Maintenance of a Spanning Tree in Dynamic Networks

Shay Kutten[1] and Avner Porat[2]

[1] IBM T.J Watson Research Center, P.O. BOX 704
Yorktown Heights, NY 10598
and William Davidson Faculty of Industrial Engineering & Management
Technion, Haifa 32000 Israel
kutten@ie.technion.ac.il
This research was supported by the fund for the promotion of research at the Technion.
[2] William Davidson Faculty of Industrial Engineering & Management,
ieavner@ie.technion.ac.il.

Abstract. Many crucial network tasks such as database maintenance can be efficiently carried out given a tree that spans the network. By maintaining such a spanning tree, rather than constructing it "from-scratch" due to every topology change, one can improve the efficiency of the tree construction, as well as the efficiency of the protocols that use the tree. We present a protocol for this task which has communication complexity that is linear in the "actual" size of the biggest connected component. The time complexity of our protocol has only a polylogarithmic overhead in the "actual" size of the biggest connected component.
The communication complexity of the previous solution, which was considered communication optimal, was linear in the network size, that is, unbounded as a function of the "actual" size of the biggest connected component. The overhead in the time measure of the previous solution was polynomial in the network size.
In an asynchronous network it may not be clear what is the meaning of the "actual" size of the connected component at a given time. To capture this notion we define the *virtual component* and show that in asynchronous networks, in a sense, the notion of the virtual component is the closest one can get to the notion of the "actual" component.

1 Introduction

1.1 Motivation and Existing Solutions

Maintaining a common database in the local memory of each node is a common technique in computing a distributed function. An important example is the case that the replicated information is the set of non-faulty links adjacent to each node. Maintaining replicas of this information is the classical "Topology Update" problem, where each node is required to "know" the description of its connected component of the network [Vis83,BGJ+85,MRR80,SG89,CGKK95,ACK90].

This is one of the most common tasks performed in existing networks since, when the topology gets to be known to all nodes, many distributed tasks can be reduced to sequential tasks. This reduction is conducted by having each node simulate the distributed task on the topology known to it. An Example for a protocol which uses this approach is the Internet OSPF interior routing protocol [OSPF].

In [ACK90] it was shown that the incremental cost of adapting to a single topology change can be smaller than the communication complexity of the previous approach [AAG87] of solving the problem "from scratch". (A variant of the algorithm of [ACK90] was implemented later as a part of the PARIS networking project at IBM [CGKK95].)

In this paper we improve the first subtask of [ACK90]: maintaining a spanning tree in the dynamic network, thus, also improving the database maintaining algorithm. Our algorithm also improves any other algorithm which uses the maintenance spanning tree algorithm as a building block. The amortized communication complexity of the tree maintenance subtask in [ACK90] was $O(V)$, where V was the number of nodes in the network. In a dynamic network the value of this parameter might be much larger than the "actual" size of the biggest connected component.

An example for such a scenario is when all the nodes in the network are separated from each other and only one edge recovers to create a two-node connected component. Therefore, The size of the network is $V/2$ times bigger than the "actual" size of the connected component. Moreover, under this scenario, according to the algorithm in [ACK90], the two nodes of the connected component exchange $O(V)$ messages. Therefore, this scenario demonstrates that the amortized communication complexity in [ACK90] is not bounded as a function of the size of the biggest connected component in the network.

The quiescence time of the algorithm of [ACK90] was high: $O(V^2)$, mainly because the algorithm merged only two trees at a time in each of its phases. Merging more trees in the same iteration of the algorithm of [ACK90] could have violated an important invariant (called the *loop freedom invariant*), the correction of the algorithm relied upon. This was, probably, the reason only two trees were merge at a time in [ACK90].

1.2 Our Solution

In this paper we provide a tree maintenance protocol with amortized communication complexity that is linear in the "actual" size A of the connected component in which the algorithm is performed (The message complexity of the previous solution was not bounded as a function of A.) The time complexity overhead of our protocol is only polylogarithmic in the "actual" size of the biggest connected component. (The time overhead of the previous solution was polynomial in the network size).

