Optimal Maintenance of Replicated Information

Baruch Awerbuch[*]
Department of Mathematics and
Laboratory for Computer Science
MIT, Cambridge, MA 02139
baruch@theory.lcs.mit.edu

Israel Cidon
IBM T.J. Watson Research Center,
P.O. Box 704 Yorktown Heights, NY 10598
cidon@ibm.com

Shay Kutten
IBM T.J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598
kutten@ibm.com

May 27, 1993

Abstract

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

In this paper we show that keeping track of history enables significant improvements in the communication complexity of dynamic networks protocols. We improve the communication complexity for solving any graph problem from $\Theta(E)$ to $\Theta(V)$, thus achieving the lower bound. Moreover, $O(V)$ is also our amortized complexity of solving any function (not only graph functions) defined on the local inputs of the nodes. This proves, for the first time, that amortized communication complexity, i.e. incremental cost of adapting to a single topology change, can be smaller than the communication complexity of solving the problem from scratch. This also has a practical importance: in real networks the topology and the local inputs of the nodes change.

The first stage in our solution is a communication optimal maintenance of a spanning tree in a dynamic network. The second stage is the optimal maintenance of replicas of databases. An important example of this task is the problem of updating the description of

[*]Partially supported by Air Force contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM. A part of the work of this author was done while visiting IBM T.J. Watson Research Center.

the network's topology at every node (the well-known Topology Update problem). For this problem we improve the message complexity from $O(EV)$ to $\Theta(V)$. The improvement for a general database is even larger if the database size is larger than E.

It is interesting to note that we improved also the results of papers that used unbounded message size, from $O(E)$ to $\Theta(V)$. The time complexity of those papers was not bounded by the size of the database at all. The time complexity of our solution is $O(E + VlogV)$. (For a general database of size M it is $(M + VlogV)$.) This is optimal for the case that $VlogV = O(E)$ (or $VlogV = O(M)$).

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 using its copy of the replicated database. This general technique is one of our main contributions.

1 Introduction

1.1 The Model

In this paper, we consider *dynamic* asynchronous communication networks. In such networks, communication is completely asynchronous, and any sequence of topological changes is possible [AAG87, AS88, Awe88]. Dynamic networks are a good approximation for realistic network models. See e.g. [Fin79, MRR80, ACG+90] and Subsection 1.4.

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). Communication is performed only by exchanging messages over the edges.

Edges may fail and recover. Whenever an edge fails an underlying lower-layer link protocol notifies both endpoints of this edge about the failure, before the edge can recover. Similarly, a recovery of an edge is also notified to each of its endpoints. A message can be received only over a non-faulty edge. Message transmitted over a non-faulty edge will either arrive, or the edge will fail (in both endpoints). Messages that arrive over any given link arrive according to the FIFO discipline.

The following complexity measures are used to evaluate complexity of communication protocols. *Amortized Communication* is the total number of messages (each containing a constant number of items) sent by the protocol, divided by the number of input events that occurred during the execution. That is, this is the "incremental" communication cost, in terms of number of messages caused by a single topological change. *Quiescence Time* is the maximal *normalized* time from the last input change until termination of the protocol. *Normalized* time is evaluated under the assumption [Awe88] that links delays vary in between 0 and 1.

1.2 The Problem Statement

A node receives as an input a (possible infinite) sequence of *local* events called *changes*. Among them, we distinguish between topolgical and non-topological changes. A *topological change* is the insertion (recovery) or deletion (failure) of an adjacent edge. *Non-topological changes* are changes in all other possible local *items*, e.g. weights of adjacent edges, various available local resources, names of local attached users, etc. It is important to stress 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.

The main problem solved in this paper is a *dynamic distributed database maintenance*. Each node maintains in its local memory a *local database* which is a subset of its (topological and non-topological) items it wishes to be replicated by all other nodes. The selection of items to be included in the local database is a local decision. (E.g. it may be desired to have the names of the users replicated, but not the weights of its links.)

The task of the database maintenance algorithm is to provide each node with a description (shadow) called *replica* of local database of every other node. Consider a connected component N of the network. The solution algorithm is required to ensures the following for every node v in N: It is required that if the changes events in N's nodes cease, then the replicas in v of all the local databases in N will be equal to those local databases. See Drawings 1 (topology database) and 2 (general database). We call the collection of replicas in a node the node's *database replica*. In practice, there is no need to require that the changes stabilize forever; stability for "long enough" period suffices. This model realistically describes the mode of operation in existing networks, where topological changes occur in bursts, separated by long idle periods.

In this work, we are mostly interested in communication and time complexity of computing an *arbitrary* function. Let M be the size of the union of the local databases. The topology maintenance example given above proves that:

Lemma 1.1 There exist functions whose computation in dynamic networks requires $\Omega(V)$ amortized communication and $\Omega(M+V)$ time.

In particular for any problem that may depend on the inputs of all nodes, the lower bound for the message complexity is $\Omega(V)$.

1.3 Special cases and Applications

The special case of maintaining a dynamic tree was addressed in an earlier version of this paper. In the current version it is used as a building block.

In the example where the items are the names of the local users, the collection of the database replicas forms a distributed user directory. When such a the directory is maintained ([PRP, AP90]) it is possible to find the address and location of each user. It is worth mentioning that while the directory of [AP90] is more efficient (though not in the worst case) it lacks the fault-tolerance offered by the current solution.

An important and interesting example is where the local database of each node consists of the set of non-faulty links adjacent to this node. Maintaining replicas of this database is the classical "topology update" problem, where each node is required to "know" the description of the connected component of the network [Vis83, BGJ+85, MRR80, SG87] (Drawing 1). While not many network wide distributed protocols are used in practice, the topology update task mentioned above is the most common such algorithm used in significant existing networks [MRR80, BGJ+85, ACG+90]. That is because when the topology gets to be known to all nodes, many distributed tasks can be performed by a sequential procedure in each node. (For example, each node looks at the network topology as it appears in its own memory, and computes the best routing, chooses a leader, etc.). In summary, topology update is a *universal* distributed solution to the class of all graph-algorithmic problems. That is, a solution to topology update gives us for "free" a solution for all other graph algorithmic problems for the network's specific graph. Although not in every distributed graph problem it is required that each node will know the whole topology, our solution to the topology update problem gives us communication optimal solutions to all the investigated distributed graph problems.

We can maintain efficiently not only a topology database, but a database of any kind of items. Similar to the use of the topology database to solve any graph problem, we can thus use the solution of the the general database maintenance problem to solve the more general *distributed function computation problem.* (For example, the problem of sorting the names of the nodes.)

The function to be computed is any function (represented by a predicate F) from the collection of local databases of the nodes. The range of the function is represented by data structures called *local outputs*, maintained at each node. The algorithm is required to make F hold for the collection of databases (as inputs) and the collection of local outputs. (For example, the node whose name is the ith position in the sorted list of names, will have i as an output [Fre83b].)

