Min-Edges = $\emptyset$.
No edge is marked, every flag is false.

Main program (distributed implementation)

Whenever message ALERT is received from a child or a topology change occurs in an edge of v or signal Edge-Marked (* is received from self *)
(* The subroutines too perform actions for these events, see code for subroutines. *)
If the event was the failure of a marked edge (v, u) Then
unmark (v, u); (*unmarking takes the edge out of Forest$_v$ *)
If Parent = u Then Parent $\leftarrow$ nil;
If the event was a topology change or an ALERT message
If Alert-Pending = false Then Alert-Pending $\leftarrow$ true; (* Notifying root of event *)
If Parent $\neq$ nil Then send message ALERT to Parent;
Else If Handshake-Pending And Chosen-Edge$\in$ Tree$_v(v)$ (* REQUEST sent, not over (v, u) *) Or Update-Pending Or Find-Pending
Then Alert-Pending$\leftarrow$ true;
Else Signal Perform-Update.

Whenever signal Update-Before-Find-Terminated
Signal Perform-Find;

Whenever signal Find-Terminated
If Alert-Pending
Then Signal Perform-Update;
Else Signal Transfer-Root((x, y)).

Whenever signal Transfer-Terminated((x, y)) (* $v = x$ *)
(*edge (x, y) is not faulty, otherwise would have encountered ALERT*)
(*topo change has highest priority, ALERT second, others less*)
Signal Merge-Trees((v, y)).

Subroutine Tree merging handshake

Whenever signal Merge-Trees((v, y))
  Handshake-Pending $\leftarrow$ true;
  Chosen-Edge $\leftarrow (v, y)$;
  If $v < y$ Then send message REQUEST to y;
  Else If Request-Received(y) Then
    Request-Received(y) $\leftarrow$ false;
    Send message ACCEPT to y;
    Update-During-Handshake $\leftarrow$ true;
    Signal Perform-Update.

Whenever signal Update-During-Handshake-Terminated
  Update-During-Handshake $\leftarrow$ false;
  If Chosen-Edge $= (v, y) \neq$ nil (* otherwise may have failed meanwhile *)
  Then
    If $v < y$ Then
      mark (v, y); (* Insert to Forest$_v$ *)
      Parent $= y$;
      Send message New-Child to Parent;
      Chosen-Edge $\leftarrow$ nil;
      Handshake-Pending $\leftarrow$ false.
    Else If New-Child-Received(y) Then
          mark (v, y);
          Chosen-Edge $\leftarrow$ nil;
          New-Child-Received(y) $\leftarrow$ false;
          Signal Edge-Marked;
          Handshake-Pending $\leftarrow$ false.
        Else Update-During-Handshake-Terminated $\leftarrow$ true.

Whenever message New-Child is received from k
  If Chosen-Edge $= (v, k)$ Then
    If Update-During-Handshake-Terminated Then
      mark (v, y);
      Chosen-Edge $\leftarrow$ nil;
      Update-During-Handshake-Terminated $\leftarrow$ false;
      Signal Edge-Marked;
      Handshake-Pending $\leftarrow$ false.
    Else New-Child-Received(k) $\leftarrow$ true.

Whenever message REQUEST is received from a neighbor k
  If Parent $\neq$ nil or Chosen-Edge $\neq (v, k)$ Then Request-Received(k) $\leftarrow$ true;
  Else (* root is waiting to (v, k) *)
    Send message ACCEPT to k;
    Update-During-Handshake $\leftarrow$ true;
    Signal Perform-Update.

Whenever message ACCEPT is received from neighbor k
  Update-During-Handshake $\leftarrow$ true;
  Signal Perform-Update.

Whenever message CANCEL is received from neighbor k
  If edge (v, k) is not marked and $\neg$ Chosen-Edge $= (v, k)$ Then
    Send message CANCELLED to k;
    Request-Received$((v, k)) \leftarrow$ false.