In an asynchronous network it may not be clear what is the meaning of the "actual" size of the connected component at a given time. To capture this notion we define the *virtual component* and show that in asynchronous networks, in a sense, the notion of the virtual component is the closest one can get to the notion of the “actual” component.

Topological changes may break a tree into several rooted trees. The execution of our algorithm is composed of iterations that are invoked continuously by every root of a tree, until the connected component is spanned. Each iteration of the algorithm is constructed from two phases: in the first phase, referred to as the *Trees Approval* phase, a set of trees is ”chosen” and ”prepared” for merging. In the second phase, referred to as the *Trees Merge* phase, the actual merge is executed by parallel invocations of the *TREE MERGE* procedure for every tree that was ”chosen” in the Trees Approval phase. Some topological changes which occur during the algorithm execution may cause the algorithm to initiate the first phase while other topological change may not influence the algorithm execution. We show that this two-phase structure enables the algorithm to maintain the *loop freedom invariant* of [ACK90] even though our algorithm does merge more than two trees at a time. (This uses especially the fact that a non-”chosen tree” does not participate in the second phase even if an edge from it to a node in a chosen tree has recovered). In that way the algorithm improves the time bottleneck of the algorithm of [ACK90].

The communication improvement is achieved by using a new message exchange policy: exchanging fewer messages between every two merging trees. Our algorithm exchanges only messages that describe the ”approved” trees and not the whole local memory of every node as in the algorithm of [ACK90]. We show that sending fewer messages during tree’s merging can, in fact, cause additional messages to be sent at later times during the run of the algorithm. (This, probably, is the reason the algorithm of [ACK90] did not try to economize on messages between trees at the time of merging). However, we manage to show that the number of these additional messages is small, leading to the improvement in the total number of messages.

Another difference from the work of [ACK90] is an adaptation we had to make to one of the building blocks used in the algorithm of [ACK90]. This is needed to cope with our algorithm’s new message exchange policy.

An additional change that had to be made in our algorithm is the addition of a new message, *range deletion* message, that instructs its receiver to update more than one item in its local database. We show that the use of that new message enables the algorithm to keep its time complexity as a function of the size of the “actual” connected component rather than a function of the number of topological changes that occurred.

The rest of the paper is organized as follows: Section 2 describes the model and the problem. Section 3 presents a high level overview of the algorithm of [ACK90] and the algorithm of [AS91] (which is used as a building block in our algorithm). Section 4 presents Phase 1 of our algorithm (Trees Approval) and Phase 2 (Trees Update). Section 5 contains the correctness and complexity analysis.

2 The Problem

2.1 The Model

The network is represented by a graph G=(V,E) where V is the set of nodes, each having a distinct identity, and $E \subset V \times V$ is the set of edges, or links. We assume that each edge has a distinct finite weight. (If the edge weights are not distinct one simply appends to the edge weight the identities of the two nodes joined by the edge, listing, say, the lower ordered node first [GHS83]).

The network is dynamic: edges may fail or 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 [MRR80,BGJ+85,ACG+90]. 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.

Messages transmitted over a non-faulty edge eventually arrive, or the edge will fail (in both endpoints). Messages arrive over any given edge according to the FIFO (First In First Out) discipline. The communication is asynchronous.

The complexity measures that are used to evaluate the algorithm performance are: (1) Amortized Communication - the total number of messages (each containing O$(\log V)$ bits) sent by the protocol, divided by the number of topology changes that occurred during the execution. (2) Quiescence Time - the maximum normalized time from the last input change until termination of the protocol. Normalized time is evaluated under the assumption [Awe88] that link delays varies between 0 and 1. This assumption is used only for the purpose of evaluating the performance of the algorithm, but is not used to prove its correctness.

2.2 Problem Definition

In response to *topological changes*, namely, recoveries or failures of edges, the algorithm is required to mark, at each node, a subset of the node's edges. The collection of edges marked by all of the network nodes is called, as in the algorithm of [ACK90], 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 (the real forest) is required to become eventually a spanning tree of the connected component.

3 Background

3.1 The Tree Maintenance Algorithm of [ACK90]

A failure of a real forest edge causes the edge to become unmarked and disconnects a real tree into two or more real trees. Our model allows multiply edges to fail simultaneously.