We can solve these general problems by replicating the local databases at all the nodes. Once every node has replicas of all the local databases it can locally compute the local output. As mentioned, this approach is commonly used in existing networks. We show that this practical approach also leads to efficient solutions.

Let us point out other examples of the general problem solved in that way: Defining as the local database items the network's edges and some additional edges properties (like edge capacity, or weight), one can solve the problems of maximal flow, BFS (for end to end routing), MST (for broadcasts), etc.

1.4 Previous approaches

The problem of computing on dynamic networks is one of the most well-studied problems in the area of distributed computing and has been extensively studied in the last 10 years. See e.g. [Gal76, Gal77, MS79, JM82, Gaf87a, AAG87, AS88, Awe88], [AAM89, Hum81, Gar89, CRKG89, RF89]. The main motivation for dynamic networks compared to static networks is that they better model real networks which are bound to suffer failures [FLP85b] and additions of new links. There is also a stronger motivation to the efficient implementation of a task that must be performed again and again, than for a task that can be performed only once. There are various tasks needed to be computed in dynamic networks, like shortest paths, minimum spanning tree, BFS, DFS, maximum flow, minimum cost flow, and others.

Weaker models (of dynamic networks) that have been suggested have mainly theoretical value: either to show impossibility (and thus to recommend the implementation of stronger primitives) or to demonstrate that even if conditions are unrealistically unfavorable, some tasks could still be performed, though at a very high cost. For example, multiple papers avoided the use of a lower layer link protocol that detects failures [FLP85a, AG, AMS89]. In practice, networks do utilize such a link protocol [MRR80, BGJ+85, ACG+90].

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 to point out that computing from scratch requires, for many important functions e.g. a spanning tree, at least $\Omega(V)$ time and $\Omega(E)$ communication [AGPV89]. Thus, the method used above is doomed to $\Omega(E)$ amortized communication.

However, $\Omega(E)$ lower bound does not apply to amortized communication complexity of dynamic network protocols. Intuitively, we can benefit from the knowledge of the past and thus economize on communication.

It is well known that this is the case in sequential computation, e.g. Frederickson shows [Fre83a] how to maintain a dynamic minimum-spanning tree with (amortized cost) of $O(\sqrt{E})$ computations per input change, while constructing a (single) tree from scratch requires $O(E)$ computations.

Unfortunately, in the distributed computation model, it is far from obvious how to reduce the incremental cost below the cost of solving the problem from scratch. In fact, it took a long time (since 1976 [Gal76]) just to implement the "blast away" efficiently. (One of the by products of this paper is yet another improvement of the blast away task. This time optimal amortized message complexity is achieved.) Let us return to the example of the problem of maintaining a spanning tree in a dynamic network. The best currently known dynamic protocol [AAG87] for that problem requires $O(V)$ time and $O(E+V \log V)$ messages, i.e. matches the performance of protocols solving the problem "from scratch". (In [AAG87] bounded message size was assumed. Previous work that allow unbounded message size [Gal77, MS79, KM86] does not achieve better performance.) The [AAG87] protocol adopts the "classical" [Gal76],[Fin79],[Seg83],[Gaf87b], [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 attempts in the literature to avoid the obvious waste associated with blast-away approach by keeping track of the past computations and using them in the future [MS79, SG89, Gaf87b, AS88]. This methods are apparently more sophisticated. (in [SG89] a new principle was introduced to decide which message is relevant, and which is obsolete. In [Gaf87b] this is generalized to a family of principles. However, it is conjectured there that in terms of worst case complexity the blast away method is the best algorithm. We disprove this conjecture in this paper.) Oddly enough, previous methods which interpolate the information from the past, get beaten by the naive "blast away" method of [AAG87] in terms of communication complexity. The reason is that the mixing of the results of the old computation with that of the new one "confuses" the nodes that thus take wrong decisions. Those decisions cause them to send messages that have to be followed later by correction messages.

1.5 Our results

This paper shows that keeping track of history makes it possible to achieve a communication-optimal protocol for computing an arbitrary function:

Theorem 1.2 Any function f can be computed with $O(V)$ amortized communication and $O(M + V \log V)$ quiescence time.

Since *all* graph algorithms for dynamic networks developed in the last 10 years required $\Omega(E)$ communication, our results improves the amortized message complexity of *all* those protocols from $\Omega(E)$ to $\Theta(V)$. In particular, this applies for

- Breadth First Search.
- Depth First Search.
- Minimum Spanning Tree.
- Shortest Paths.
- Maximum Flow.
- Topology Update

Let us mention that, for all the problems above, we achieve the lower bound on communication complexity. For some of those problems, our improvement in communication comes on the expense of a slight increase in time, from $\Theta(V)$ to $\Theta(V \cdot \log V)$. However, when $O(V \log V) = O(M)$ our

Reference	Amortized message complexity	Quiescence Time
[AAG87]	$O(MV + E + V \cdot \log V)$	$O(M + V)$
This paper	$O(V)$	$O(M + V \cdot \log V)$
[BGJ+85]	$O(E)$ (unbounded message size)	$C + V$
Lower bound	$\Omega(V)$	$\Omega(M + V)$

Figure 1: Our improvement for computing a function.

time is optimal too. This improves the time complexity of [BGJ+85] (who have the best message complexity for topology update). Our improvements for a general problem (and for the list of the problems mentioned) are stated on Figure 1. In the figure, C is the number of changes that occurred during the execution. (Note that C may not be bounded as a function of M.)

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

To achieve this improvement we maintain dynamic trees and send over them the updates about the changes in the local databases. Therefore, we first introduce a dynamic *tree maintenance algorithm* which serves as a basic building block for all other database maintenance algorithms. This algorithm maintains a loop-free forest structure at all times which converge to a spanning tree when the topological changes cease. The amortized message and quiescence time complexities of this basic algorithm are V and $VlogV$, respectively. The tree maintenance algorithms by itself is a special case of a database maintenance algorithm and can support the maintenance of a constant size local database.

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 results of the old computations help in constructing the new computation, rather than being an obstacle.

Several problem arise in implementing the above approach. One which relates to the tree maintenance is the choice of the gluing edges. Note that in an asynchronous model there must be transient cases were some nodes "heard" of topological changes, while others have not. Thus they may have different information and choose different edges, causing (if no special care is taken) either deadlocks or cycles in the "tree". Another problems relates to the sending of updates about changes over the tree; that is: how to preserve the $O(V)$ amortized complexity despite the fact that the tree edges are dynamically replaced by other edges, at the time that the updates flow over the tree. In other words: how do we prevent each node from receiving such an update 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 the nodes to resolve inconsistencies between their views of the database.

Our algorithm can be used also for tasks other than computing a dynamic function in the memories of the nodes. For example, it is easy to use our algorithm to improve the communication complexity of the blast away task [AAG87], and therefore also to improve (in the dynamic networks model) the amortized complexity of the End To End communication problem [AG] from $O(E)$ ([AAG87]) to $O(V)$.