Whenever message ALERT is received from a child or a topology change occurs in an edge of v or signal Edge-Marked
  (* Main and the other subroutines also perform actions for these events, see there. *)
  If the event was the failure of an unmarked edge (v, u) Then
    Request-Received$(u) \leftarrow$ false;
    If Chosen-Edge$= (v, u)$ then
      Chosen-Edge$\leftarrow$ nil;
      Handshake-Pending $\leftarrow$ false;
      Update-During-Handshake$\leftarrow$ nil.
      Update-During-Handshake-Terminated$\leftarrow$ nil;
      New-Child-Received$(u) \leftarrow$ false;
  Else If Parent = nil Then
      If there exists an edge (v, w), $w \neq u$, such that Request-Sent$((v, w))$
      Then send message CANCEL to w;

Whenever message CANCELLED is received from k
  If Chosen-Edge $= (v, k)$ then Chosen-Edge $\leftarrow$ nil;
  Handshake-Pending $\leftarrow$ false.
  (* Main will also signal Perform-Update *)

UPDATE, with INTERRUPTIONS

Whenever message Perform-Update is received from Parent Or signal Perform-Update
  If $\neg$ Update-Pending Then
    Update-Pending $\leftarrow$ true;
    Alert-Pending $\leftarrow$ false
    For every child u (* $(v, u) \in$ Tree And $u \neq$ Parent *)
      Send message Perform-Update to u;
      Update-Ack$(u) \leftarrow$ false.
    Invoke LOCAL-UPDATE; (*with termination detection, see Section 4.1.2.1 *)
    Local-Termination $\leftarrow$ false.
  Else Alert-Pending $\leftarrow$ true. (* Perform another Update when the current one ends *)

Whenever the termination of LOCAL-UPDATE is detected
  Local-Termination $\leftarrow$ true;
  If for every child w Update-Ack(w) Then perform procedure Report-Update.

Whenever message Update-Echo is received from a child u
  Update-Ack(u) $\leftarrow$ true;
  If for every child w Update-Ack(w) And Local-Termination
  Then perform Procedure Report-Update.

Procedure Report-Update
  Local Termination $\leftarrow$ false;
  Update-Pending $\leftarrow$ false;
  For every child w, Update-Ack(w) $\leftarrow$ false;
  If Parent $\neq$ nil Then send message Update-Echo to Parent;
  Else if Alert-Pending then Signal Perform-Update
      Else If Update-During-Handshake then Signal Update-During-Handshake-Terminated
          Else Signal Update-Before-Find-Terminated;

FIND, with interruptions

Whenever message Perform-Find from Parent is received from Parent or signal Perform-Find
  Find-Pending $\leftarrow$ true;
  Min-Edges $\leftarrow \emptyset$;
  For every child u
    Send message Perform-Find to u;
    Find-Ack(u) $\leftarrow$ false.
  If v has no children Then perform procedure Report-Find.

Whenever message Find-Echo(edge (x, y)) is received from child u
  Min-Edges $\leftarrow$ Min-Edges $\bigcup \{(x, y)\}$;
  Find-Ack(u) $\leftarrow$ true;
  If for every child w Find-Ack(w) Then perform procedure Report-Find.

Procedure Report-Find
  Find-Pending $\leftarrow$ false;
  Let $(x, y) = \min(*weight*)(\{(v, w) |\ \nexists (w, z) \in$ Tree $\} \bigcup$ Min-Edges);
  If Parent $\neq$ nil Then send message Find-Echo((x, y)) to Parent;
  Else Signal Find-Terminated((x, y));

Rootship Transfer with interruptions

Whenever Transfer-Root((x, y)) message is received from Parent or Transfer- Root((x, y)) signal
(* If next edge on route to x failed then the topo change treatment *)
(* in Main already removed that edge from Tree, since a topo change *)
(* event is handled with a higher priority *)
If Alert-Pending Or $\nexists k$ such that $(v, k) \in$ Tree and $(x, y) \notin$
My-Side(k) Or ($x = v$ And (x, y) is $faulty$)
Then Signal Perform-Update (* restart Main *)
Else If $x = v$ Then
Parent $\leftarrow$ nil;
Signal Transfer-Terminated.
Else
let k be such that $(v, k) \in$ Tree And $(x, y) \notin$ My-Side(k);
Send message Transfer-Root((x, y)) to k;
Parent $\leftarrow k$.