In order to span the entire connected component, the algorithm of [ACK90] uses the following scheme to reconnect the tree, until it spans the connected component. Every real tree locates its *minimum outgoing edge*- the edge with the minimum weight among all the edges leading from (nodes of) this real tree to (nodes of) other real trees. If the other endpoint is also the minimum outgoing edge of the other real tree, then it is guaranteed that the endpoints will agree . (Unless, of course, additional topological changes occur, e.g. failure of the minimum edge). Therefore, at each algorithm iteration two distinct real trees are merged into a larger real tree, by marking the minimum edge that connects them. The high level description of the algorithm of [ACK90] appears in figure 1.

Two main distributed subroutines are used in the algorithm (for the full list of the properties of two subroutines refer to section 5):(1) FIND- a subroutine which is used to find the minimum outgoing edge of the real tree using a dynamic data structure maintained by each node. (2) UPDATE- a subroutine that is used to update the mentioned dynamic data structure. The dynamic data structure at each node, v, is v's approximation of the real forest and is called v's *forest replica*. The approximation (a subset of v's forest replica) of Node v's real tree is called v's *tree replica*.

The UPDATE subroutine 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 a failure, or marking an edge that the algorithm decided to use to merge two trees) updates its forest replica, and communicates the change over the marked edges to its whole real tree. Observe that in the case that the forest replicas of the endpoints of an edge disagree, it is neither obvious how such an "agreement" is to be reached, nor which one of the endpoints is "more correct" [1] (better reflects the real forest). In order to resolve this conflict, the UPDATE subroutine is based on an idea, which was called the *Tree Belief principle* in [ACK90]. This principle is used between every two neighbors, in order to decide which of their forest replicas is "more correct" regarding every Edge e. In other words, which replica reflects a later (in history) status of Edge e. The two neighboring nodes apply the Tree Belief principle for every topological item that appears in their replicas, namely, for every edge (x, y). Consider a graph, U, which is the union of the tree replicas of nodes u and v. According to this principle, Node v is "more correct" about an edge (x, y) than its neighbor u, if in U the undirected path from u to edge (x, y) starts at Node v. (The intuition is that if Node v is not correct, an indicator for the error should have arrived, sooner or later, over the mentioned path). It is obvious that if for an edge (x, y) there appear two distinct undirected paths from v, one starting with edge (u, v), while the other does not, this principle could not have been applied. Therefore, the following invariant is enforced by the algorithm in order to realized this principle.

Definition 1. *The* Loop Freedom invariant *([ACK90]): For every real tree T, the union of the tree replicas of the nodes of T does not contain a cycle.*

[1] This is due to the asynchronous nature of network (we do not assume a real time clock) and the bounded size of the message (we cannot aloud to number all the messages using a counter that counts to infinity).

The UPDATE subroutine conducts the correction of the replicas using the Incremental Update technique.

Definition 2. *Incremental Update technique- The node with the correct data structure sends each neighbor node a message per error that appears in its neighbor's data structure. The message describes the place of the error in the data structure (thus, the neighbor can correct the error).*

This technique is based on the *Neighbor-Knowledge assumption*, namely that each node "knows" the content of the local memory of its network neighbors. This assumption holds for the algorithm of [ACK90] because the algorithm maintains in each node's local memory also an estimate of the forest replicas of its neighbors. Let Node u's *mirror* of node v (denoted by $Mirror_u(v)$) be the data structure, at Node u, that represents the estimate that Node u has for the replica of u's neighbor v.

