OPTIMAL FAULT-TOLERANT DISTRIBUTED SPANNING TREE WEAK CONSTRUCTION IN GENERAL NETWORKS

by

Shay Kutten

Technical Report #432

August 1986
Optimal Fault-Tolerant Distributed Construction
of a Spanning Tree

Shay Kutten

Department of Computer Science
Technion, Haifa 32000
Israel

ABSTRACT

We study the basic problem of constructing a spanning tree distributively in an asynchronous general network, in presence of faults that occurred prior to the execution of the construction algorithm. Failures of this type are encountered, for example, during a recovery from a crash in the network. This problem is fundamental in computer communication networks, e.g., for routing, and as a subroutine for other distributed algorithms. Since we do not assume a global knowledge about the network, no fault-resilient algorithm can guarantee termination. Thus we investigate the border area between the possible and the impossible. We present for the first time an optimal (in terms of the message complexity) algorithm which eventually constructs a spanning tree (but does not detect the termination) in spite of faults in general networks. Although our algorithm is faults-resilient, the order of the number of messages it uses is the same as that required by a non-resilient algorithm. For a network with \( m \) communication lines and \( n \) processors, \( k \) of which start the algorithm, the algorithm we present uses at most \( O(n \log k + m) \) messages. Another main contribution of this paper is the method by which we have modified an existing algorithm for a fault-free network.
1. INTRODUCTION

The problem of a spanning tree construction in asynchronous distributed systems is widely studied (e.g. [GHS83, KKM85, KMZ83]). In this problem it is required that the nodes cooperate to construct a spanning tree, the root of which will know that it is the root, and any other node will know the edges leading to its father and sons in the tree. The spanning tree construction problem is fundamental in the area of computer communication (e.g. for routing), and in many distributed problems ([GHS83]) and have been studied by many researchers with respect to various models and cost measurements in reliable networks.

Real systems, however, are not reliable [FLP85].

1.1. The Basic Model

The basic model (before considering possible faults) consists of a distributed general network of \( n \) identical processors (except for their having distinct identities- denoted ids), \( k \) of which start the algorithm. The network includes \( m \) communication lines, each connecting a pair of processors. The network is asynchronous (the time to transmit a message is unpredictable). The processors all perform the same algorithm, that includes operations of (1) sending a message over a communication line, (2) receiving a message from a pool of unserviced messages which arrived over communication lines, and (3) processing information locally. Any subset of the processors may start the algorithm. We view the communication network as a general undirected graph, where nodes represent processors and edges represent communication lines. To measure the efficiency of an algorithm, we use the common measure of the maximal possible number of messages transmitted during any execution, where each message contains at most \( O(\log \text{MaxId}) \) bits (see e.g. [GHS83]).

1.2. The Model with Faults

Now consider the possibility that some nodes in the network may be faulty. We investigate fail-stop failures: a faulty node is a node which never transmits a message, and every message transmitted to it is lost. Because our basic model is asynchronous, it cannot be distinguished whether a message is delayed or lost. The presence of these faults motivates the spanning tree construction, as a method to reorganize the network. (Reliable hardware equipment makes failures of the most general type quite rare [G82]. Also, failures of more general types are hard or impossible to solve [F83, FLP85].)
1.3. Previous Results

In [KW85] an optimal algorithm was constructed, to elect a leader and construct a spanning tree in a complete network where at most $t$ processors (or communication communication lines) may have failed before the execution of the algorithm. The algorithm is optimal for any $t < \frac{n}{2}$ (for a larger $t$ the election problem cannot be solved by any algorithm). In [BK86] the results of [KW85] were generalized and an optimal spanning tree construction algorithm was presented for complete networks in which any number of communication lines and nodes may have failed. When a leader election is possible, this algorithm also elects a leader. However, when the failures do not leave a non-faulty connected component of more than half the nodes, a leader cannot be elected. The algorithm of [BK86] can also be applied in general networks, when the number of nodes is known.

1.4. Results in this Paper

In this paper we continue to investigate what can be done on the border of the impossibility. For this kind of faults, (fail-stop faults which occurred prior to the execution of the algorithm) we carry the investigation to general undirected networks. Even with these assumptions about faults, it is obvious that no fault-tolerant leader-election algorithm can guarantee termination in the case of a general graph (if no global information, i.e. the number of nodes, is known).

We develop for the first time an optimal fault-resilient algorithm (optimal in the number of messages it sends). Using this algorithm we show that the message complexity of fault resilient spanning tree construction is not higher than the complexity of a non-fault-resilient algorithm, i.e. $O(n \log k + m)$. However, we believe that another main contribution of this paper (together with [KW85], and [BK86] it generalizes) is the systematic conversion of algorithms which makes them fault-resilient.

Open problems are given in the last section.