1.6 Bounded counters and applications to self-stabilization

As in previous works [AAG87, AG88, AMS89, AAM89, AGH-90, AG, AGR90], we do *not* use "unbounded counters" which are being incremented upon each topological change in the network, theoretically counting to "infinity". However, such counters are frequently used in practice, since a relatively small counter (64 bits per message) suffices to represent a huge number of topological changes (exceeding the expected number of nano-second until the time some astronomers foresee the end of the universe.)

Considering the above, one could question the practical value of attempts not to use such unbounded counters. This is even more questionable regarding the fact that the results in previous papers can be trivially obtained (or even improved significantly) if unbounded counters are permitted [AAG87, AG88, AMS89, AAM89, AGH-90, AG, AGR90]. The results of this paper, however, are not trivialized if unbounded counters are permitted. In fact, our results improve, both in amortized message complexity and in time complexity, the works that do use unbounded counters [BGJ+85, Vis83].

Still, there are practical reasons for getting rid of unbounded counters. One of them is to overcome another case of failure: that of a loss of nodes' memory (and consequently the highest used counter values). Another is the impracticality of handling the counters in a hardware switch in new generation networks [ACG+90].

The most significant importance of bounded counters protocols is in that they can lead to self-stabilizing protocols, that can start in arbitrary initial state. The the self stabilization property was introduced by [Dij74] and was emphasized by [Lam84] and numerous later researchers. The papers on self stabilization in general graphs [DIM89, KP89] use unbounded counters (or assume that a leader is given) that are *properly initialized*, thus undermining the independence on initial conditions.

An interesting by-product of our work is that it achieves the so-called "self-stabilizing extensions" of [BGM88, BP89, AG90, DIM89, KP89] without using unbounded counters or assumptions about the existence of a network leader. That is, any protocol can be transformed into self-stabilizing protocols, benefiting from our improved complexity.

1.7 Practical Applications of our work

From the practical standpoint, it is desirable not only to reduce the complexity of a protocol, but also to achieve a number of additional qualitative properties. Recall that the tree maintenance algorithm constantly changes the "tree-edges", namely the set of all edges marked as belonging to the tree at a given time, so that in the steady-state tree-edges really form a tree. The additional properties refer to the properties of tree-edges in the transient states.

Loop-Freedom: At all times, the set of "tree-edges" does not contain a cycle.

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

Loop-Freedom is essential in the environment of hardware-based fast packet switching [Tur88, CG88, ACG+90] [CGK88, CS88] and asynchronous transfer mode (ATM)[XVI88] which are the most dominant proposals for the future commercial integrated data/voice/video networking systems (B-ISDN).

Path-Preservation is important in virtual circuit-switching environment.

The properties of loop-freedom and path-preservation has been studied for a long time [Gal77, MS79, SS81] in the networking literature. Our protocol is the first one which does achieve those properties with bounded complexities.

As for the database maintenance, even if the network never stabilizes, but the tree edges do not fail, then it is guaranteed that every update will reach all nodes.

1.8 Structure of the paper

As a first stage in our solution for the general problem, we solve the problem of maintaining a dynamic spanning tree. Sections 2 and 3 explain the tree maintenance algorithm, that is a component in the more general algorithm. Section 2 presents the algorithm except for two subroutines FIND and UPDATE. The UPDATE subroutine, that is our main contribution in this part of the paper, is presented in Section 3 together with the FIND subroutine. Section 4 explains the $O(V)$ amortized complexity.

Section 5 gives the key points in the solution to the more general problem. The time complexity is dealt with in Section 6. The appendices contain code, explain the distributed implementations of the modules (whose high level description appears in the body of the paper) and prove the properties.

2 Preliminaries

In subsection 2.1 we define formally the tree maintenance problem whose solution is a building block for the more general problem. In Subsection 2.2 we describe the very simple main program of the algorithm. The subroutines, that are the main novel part of the tree maintenance, appear in the next section (3

2.1 Formal Definition of the Tree Maintenance Problem

In response to the topological changes (Subsection 1.1) the algorithm is required to mark in each node (i.e. put in the local output; see Subsection 1.1) a subset of the node's edges. We call the collection of edges marked by all network nodes the *real forest*. The requirement imposed on the algorithm is that the real forest is in fact, a forest at all times. Trees in this forest are called *real trees*. If the input stops changing then the output (real forest) is required to become eventually a spanning tree of the (connected component) of the final network (Drawing 3). Note that the definition does not require that any one node will "know" the entire real forest.

2.2 Informal Description

For ease of description we first describe the tasks performed by the algorithm as if the algorithm was not distributed. (Still, 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).

2.2.1 Overview

Recall that the *real forest* maintained by the algorithm is composed of locally marked edges. Failures of edges cause them to become unmarked, and potentially disconnects a *real tree* into two or more real trees. The algorithm thus "glues" distinct real trees of this forest into larger real trees, by marking edges that connect them.

We adopt a concurrent implementation of Kruskal's algorithm [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 only mark an edge which is the minimum outgoing edge of the real trees of both its endpoints. This operation is repeated as long as there is more than one real tree in a connected component of the network.

The naive way of finding an outgoing edge in a distributed network 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 name of the real tree in which they are members [GHS83]. (If they are membered in real trees with different names then this is an outgoing edge.) This requires $\Omega(E)$ messages.

Our main contribution in the tree maintenance part of the paper 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.

2.2.2 Main Program

The high level description of the algorithm outlined above appears in Figure 2. (Details about the distributed implementation are deferred to Section C. It uses the FIND subroutine to find the minimum outgoing edge of a real tree. The UPDATE subroutine is used in order to update a data structure, such that the complexity of the FIND subroutine will be reduced. Note that UPDATE is used more than once. The reason is explained in Subsection 3.3.

3 UPDATE and FIND

In this section we present the most novel part in the tree algorithm: the subroutines used by the main program described in Section 2. Subsection 3.1 introduces the dynamic data structure used in order to reduce the complexity of finding an outgoing edge. Subsection 3.2 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 topological change occurs
   call UPDATE (*correct tree replicas*)
   call FIND (*chose min outgoing edge*)