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

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

Whenever two trees choose same min outgoing edge
    For each of the trees separately call UPDATE
    (* After UPDATE terminates: *)
    Mark the chosen edge at both of its endpoints (*merge*)
```

Fig. 1. The main Algorithm of [ACK90].

3.2 The Maintenance-of-Common Data Algorithm of [AS91]

In the *TREE MERGE* procedure, which is invoked in our algorithm's second phase (described in section 4.2), a new version of the algorithm of [AS91] is used as a distributed procedure. The purpose of *TREE MERGE* is to update the different replicas at all the nodes of all the *approved Trees* with the topological changes. In the algorithm of [ACK90] this task was conducted, as mentioned earlier, using the UPDATE subroutine. Using a somewhat modified version of the algorithm of [AS91] techniques, Procedure *TREE MERGE* improves the time complexity of our algorithm.

In this section we describe the intuition behind the original version of the algorithm of [AS91] (for full details see [AS91]). The model in [AS91] is a communication network of $n + 1$ nodes arranged in a chain, each holding an m-bit local input array. The first node in the chain (W.l.o.g the left node in the chain) is termed the *broadcaster*. The task of the algorithm is to write in the local memories of all the network nodes the value of the input of the broadcaster. The broadcaster's array is considered to be the "correct" array and the broadcaster is in charge of "correcting" the other "wrong" arrays. We use the term *thread* to denote a single invocation of the algorithm that runs on a single network node at a time (but can migrate). The distributed algorithm is built from such *threads* which can migrate from one node to the node's neighbor in a direction away from the broadcaster, namely, from a node to its right-hand neighbor.

The algorithm places one thread, the first invocation of the algorithm, in charge of the entire protocol - that is, in charge of correcting all the local arrays using an Incremental Update technique (see definition 2). Namely, this thread starts correcting every error it "knows" about in its neighbor. Only one message per error is sent. Once its neighbor's array is correct the thread moves on to that neighbor and continues its operating. However, this technique can be poor in terms of time complexity due to the fact that the Incremental Update technique puts sever restrictions on the use of pipelining. (Intuitively, only when a Node v "knows" its neighbor had the opportunity to correct on item but declined, does v "know" that the item is correct; only then can v correct the next neighbor on the chain). To improve the time complexity of the Incremental Update, the algorithm of [AS91] adopts a (very weak) version of message pipelining: each invocation of the algorithm works recursively and creates two "child" threads. Each of the child threads is in charge of correcting half of its parent's array in all the network nodes. The first child thread is in charge of correcting the lower half of its parent array and the second child is in charge of correcting the upper half of its parent array. The second child's correction messages are sent only after the first child has finished sending its correction messages, therefore, keeping the increasing bit order of the correction according to the *incremental update* technique. After a thread finishes to correct the array it is in charge off it delegates to its neighbor. Therefore, the two threads move along the network nodes relative to one another in their work. The parent thread itself tags along just behind its children. In order to correct its smaller replica, a thread runs the same protocol as its parent thread. In particular a child thread can create children that are in charge of arrays that are still smaller than its own, and so forth. Splitting the correction of an array in that manner enables the two threads to work in parallel on different nodes and therefore, improves the algorithm time complexity.

The way in which a thread creates a child threads is by sending a *thread-carrier* message to the right-hand node in the network. This message carries the indexes of the smaller array that the new child thread will be in charge of. Had each thread created child threads immediately, the message complexity of creating these threads could have been a dominant factor in the protocol's message complexity. In such a case there could be arbitrary more *thread-carrier* messages than error-correction messages. The algorithm of [AS91] avoids this problem by allowing a thread to create child threads only if it has corrected "enough" errors in its neighbor's replica. In this manner the algorithm amortized the communication costs associated with the creation of children on the communication cost of the correction messages.

4 Our Algorithm

4.1 Phase 1: Trees Approval

As mentioned earlier, one of the bottlenecks in the time measure in the algorithm of [ACK90] is due to the fact that the algorithm merges only two trees at each iteration. A worst case scenario for this algorithm is a network that is arranged in a chain and in each iteration of the algorithm a tree consisting of a single node merges with the tree in the left side of the chain until in V iterations the whole network is spanned. In this scenario the algorithm's quiescence time is the sum of an algebraic sequence- $O(V^2)$. The first phase of our algorithm, the Trees Approval, removes this bottleneck by enabling more than two trees to merge at a time, and still we manage to avoid violating the *loop freedom invariant*. In this section we describe this phase in detail. The high-level description of the first phase of the algorithm appears in figure 2. For the sake of simplicity we describe our algorithm as a sequential algorithm (the distributed implementation is described in the full paper).

Whenever a tree chooses a minimum outgoing edge it sends over that edge a *request* message, namely, a request to merge, directed at the tree over the other endpoint of the edge. This is the same technique used in the algorithm of [ACK90] (described in section 3.1). The different step taken in our algorithm is conducted whenever two trees have chosen to merge through the same outgoing edge, the *core edge*. Node r, the node with the higher identity between the core edge's endpoints, starts a broadcast wave of a *registration* message[2]. This message propagates over tree edges to all the nodes in r's real tree. Let Nodes u and v be two neighbors that belong to different trees. When the *registration* message reaches u which already got a *request* message from v, Node u marks the request as granted and propagates the *registration* message also to v through (u, v). A merging *request* that arrives at u after the *registration* message is received at u, has to wait for the next *registration* message (namely, for the next algorithm iteration). The registration message is propagated transitively to all the real trees that their *request* messages arrived before the *registration* message (originated from r). We refer to these real trees as the *approved trees*.

[2] This *registration* message is substantially different from the *initiate* message of the algorithm of [GHS83]. In the algorithm of [GHS83] Node n that sends a *connect* message to Node n' may merge with n''s tree even though n' has already received the *initiate* message- in certain cases. In our algorithm a tree always has to wait if its merging request has arrived after the *registration* message. This major difference arises from the fact that our model is dynamic. Our algorithm operates in the spirit of "two-phase commit" protocols. It first attempts to "lock" ("approve") some of the trees and only after it stops "approving" it starts allowing them to merge.

When Node r is notified (by the standard Broadcast and Echo technique- the PIF algorithm [Seg83]) that the broadcast of the registration message has terminated, it broadcasts a *start-update* message to the roots of all the approved real trees to invoke the UPDATE' a version of the UPDATE procedure of [ACK90]. This procedure is invoked separately by the root of each approved real tree T on receiving the *start-update* message. This invocation is responsible for updating T's replica as it appears at T's root at all of T's nodes.

The number of failures of edges may be arbitrarily larger than the size of a node's real tree. Therefore, in order to ensure that the number of messages that the algorithm sends is a function of the size of the connected component, a change has to be made in procedure UPDATE. We add a new update message *range deletion* that carries one topological item (u, v). This message instructs its receiver Node w to delete all the topological items that appear in w's tree replica lexicographically after the topological item that appeared in the previous *correction* message. If the *range deletion* message is the first message that the node receives, the node deletes all the topological items that appear in its tree replica lexicographically before (u, v).

When Node r detects, using the termination detection algorithm of [DS80] (the same technique is used in the algorithm of [ACK90]) that all the activation of UPDATE have terminated, it invokes the second phase of the algorithm- Trees Merge (describes in section 4.2).

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

Whenever trees merge or a topological change occurs
    Call UPDATE' (*correct tree replicas*)
    Call FIND (*choose min outgoing edge*)

Whenever two trees choose the same min outgoing edge
    Broadcast & Echo registration message (*approving trees*)
    Invoke procedure UPDATE' separately for each of the approved trees
    Invoke Trees Merge (*second phase*)
```