Whenever two trees chose same min outgoing edge
   for each of the trees separately call UPDATE
   then mark the chosen edge at both of its endpoints (*merge*)
```

Figure 2: 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, on the expense of a slight increase 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 (and thus the edge to that neighbor is outgoing).

Let us describe our dynamic data structure in more details. 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. (That is: v's edges that appear in v's replica are exactly v's marked edges.) A forest replica is demonstrated in Drawing 4.

If node v's forest replica includes several trees, still v's marked edges all belong to 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 the FIND subroutine to identify the outgoing edges. Note that a node's forest replica may contain other trees! They are used (by the UPDATE subroutine) just to minimize the cost of updating the tree replicas. A tree replica (as a part of a forest replica) is demonstrated in Drawing 4.

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 all its real tree.

3.2 Properties

The key properties of the subroutines are:

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

1. Once UPDATE operates in a real tree, the real tree can only shrink as a result of failures of tree edges. (This property is guaranteed by the main program: the real tree does not merge with another while UPDATE is performed. Thus it cannot grow.)
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. (This is a result of the previous property.)

- FIND subroutine 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 process FIND are similar to those of UPDATE:

  1. Once FIND operates, real tree can only 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. (To be more exact it is an minimum weight outgoing edge of a snapshot [CL85] of the real tree.)

3.3 The loop-freedom invariant

Consider the nodes of a single real tree, and their tree replicas (excluding other parts of the forest replicas). Note first, that the union of these tree replicas 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 replicas union* of a real tree contains every edge that is "believed" by some node in the real tree to belong to this real tree (Drawing 5).

The UPDATE process (described in Subsection 3.4) uses strongly the following replicas-union invariant maintained by the algorithm:

Definition 3.1 (Strong loop-freedom invariant) We say that the collection of forest replicas (one forest replica per node) preserves *strong-sense loop-freedom* if for every real tree the union of the tree replicas of the nodes of that tree (*tree replicas union*) does not contain a cycle.

Intuitively, when the invariant is kept the tree replicas of nodes of a particular real tree may be different, but at least maintain some consistency. This is important, since in distributed operation one cannot avoid the case in which one node learned about a failure of a marked edge, but has not yet notified other nodes. Thus its tree replica is different than those of the other nodes in the same real tree.

Given a particular real tree for which the invariant holds, it is relatively easy to see that the operations of process UPDATE do not violate the invariant. Now consider the case where two real trees ("left" and "right") merge. The invariant holds for each of the merging real trees separately, but potentially can be violated after the merge. The reason for this is that some node v may appear both in the union of tree replicas of "left" as well as in the union of tree replicas of "right". (Intuitively- there are nodes in the left real tree who "believe" that v is in the left real tree, and nodes in the right real tree who believe that v is on right.)

Note that v actually appears at most in one of the real trees. Thus, at least in one of those replicas, an edge leading to this node appears erroneously. In such situations, the invariant might be violated if the edge is marked and trees are merged with no further actions. As explained in Lemma 3.2 below, one of the purposes of process UPDATE is to eliminate such a possibility.

Lemma 3.2 Suppose that before an edge that was selected by both endpoints is marked, UPDATE is performed in both trees. Then, marking this edge (and gluing the trees together) does not violate the strong loop freedom invariant.

Proof Sketch: : Recall that UPDATE is performed separately by two disjointed real trees (who chose previously the same edge over which to connect). By property (2) of UPDATE, tree replicas of both the right and the left will be a subset of the initial real trees. Therefore, the tree replicas in the memories of the nodes in the right and left real trees must be distinct after both invocations of UPDATE terminate. Thus when the merging edge is marked, no cycle will appear in the tree replicas union of the merged real tree. ∎

(Intuitively it is because of property (1) of UPDATE: a part may leave, say, the left real tree during the execution of UPDATE, but cannot join the right. Thus such a part will appear either in the tree replicas union of the left real tree, or in none of the unions; hence it cannot violate the invariant.)

3.4 Process UPDATE

The UPDATE process attempts to make 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.) The UPDATE process terminates when all the LOCAL-UPDATE procedures terminated. (The distributed implementation is explained in Section C.)

Observe that in case that the tree replicas of the endpoints of an edge disagree, it is nor obvious how such "agreement" is reached, neither which one of the endpoints is "more correct". (Intuitively, left is "more correct" then 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 a "correct" way. Consider some "sample" marked edge, over which we run procedure LOCAL-UPDATE Let us call the endpoints of the sample edge "left" and "right". (The pseudo code appears in the appendix.)

By the invariant, for each edge in the tree replicas union, there is a unique undirected path in the union that starts with this edge (u, v) and ends with the sample edge. If this path enters the sample edge thru the left (right) endpoint, then we call edge (u, v) a "left" ("right") edge (Drawing 5). It will be shown that the tree replica of the left endpoint is "more correct" regarding "left" edges. Thus a left edge that appears in the replica of the left endpoint but not at the right endpoint is copied also into the replica of the right endpoint (Drawing 6a). A left edge that does not appear in the replica of the left endpoint is removed from the replica of the right endpoint (Drawing 7). Similarly, the right endpoint replica is considered "more correct" regarding right edges. We term this method (of deciding who is "more correct" about an edge according to who is closer on the tree to the edge) the *tree belief principle*.

3.4.1 Use of the Other Trees in a Node's Replica

Some care is required here in order to save communication. Whenever a new edge is marked, we do not attempt to save communication, and simply transmit the whole forest replicas of the endpoints over the edge. Thus, in later activations of LOCAL-UPDATE the left endpoint, for example, already knows the replica of the right one (at least regarding left edges). Thus it suffices that the left endpoint transmits only corrections to this replica, rather than its whole replica (Drawing 6b). Clearly this does not change the algorithm. However, the amortized complexity is reduced significantly. This reduction uses also the fact that nodes remember a forest replica, not only their tree replica. The reduction is demonstrated in Example 3.3.

Example 3.3 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 real tree $T\prime$ (Drawing 6b). Assume further that "left" "knows" that the forest replica of "right" contains the description of real tree T_1. Now "left" must send "right" only the information about the new marked left edge, rather than the whole description of real tree T_1 that now became a part of T.

3.5 Process FIND

To find 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 this is an outgoing edge. That 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. For that we use a method that is rather standard in distributed computing. It is described (with the other details of the distributed implementation) is Section C.

4 Message Complexity of UPDATE

Most of the analysis of the amortized message complexity is given after the explanation of the distributed implementation. However, it is possible already to count the number of times an identity of an edge is exchange between nodes in procedure LOCAL-UPDATE. This is actually the message complexity of Process UPDATE. The proof of the following theorem appears in Appendix B.

Theorem 4.1 Assume that exactly k topological changes occur. Then the number of edges identities exchanged by process UPDATE during all the execution of the algorithm is is $O(Vk)$.