Fig. 2. The First Phase - Trees Approval Algorithm.

4.2 Phase 2: Trees Merge

Node r invokes the merge of all the approved trees (the last line of Figure 2) by broadcasting a *start-TREE MERGE* message that propagates to all the nodes of the approved trees. Every local root u of an approved tree T that receives *start-TREE MERGE* message initiates an invocation of the *TREE MERGE* procedure. We refer to Node u as the *broadcaster* of the procedure. Every TREE

MERGE procedure is an invocation of a new version of the algorithm of [AS91] where the "correct" replica is the tree replica as it appears in the broadcaster's local memory. The broadcaster is responsible for updating all the local replicas in all the nodes of all the approved trees. Note that there are parallel broadcasts- each broadcast is initiated by a root of an approved tree. Every message of the procedure contains its broadcaster identity, therefore, every node that receives such a message can relate it to its appropriate invocation. Each procedure invocation starts when its broadcaster, say Node u, updates in parallel all of its neighbors in u's real tree. As a part of this update process u also sends its real tree replica over Edge (u, v), over which it got the *registration* message. This is, in fact, an Incremental Update: sending the whole u's tree replica to a neighbor v when $Mirror_u(v)$ is empty.

Recall that the algorithm in [ACK90] required that u sends the whole *forest replica*, namely, sending also replicas of trees that u does not belong to them. Transmitting the real tree replicas, rather than the forest replica, violates the Neighbor Knowledge assumption- a node u no longer holds in its local memory, in $Mirror_u(v)$, an accurate copy of its neighbor v's local replica. Therefore, it is possible that later u will transmit correction messages that describe topological items that already appear in v's forest replica. Note that these sort of duplicate messages cannot be sent in the algorithm of [ACK90]. In section 5.2 we prove that these duplicates do not increase the order of the message complexity of our algorithm.

As was mentioned earlier, the TREE MERGE procedure is a version of the algorithm of [AS91]. However, weakening the neighbor knowledge assumption, that the algorithm in [AS91] relies upon, requires an algorithmic step to be taken in our procedure whenever a node receives a *correction* message. Assume that node w receives a *correction* message from its neighbor, Node z, that tells w to add an edge (x, y) to w's forest replica. Assume further that adding that edge would connect two separate trees that appear in w's forest replica. The new algorithmic step that is taken by w is to remove every edge (y, u) that appears in w's forest replica but does not appear in the mirror of Node z, $Mirror_w(z)$. When a broadcaster u, a root of an approved tree, is notified that the TREE MERGE procedure has terminated, u marks the Edge (u, v) as a real tree edge. Node u also marks Node v as its parent. (This is done only if u is not the node with the higher identity of the two endpoints of the core edge- node r). Every node in every approved tree knows the number, *DOWN-FLOW counter*, of approved trees to which it and its ancestors in its real tree propagate the *registration* message. If *DOWN-FLOW counter* equals to zero, then u sends a *termination* message to its parent notifying the parent that the TREE MERGE procedure has terminated. A node w that the number of termination messages it gets is equal to *DOWN-FLOW* sends a termination message to its parent. When Node r is notified that all the TREE MERGE procedure invocations have terminated, r starts a new iteration by an invocation of a new search for a minimum outgoing edge for the new tree.

5 Analysis

In this extended abstract we state the lemmas and theorems. The proofs are deferred to the full paper. In our proofs we use the following reworded Lemmas and properties from the algorithm of [ACK90]:

Lemma 1. *(Lemma C.1 [ACK90]) FIND subroutine terminates.*

Lemma 2. *(Lemma C.2 [ACK90]) UPDATE subroutine terminates.*

Lemma 3. *(Lemma C.3 [ACK90]) Upon the termination of UPDATE the tree replica at each node 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.*

In our proofs we use the transition system with asynchronous message passing model of [Tel94]. In order to induce a notion of time in executions we use the following *casual order*:

Definition 3. *[Lam78] Let H be an execution. The relation $<$, called the* casual order*, on the events of the execution is the smallest relation that satisfies:*
(1) if h and f are different events in the same node and h occurs before f, then $h < f$.
(2) if s is a send event and r the corresponding receive event, then $s < r$.
(3) $<$ is transitive.

5.1 Correctness

As long as the loop-freedom invariant is kept the following three lemmas are follows:

Lemma 4. *The real forest is indeed a forest at all the algorithm execution.*

Lemma 5. *The direction (parent pointer) of every real tree's edges induces only one root at every real tree.*

Lemma 6. *The real trees are disjoint.*

Lemma 7. *The algorithm's first phase, the Trees Approval phase, preserves the loop-freedom invariant.*

Proof Sketch: : During the first phase edges can only be deleted from a node's replica. ∎

Lemma 8. *The algorithm's second phase, the Trees Merge, preserves the loop-freedom invariant.*

Let u and v be two neighboring nodes in a real tree, and T_u and T_v be the tree replicas of nodes u and v respectively.

Theorem 1. *When the algorithm's second phase (the Trees Merge phase) terminates, T_u and T_v are identical.*

5.2 Amortized Communication Complexity

The following definition attempts to capture the "maximum information" a node may have on the actual connected component it belongs to. That is, if a node z receives a message that was originated by some far away node x, and every node on the message's way (from x to z) forwarded the message before any fault was detected, then x *may* still be in Node z's connected component. There is no distributed algorithm that can detect that this is not the case. Thus the message complexity of any algorithm must be based on the assumption that x and z may still be in the same connected component.

Definition 4. *Edge* (x, y) *is a* z-virtual directed edge *if (1) there is a causality chain of* send *and* receive *events ([Lam78]) from Node* x*, through Node* y*,* (x, y)=(w_1, w_2)*,* (w_2, w_3)*,* (w_3, w_4)*, ...* (w_{l-1}, w_l)=(w_{l-1}, z) *to Node* z*, and (2) between every receive event (on this chain)* (w_{i-1}, w_i) *and the following (on this chain) send event* (w_i, w_{i+1}) *there is no event at Node* w_i *of a failure of edge* (w_{i-1}, w_i)*.*

Definition 5. *Node* z*'s* Virtual Component*: the set of* z*-virtual directed edges.*

Definition 6. Physical Component*: The set of nodes that from a global point of view ([Lam78]) of the network are in the same connected component.*

Lemma 9. *Every node that appears in Node* u*'s tree replica is also in* u*'s* Virtual Component*.*

Lemma 10. *The size of* u*'s tree replica is bounded from above by the size of* u*'s* Virtual Component*.*

Proof Sketch: : The lemma follows from lemma 9. ∎

Let A_k be Node k's *Virtual Component.*

Lemma 11. *For every execution there exists an equivalent execution ([Lam78]) where Node* k*'s physical component is* A_k*.*

Theorem 2. *Every algorithm that maintains, in a node, a replica of the node's connected component (if the input stops changing then the output, the node replica, is required to become eventually the node's connected component) cannot maintain a smaller replica than our algorithm's tree replica.*

From Theorem 2 it follows that defining an algorithm measures as a function of the virtual component size is the most accurate measure according to which a distributed algorithm can be measured.

Define A as the union of the trees replicas of all the nodes in the same physical component when the last topological change occurs.

Theorem 3. *Assume that* k *topological changes occur. Then the number of edges identities exchanged during the algorithm execution is* $O(kA)$*.*

5.3 Quiescence Time

Lemma 12. *Every invocation of procedure TREE MERGE terminates.*

Lemma 13. *Every edge failure is unmarked in at most one of a given tree node's replica.*

Proof Sketch: : An edge failure disconnects the real tree and the two edge's endpoints become nodes in distinct real trees. ▮

Lemma 14. *The quiescence time of the algorithm's first phase, Trees Approval Algorithm, is $O(A)$.*

Lemma 15. *The quiescence time of the second phase of the algorithm- Trees Merge, is $O(A \log^2 A)$.*

Theorem 4. *The algorithm quiescence time is $O(A \log^3 A)$.*

6 Acknowledgments

We would like to thanks Israel Cidon for helpful discussions.

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.

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.

ACK90. Baruch Awerbuch, Israel Cidon, and Shay Kutten. Optimal maintenance of replicated Information. Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science (FOCS 90), St. Louis, MO, USA, pp.492–502, October 1990.

AS91. Baruch Awerbuch and Leonard J.Schulman. The maintenance of common data in distributed system. In Proc. 32nd IEEE Symp. On Foundations of Computer Science, October 1991.

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.

CGKK95. Israel Cidon, Inder Gopal, Mark Kaplan, and Shay Kutten. Distributed Control for Fast Networks. IEEE Transactions on Communications,Vol. 43, No. 5, pp. 1950–1960, May 1995.

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

GHS83. R.Gallager, P.Humblet, and P.Spira. A distributed algorithm for minimum weight spanning trees. ACM Transaction on programming language and Systems, 4(1):66-77, January 1983.

Lam78. Lamport, L. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21 (1978), 558-564.

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.

OSPF. J.Moy. OSPF Version2 RFC1247, October 1991.

Seg83. A. Segall. Distributed network protocols. IEEE Transaction on Information Theory, IT-29(1):23-35, January 1983.

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

Tel94. Gerard Tel. Introduction to Distributed Algorithms. Cambridge University Press, 1994.

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