5 The Database Maintenance Problem

For clarity we only explain here the method for performing topology maintenance. The method to maintain a general database is basically identical.

5.1 Problem Definition

In response to the topological changes (or the other changes, in the case of a general database) the database is updated. If the topology changes cease then it is required that each node will have in its memory a correct description of the node's connected component. (It is allowed to contain an incorrect description of other connected components.) This description of the whole network is called the *topology replica* of the node.

5.2 Why does a Trivial Change to the Tree Algorithm Not Suffice

The tree maintenance algorithm also maintains a database (the forest replicas). In principle, one can adapt it for maintaining topology with a very small change. However, the amortized complexity in such a case would be $O(E)$ for topology update, and $O(M)$ (the size of the union of the local databases). for the general database case, instead of our goal of $O(V)$. Let us point at the exact place in the tree algorithm where the complexity would be increased beyond $O(V)$ if used for topology (or general database) update.

In the tree maintenance algorithms whenever an edge is marked as a real tree edge, a complete exchange of the forest replicas is performed between the two merging roots. In the tree maintenance part each replica is of size $O(V)$. Therefore $O(V)$ messages are exchanged. However, in the case of topology update each replica is of size $O(E)$ (or M for a general database which might be larger than E.) Thus even the complexity of this exchange alone is more than $O(V)$ per topology changes.

However the tree maintenance algorithm can be used with $O(V)$ amortized complexity to maintain a database of a constant number of items per node. We use this fact strongly in Subsection 5.5

5.3 Reducing Message Complexity

Our topology maintenance algorithm uses the tree maintenance algorithm. In periods where the tree stabilizes, a new TOPO-UPDATE subroutine is activated. Note that several edges may have been added to the tree since the last activation of TOPO-UPDATE. Let us call them the *new* real tree edges.

The idea of TOPO-UPDATE is still somewhat similar to that of UPDATE used in the tree algorithm. However, it does not exchange the full topology replicas over each new real tree edge (unlike the full forest replicas that are exchanged in the tree maintenance). We allow only an exchange of no more than V items. For that we introduce the notion of nodal counters. Whenever a topological change occurs in the node (addition or deletion of an adjacent edge) the node increments its counter by one, and *stamps* the change with the new value of the counter (Drawing 8). That is- the pair (change, value of counter) is recorded by the node and will be reported to other nodes (at the time TOPO-UPDATE will be performed) together. The data structure maintained by each node is not only the topology replica, but also the lists of changes sent by all the other nodes. (This seems to require infinite memory. The method to bound the memory in explained in Section 5.5.)

The method to bound the values of the counters is deferred to Subsection 5.4. Meanwhile let us show how they can be used to reduce the amortized message complexity assuming that a message can contain a constant number of values of counters.

Let us now define the V items exchanged by the endpoints "left" and "right" of a new tree edge. For each node v in V the endpoints exchange the highest counter of v they "know" of

Let u be a node in the real tree of “left” and “right”. (As in the tree algorithm, we do not send updates regarding nodes in other real trees. This is important for Subsection 5.4.) Let u_{left} be the highest value of the counter of node u appearing in the list of u's changes in “left”. Similarly, u_{right} is the highest value of the counter of u “known” “right”. If $u_{left} > u_{right}$ then “left” sends “right” every change of u that is stamped with a counter higher than u_{right} (up to, and including, the one stamped with u_{left}).

As for non-new edges, each of their endpoints already “knows” what is the highest stamp of u “known” to the other endpoint. Thus it sends only changes of u with higher stamps. This operation is repeated by the two endpoints, for each node u in the real tree.

5.4 Bounding the Counters

Using the counters we managed to send only $O(V)$ messages per change. Note, however, that we have introduced a new problem of unbounded bit communication complexity since we have not yet shown a method to bound the size of a counter. In the following we will ”repair” this flaw by bounding the counter by a polynomial in the network‘s size.

5.4.1 Basic Idea

The bounding of the counter is, intuitively, a “reset” operation rather than a wrap around. When a counter of node u reaches its bound we basically reset it back to zero. Now it is needed to erase all the information regarding node u in the memory of the other nodes. Otherwise they will “remember” higher counter values of u (those of before the reset of u's counter) and thus “believe” they “know” later information. (This could cause them to send outdated information to their neighbors who “heard” changes of u that occurred later, but had lower stamps values.)

This erasure may be impossible, since some of the other nodes may at this point not be in the same connected component. Actually we do not attempt to erase u's information at nodes in other real trees. This erasure will be performed after they join u's real tree, and before they are permitted to send any information about u. (Recall that in TOPO-UPDATE, a node sends information only about other nodes in its real tree. This means that u does have the opportunity to erase u's outdated information in nodes that join its tree, before they have the chance to propagate this outdated information.)

5.4.2 The Reset

A node whose counter reached the bound (to be computed later) resets its counter to zero. Next it considers all its edge as if they were inserted just now. Thus it increments its counter starting from zero, and stamps its edges with new values of the counter. All the previous changes are erased from the list of changes.

The time between two reset operations in node u is called a ”phase” of node u. In order to preserve the correctness of the unbounded version, node u must guarantee previous to any database exchange or update that all the nodes participating in the exchange are aware of node u's new phase. In order to do that a node keeps the “good” list of nodes that have “heard” about its last reset operation. Database exchange starts only after Process UPDATE of the tree algorithm has terminated and each node has checked that all nodes of the tree (including the recently added ones) are included in its "good" list. If some nodes in the real tree are not included in the "good' list of some node u (in the same real tree) then u instructs them to erase all their information regarding itself. (This means that the highest stamp of u "known" to them will now be zero.) After such a node acknowledges, it is added to u's "good" list. The details of the distributed implementation are deferred to Appendix C.

Finally let us note that an erasure operation may fail because of some tree edges failure. Thus we first perform a single erasure operation between a selected pair of nodes. If this fails, the message cost is $O(V)$ and can be amortized on the topology change that caused the failure. If, however, the erasure was successful, then we proceed with all other erasures in parallel. Of course these parallel erasures may fail too. However, our potential function (for the amortized complexity) includes the values of the counters. The decrease in the value of the erased counter in the successful single attempt is high enough to compensate for the cost of the failed parallel erasures. This dictates the bound on the counters. The computation of this (polynomial) bound is deferred to the full paper.

5.5 Memory and Quiesces Time Complexities

Recall that an "outside" event can delete an item from the local input of a node. (For example an edge may fail.) It may happen that all but a small number M of items were deleted. Thus only M time is required in order to send their description. However, the algorithm as described so far, keeps the list of delete events too. This list may be much longer than M. This consumes memory and takes a long time to be transmitted.

Let us recall the purpose of keeping the list of deletes. Assume that when some node was disconnected from the rest of the network, many delete events occurred. When the node is reconnected, it is informed about all these delete events, in order to remove the corresponding items from its database replica. The number of delete events that occurred while the node was disconnected may be arbitrarily large (since insert events could also occur). Let us show how to bound the number of delete events a node must keep.

First notice that if many changes are remembered for the same item (insert, delete, insert again, delete again, ...) then only the last (highest stamp) change must be kept. In the case of topology database this immediately reduces the table size to be one entry per potential link. However, if only this technique is applied, a node is still required to remember all past deleted edges (or other items). In the following we explain how a node can keep a database which is proportional in size to the actual number of non-deleted items.

Let m_v be the number of (non-deleted) items in the local database of node v. Our algorithm keeps the following invariant for each node v:

Definition 5.1 (Bounded memory invariant) We say that the items in the local database of a node v keep the *bounded memory invariant* if the stamp of each item in the local database is larger than the value of the node's counter minus $3m_v$.

Note that the insertion of an item increases $3m_v$, and thus the stamping of the item cannot violate the invariant. When an item is deleted there is a danger that the invariant will be violated. To prevent this, node v restamps its items. In principle the node creates artificial events of insertion for all its non-deleted items. They are thus stamped with higher values of the counter.

Note that after this is done, the last m_v changes are all (artificial) insertions with no deletions between them. We call this procedure a *local compaction.*

Since the artificial inserts must be communicated to other nodes, the local compaction increases the message complexity. The following lemma states that the amortized message complexity is at most doubled.

Lemma 5.2 (Artificial Inserts) Let k be the total number of changes during the run of the algorithm. Then, the number of artificial inserts is at most k.

The proof appears in Appendix E.

Recall (Subsection 5.2) that the tree maintenance algorithm can be used to maintain (in addition to the tree itself) a fixed size local database for every node. We use it to broadcast to every node the value of the counter of every node, and the number m_v of non-deleted items at every node v.

Now every node needs to remember only the changes of v that are stamped with values that are the last known counter of v minus $3m_v$. Any item whose last insert is stamped with a lower value is thus deleted (without an explicit delete message).

It is now clear that once topology and local database changes cease no more than three times the final number of items of the union of the local databases will be exchanged. Since pipelining is used in that exchange, and it is done over a tree, the quiescence time will be linear with that size ([CKMP89]), which is a lower bound.

6 Reducing Time Complexity

For the sake of simplicity, we have only described a version of the tree maintenance algorithm whose quiescence time complexity is $O(V^2)$. In order to reduce it to the claimed $O(V \cdot \log V)$ it is required to be able to merge more than two trees in the same merge (similar to the case in [GHS83]). If this is done, some additional operations are required to maintain the strong loop freedom invariant. (Intuitively, instead of removing inconsistencies between the tree replicas union of two trees, we must do it for more than two trees.) The details are deferred to the full paper.

References

[AAG87] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, 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.

[ACG+90] Baruch Awerbuch, Israel Cidon, Inder Gopal, Marc Kaplan, and Shay Kutten. Distributed control for paris. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990. To appear.

[AG] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization: An efficient technique for end-to-end communication. Unpublished manuscript.

[AG88] Yehuda Afek and Eli Gafni. End-to-end communication in unreliable networks. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 131–148. ACM SIGACT and SIGOPS, ACM, 1988.

[AG90] Anish Arora and Mohamed Gouda. Distributed reset. see abstract, 1990.

[AGH-90] Baruch Awerbuch and Oded Goldreich and Amir Herzberg A quantitative approach to dynamic networks Proceedings of the ACM PODC 1990.

[AGPV89] Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A tradeoff between information and communication in broadcast protocols. *J. of the ACM*, 1989.

[AGR90] Yehuda Afek, Eli Gafni, and Adi Rosen. Slide - a technique for communication in unreliable networks. Unpublished manuscript, January 1990.

[AM86] Baruch Awerbuch and Silvio Micali. Dynamic deadlock resolution protocols. In *Proc. 27th IEEE Symp. on Foundations of Computer Science*, October 1986.

[AMS89] Baruch Awerbuch, Yishay Mansour, and Nir Shavit. End-to-end communication with polynomial overhead. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989.

[AP90] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990.

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 206–220, October 1988.

[Awe88] Baruch Awerbuch. On the effects of feedback in dynamic network protocols. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 231–245, October 1988.

[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.

[BGM88] J.E. Burns, M.G. Gouda, and R.E. Miller. On relaxing interleaving assumptions. Technical Report GIT-ICS-88/29, Georgia Institute of Technology, 1988.

[BP89] J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[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. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 74–89. ACM, August 1988.

[CKMP89] Israel Cidon, Shay Kutten, Yishay Mansour, and David Peleg. Greedy packet scheduling. Unpublished manuscript, December 1989.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comput. Syst.*, 3(1):63–75, February 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. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 224–236. ACM SIGCOMM, ACM, September 1989.

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

[Dij74] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643–644, 1974.

[DIM89] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, 1989. Also, available as MCC Technical Report No. STP-379-89.

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

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

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

[FLP85a] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. of the ACM*, 32(2):374–382, 1985.

[FLP85b] 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.

[Fre83a] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees. In *Proc. 15th ACM Symp. on Theory of Computing*. ACM SIGACT, ACM, April 1983.

[Fre83b] Greg N. Frederickson. Tradeoffs for selection in distributed networks. In *Proc. 2nd ACM Symp. on Principles of Distributed Computing*, pages 154–160. ACM, August 1983.

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

[Gaf87a] Eli Gafni. Topology resynchronization: A new paradigm for fault tolerance in distributed algorithms. In *Proceedings of the Amsterdam Workshop on Distributed Algorithms*. CWI, July 1987.

[Gaf87b] Eli Gafni. Topology resynchronization: A new paradigm for fault tolerance in distributed algorithms. In *Proceedings of the Amsterdam Workshop on Distributed Algorithms*. CWI, July 1987.

[Gal76] Robert G. Gallager. A shortest path routing algorithm with automatic resynch. Technical report, MIT, Lab. for Information and Decision Systems, March 1976.

[Gal77] Robert G. Gallager. An optimal routing algorithm using distributed computation. *IEEE Trans. on Commun.*, 25:73–85, January 1977.

[Gar89] Jose J. Garcia-Luna-Aceves. A unified approach to loop-free routing using distance vectors or link states. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 212–223. ACM SIGCOMM, ACM, September 1989.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Lang. and Syst.*, 5(1):66–77, January 1983.

[Hum81] Pierre A. Humblet. An adaptive distributed dijkstra shortest path algorithm. Technical Report CICS-P-60, Center for Intelligent Control Systems, MIT, May 19881.

[JM82] J. Jaffe and F. Moss. A responsive distributed routing protocol. *IEEE Trans. on Commun.*, 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.

[KP89] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. Technical Report MCC Technical Report Number STP-379-89, MCC, 1989.

[Lam84] Leslie Lamport. Solved problems, unsolved problems, and non-problems in concurrency. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984. Invited lecture.

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

[MS79] P. Merlin and A. Segall. Failsafe distributed routing protocol. *IEEE Trans. on Commun.*, COM-27:1280–1288, September 1979.

[PRP] SC66/0601 PRPQ, program number 5799-CZE. Callup instalation and refference guide. IBM Product Manual.

[RF89] Balasubramanian Rajagopalan and Michael Faiman. A new responsive distributed shortest path routing algorihtm. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 237–246. ACM SIGCOMM, ACM, September 1989.

[Seg83] Adrian Segall. Distributed network protocols. *IEEE Trans. on Info. Theory*, 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.

[SG87] L. Shrira and O. Goldreich. Electing a leader in the presence of faults: A ring as a special case. *Acta Informatica*, 24:79–81, 1987.

[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.*, COM-29(5):689–695, May 1981.

[Tur88] J. S. Turner. Design of a broadcast packet switching network. *IEEE Trans. on Commun.*, 36(6):734–743, June 1988.

[Vis83] U. Vishkin. A distributed orientation algorithm. *IEEE Trans. on Info. Theory*, June 1983.

[XVI88] CCITT Study Group XVIII. Report 55 (a)-(c), seoul meeting. Report 55, CCITT, February 1988.

A APPENDIX: LOCAL-UPDATE: Code in node v

Whenever LOCAL-UPDATE is invoked or after processing a DIFF message
    $\forall$ k s.t. (v,k) is marked and Forest(j) $\neq$ undefined) (*Forest(j): forest replica received from j*) do
        send message DIFF(diff(v,k)) to k
        (*forseeing the way k will change its Forest replica:*) Forest(k) := Forest(k) $\oplus$ diff(v,k)

for (*receiving message*) DIFF(Add,Del) from k
    Forest := Forest $\oplus$ (Add,Del) (*updating forest replica*)
    Forest(k) := Forest(k) - tree$_k$(Forest(k) - $\{(v,k)\}$) $\cup$ tree$_k$(Forest - $\{(v,k)\}$)
    (*guessing forest replica update performed in k*)

Figure 3: **Procedure LOCAL-UPDATE**

B Proof of the UPDATE complexity Theorem

.

Proof Sketch: The full proof is deferred to the full paper. Let us give an intuitive explanation. Consider first the exchange of the forest replicas on a new real tree edge, at the time that the edge is marked. This edge was selected to be put into the tree because of some topology change (either the failure of another edge, that disconnected the real trees, or the recovery of this edge that now became marked.) The two endpoint exchange their forest replicas that are $O(V)$ edges each. Thus $O(V)$ identities of edges are exchanged per topological change.

Consider now the operation of procedure LOCAL-UPDATE. We show that when LOCAL-UPDATE deletes an edge from the tree replica of a node, this corresponds to a unique delete event in this edge's endpoint. An insert of an edge into the tree replica of a node corresponds to a unique marking event for that edge in the edge's endpoint. (Recall that the latter is bounded by the number of input change events.)

Assume for the discussion that the events (both input changes and edges' marking) are numbered. Consider a node v who makes changes (inserts and deletes) to its tree replica edges of node u. In the correspondence we show, of every two changes in u's edges in i tree replica, the later change corresponds to a an input change (in u) whose number is higher.

Consider the effect of the change in u on u's real tree when UPDATE is performed. Note that the activation of LOCAL-UPDATE causes actually a broadcast of the information about the change to be sent over the tree: First u changes its tree replica, then its neighbors adopt (according to the tree believe principle) their replicas to agree with that of u about the change. Next their neighbors change their replicas, etc. Thus we may we speak about a change-message being propagated.

As long as nodes v and u are in the same real tree, the change- messages of u arrive at v on the unique path on the tree. Thus they indeed arrive in the correct order of their numbering. Thus each arrives exactly one.

Consider now the case that v becomes disconnected from u (by a failure in a marked edge on the path between u and v on their real tree). Assume further that the real trees of u and v are reconnected, by marking another edge. There seems to be a danger that v will receive a change-message about change c of u over the path in the old real tree, and also the path in the new real tree.

This, however will not happen. To see why recall that after the new edge to connect the real trees is selected, Process UPDATE is executed in each of the real trees separately. Assume that node v have received the change- message about change c. Then, when UPDATE is performed in v's real tree (separately) the endpoints of the connecting edge will receive the message-change about c too (or even one of them have heard of a later change). Moreover, the endpoint in v's real tree will "know" that its neighbors in the real tree already incorporated c in their replicas. Thus it will not send c again. ∎

C Distributed Implementation of the Tree Maintenance

We elaborate on the high level overview given in Sections 2 and 3. We first assume the existence of subroutines (UPDATE and FIND) that tell each node which of its edges is outgoing. (The subroutines, to be described in Subsections C.1.2 to C.1.4, are the distributed version of Subsection 3.4 and Subsection 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 C.1). (Note, though, that at no point of time it is known that no additional topological changes will not occur.) Later (Subsection C.2), we describe the additional steps taken when topological changes occur during execution.

C.1 Operations When no Additional Topological Changes Occur

C.1.1 Main Program

Each tree has a unique root who coordinates the connection of this tree to others. Its first task is to activate the UPDATE and FIND subroutines that eventually terminate 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. (The rootship *migrates.*) The rootship continues to migrate until it reaches the node adjacent to the minimum outgoing edge.

When this node becomes the root, it tries to reach an agreement with the other endpoint of the minimum outgoing edge to merge the the two real trees. If the other endpoint is also the root of its real tree, and this edge is also the minimum outgoing edge of the other real tree, then it is guaranteed that the endpoints will agree. (Recall that this holds if no additional topological changes occur.) The edge will be marked on both sides as a tree edge, and one of the endpoints will be chosen to be the unique root of the united tree.

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

C.1.2 Subroutines: Search for the Minimum Outgoing Edge

Recall (Subsection 3.5) that a node discovers 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 assume, w.l.o.g. that each node knows the identities of its neighbors.) It is left to 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 edge to find the global minimum.

Let us start with process UPDATE. The root broadcasts an instruction to the real tree nodes to perform procedure LOCAL-UPDATE. A node which receives the broadcast forwards it to its children on the tree. (Node v is a *child* of node u if the route from v to the root over the tree passes via u.) Next it performs procedure LOCAL-UPDATE (Subsection 3.4.) The root detects that all the activations of LOCAL-UPDATE terminate, using the termination detection algorithm of [DS80].

The tool used to find the minimum outgoing edge (given that each node "knows" which of its adjacent edges is outgoing) is the standard WAVE&ECHO search e.g. [BGJ+85, DS80, GHS83, Seg83, AM86]. (We forward the search only via the real tree (marked) edges.)

The WAVE is a broadcast, similar to the one explained for process UPDATE.

In the ECHO part of this algorithm each node waits for all reports from its children. (A leaf does not need to wait). Next it compares the minimum weight reported by its tree children, and the weight of its adjacent minimum outgoing edge. 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 each of its children. The minimum edge reported to the root is selected to be the minimum outgoing edge of the real tree.

C.1.3 Subroutines: Root Migration

Consider again the search for the minimum outgoing edge (previous Subsubsection). 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 adjacent to it or 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 the pointers to the endpoint of the minimum edge.

C.1.4 Agreement between Two Real Trees

,

Out of the two endpoints of the minimum outgoing edge, only the one with the lower nodal identity is responsible for offering a connection. This offer is recorded in the higher identity endpoint. When the tree of the higher endpoint has chosen this edge too (and transferred the rootship to its endpoint) it sends the lower endpoint a message agreeing to the offer. (The tree of the higher endpoint may have chosen this edge either before or after it received the offer.)

C.2 Additional Steps taken when Topological Changes Occur

No additional steps are necessary to guarantee termination of the WAVE&ECHO search:

Lemma C.1 Topological changes do not prevent the termination of the WAVE&ECHO search.

Proof Sketch: : If an edge becomes operative during the execution of the WAVE&ECHO, then it is still not a part of the real tree, and thus does not participate in the search at all. If a real tree edge between a parent and a child fails, then it leaves the tree. Thus the child stops to be a child, and the parent will not wait for its ECHO before sending an ECHO to its own parent (or terminate, if it the root). The child which lost its parent will not send an ECHO message. ∎

Comment C.2 A similar argument holds for the termination detection of process UPDATE.

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.

Each non-root node has a pointer to its parent on the real tree. The root has no parent. When the edge of a node to its parent fails, it becomes a root. Thus there is always a root. Like the previous root, the task of the new root starts by searching for an outgoing edge. If a search for a minimum edge is on-going (including process UPDATE) then the new root waits until it detects its termination. Then it restarts the search. (Termination is detected simply when all children are marked as having sent the ECHO reply.)

If any node on the real tree notices any other topology change, then it sends an alert message to the root. This enables the root to know that the minimum edge may have changed. (Agreement is guaranteed to be reach only if the chosen edge is the minimum outgoing edge. Thus an offer to connect another real tree over non-minimum edge may never be agreed upon by the other endpoint. Thus if no agreement was reached, the real tree must search again for the new minimum outgoing edge.)

When the root receives the alert message it must first check whether an agreement with another real tree has been reached. It may have offered a connection to another tree, and has not received an agreeing message from the other tree. (The case that the root offered 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 no such offer were made then the root restarts a search for the minimum outgoing edge.

However, if the root did offer a connection to another real tree (and has not yet received an agreement) then the root tries to check the status of the offer. It asks the other endpoint, whether it has already agreed to the offer. If the other endpoint did agree, then it need not answer the question since the agreement will eventually arrive (unless the edge fails, cancelling the connection). Otherwise the higher endpoint sends a reply that cancels the offer. . (Since the edge may no longer be the minimum outgoing, then the other real tree may never chose it for connection. Thus the offer must canceled, so that another offer can be made on the current minimum.) That frees the offering root to restart the search for the current minimum outgoing edge.

D Distributed Implementation of the General Database

As in the tree algorithm, each node maintains a data structure per each of its neighbors on the tree. This data structure is an approximation of the database replica of this neighbor. Whenever a real tree edge fails, the data structure associated with it is discarded.

In addition, whenever a change occurs, the node stamps the change, as described in Subsection 5.3. This can generate artificial changes, as described in Subsection 5.5. This can also cause the reset of the counter (and the erasure of the "good' list) of the node, as described in Subsection 5.4. Note that these are all local operations, that are not communicated at this time to the other nodes.

In addition to the above local operations, the general database maintenance algorithm contains one subroutine that is activated by the root of a real tree. This root is determined by the tree maintenance algorithm.) The subroutine is activated whenever the real tree determines that it is a spanning tree. (That is: whenever Process FIND terminates without finding an outgoing edge, and the root has not been notified yet, by an alert message, about another topology change.)

This subroutine, called Process DATABASE, first selects by a WAVE&ECHO (similar to Process FIND) a node that needs to erase its list in another node if such node exists. If affirmative, then the root instructs that node to perform a single such erasure. The node performs this remote erasure by sending a reset message to one of the nodes not in its "good" list, receiving an acknowledgement (meaning that the erasure was performed) and notifying the root.

Next the root instructs the the real tree nodes to perform all other remote erasure operations they need to do. For simplicity and time complexity reasons, each node that must erase its list in some other nodes, erases its list in all other nodes in the real tree. It can be shown that this does not increase the order of the amortized message complexity. The termination of the resetting process is detected by the root using the termination detection algorithm of [DS80].

Next Process DATABASE-UPDATE is activated by the root. (We term it TOPO-UPDATE if the database to be maintained is only the topology database.) This process is similar to Process UPDATE. The real tree nodes are instructed, by a WAVE broadcast from the root, to start the update process.

Endpoints of a real forest edge detect that this edge is new in the case that their data structure for the other endpoint is empty. They exchange a vector of V values, that contains, per each node v, the highest counter value stamping a change received from node v.

Next procedure LOCAL-DATABASE-UPDATE is invoked by the endpoints of each real tree edge. The changes exchanged by this procedure were described in Subsection 5.3. The operation each node performs on the changes it received were described in Subsection 5.5.

E Proof of the Artificial Inserts Lemma

We assume that at some point a compaction was performed in node v. After this process there are B consecutive "artificial" events stamped with consecutive counter values. We will compute how many "real" events must take place before the next compaction is triggered and how many "artificial" events will be created. By induction we assume that the above B artificial events were preceded by at least B real events (so they are already paid for).

Let us assume that the B artificial events are followed by x real events which consist of i insertions and d deletions. ($i + d = x$.) Clearly at this point the number of existing items in the local database of node v after the xth real event are $m_v = B + i - d$. The earliest existing item is at distance K from event x where $K \leq B$. (Some deletion operations might have deleted some items or all items in the list of artificial events). K can be even negative since it might be the case that all the first B items have been deleted together with some of the following real additions items.

As long as the bounded memory invariant holds: $x + K < 3m_v$ or $i + d + K < 3B + 3i - 3d$ $\geq> 4d < 3B - K + 2i$. Since the x real event is the first to violate this than it must be a deletion. At this point $d => (3B - K + 2i)/4$. for the first time. The minimal value of d is when $B = K$ and therefore: $d \geq (B + i)/2$.

At this point the compaction process will create m_v artificial events following the x real events. The ratio $\frac{m_v}{x}$ represents the number of artificial events that can be created per a single real event. $\frac{m_v}{x} = \frac{m_v}{i+d} = \frac{B+i-d}{i+d}$. This ratio will be maximized by minimizing the value of d. Therefore: $\frac{m_v}{x} \leq \frac{B+i-\frac{B+i}{2}}{i+\frac{B}{2}+\frac{i}{2}} = \frac{B+i}{B+3i} \leq 1$. Therefore, at most one artificial event will be triggered for each real event. ∎

```
Definition: Let A,B,C be sets of edges.
    A ⊕ (B,C) is defined as A ∪ B - C.
    A proj B is defined as { (i,j) s.t. ((i.j) ∈ A and ∃ k s.t. (k,j) ∈ B) }
function diff(i,k: nodes; returns ((Add, Del): pair of lists of edges)
    (* Add: list of edges to be added to Forest(k) since they are in Forest(i), and are on*)
    (* i's side of the tree. Del: edges that k should delete since they don't appear in i's Forest although*)
    (* they are on i's side of the tree (even according to k's Forest) *)
    Add := ((Forest(i)-Forest(k)) proj first-arg's-side(i,k)
    Del := ((Forest(k)-Forest(i)) proj first-arg's-side(i,k)
    return(Add,Del)

Function first-arg's-side(i,k: nodes; returns a tree)
    (* returns the first argument's side of tree_i relative *)
    (* to its edge from k *)
    return (tree_i(Forest(i) - {edge (i,k)}))

Function tree_i(f: forest; returns a tree)
    (* returns the tree in f that includes node i *)
    (* convention: Tree is tree_i(Forest) in i *)
```

Figure 4: **Functions used by Procedure LOCAL-UPDATE**