1.5. The Problems Facing the Designing of Fault Resilient Algorithms

In distributed spanning tree construction algorithms, trees are merged to eventually span the whole network. Every tree has a unique temporary leader (represented by a token), that decides on the tree expansion or merging. This method seems essential to prevent contradictory decisions (like two trees merging via two edges and creating a circle). This temporary leader tries to expand the tree in exactly one direction, both in order to ensure correctness and to save messages. From the correctness point of view it seems that expanding
in more than one direction can create a cycle, or cause two trees to be deadlocked (waiting for each other’s response). From the complexity point of view it seems that if actions corresponding to the two directions do contradict, at least one of them has caused redundant work.

In the presence of undetectable faults the above approach can cause the algorithm to be deadlocked, waiting for an answer from a faulty node, or over a faulty edge. If expansion, however, is tried only over an edge which has been proved to be non-faulty, high complexity can be caused, since it may happen that the next edge to be proved non-faulty is always far from the temporary leader of the tree.

A convenient method to design and describe distributed algorithms is the tokens method [KKM85]. A token is a process which can move from a node to its neighbor, carried by a message. In a spanning tree construction algorithm, a node in which a token currently resides, is the temporary leader. An algorithm described by defining the actions of tokens, can easily be translated to an algorithm that is described by defining the actions taken by node.

For unreliable networks, the tokens method seems inconvenient, since a token sent over a faulty edge (or to a faulty node) is lost, and the algorithm may deadlock.

1.6. Making Algorithms Fault Resilient

In presence of faults we choose to use more than one token in a tree. Most decisions are done locally (without consulting the temporary leader), and contradicting actions are prevented by the following method: each token acts according to the global information it has about the state of the tree. In this paper this information is a value from an ordered set. The true value is always equal or higher than the value known to the token. Thus the temporary leader must be consulted only when this information alone does not suffice to make the decision. This information is the identity of the temporary leader of the tree and a bound on the generation (which is equivalent to the level in [GHS83], or the phase e.g. KKM85) of the tree. (Other kinds of information are used in [BK85] and [KW85].)

Using this method we modify the existing election algorithm of [KKM85] (and [K85]) to achieve our faults-resilient spanning tree construction algorithm.

2. CONSTRUCTING A SPANNING TREE IN GENERAL NETWORKS

In this section we present for the first time an optimal algorithm to construct a spanning tree in presence of faults in a general network. As explained in Section 1, termination detection cannot be guaranteed since no
global knowledge is assumed. In particular, we do not assume that $n$, the number of nodes, is known to any node. We term this the weak spanning tree construction problem. (This problem was independently investigated by [AG85].)

We make the algorithm of [KKM85] (and [K85]) resilient and optimal. Every node initially sends test messages over all its edges. Thus if an edge is not faulty, its endpoints will eventually know it, and mark the edge non-faulty. Each edge may receive the marks: non-faulty, tree-edge-to-father, tree-edge-to-son, and deleted. Initially no edge is marked. Every node is in a rooted tree, initially containing only itself. Each tree has an associated value called generation (or phase), which is initially 0, and is set to 1 if its root node starts the algorithm spontaneously. Each tree has also a single main token residing usually in the root. Main tokens of trees are used to create new trees, so that the creation of a new tree (main token) in generation $g+1$, involves the destruction of at least two main tokens in generation $g$. Since each tree has only one main token, the number of trees of generation $g+1$ is at most half the number of trees of generation $g$.

Each node in a tree generates a local token, the task of which is to annex the node’s neighbors to the node’s current tree. Several local tokens of a tree $T$ may annex in parallel nodes to the tree. At the same time, local tokens of other trees may annex nodes of $T$ to the other trees. Each local token always carries the correct tree’s generation, and root, since both values never change (until the node joins another tree). The tree information that the local token may not have is the answer to the question “does the main token still exist?” Thus the local token must consult the tree root only when it seems likely that a tree of a higher generation can be created (i.e. when the local token enters a node that belongs to another tree of the same generation).

We now outline the algorithm. A more detailed description appears in the sequel.

the main token of each tree waits in the tree root until it is waken by a local token of the same tree. The local token of a node, $v$, tries to annex $v$’s neighbors to $v$’s current tree. To annex a neighbor, the local token travels to the neighbor, carrying the generation of $v$’s tree, and the identity of its root. If the generation of the neighbor’s tree is lower, then it is annexed as $v$’s son. If the generation of the neighbor’s tree is higher, then $v$’s local token is destroyed. If the generations are equal (and the neighbor belongs to another tree) then a meeting must be arranged between the main tokens of the two trees. The local token arranges that meeting only if the $id$ of its root is higher than the $id$ of the other tree’s root. To arrange the meeting it goes to its tree root and wakes the main token. Waking main token travels to the other tree root. If it arrives and finds the other main token, then the two main tokens are destroyed, and a main token of a higher generation is created.

Note that at first only one node belongs to the new tree. The two old trees may still exist, and even grow.
Eventually, only one tree remains in every connected component.

Some care must be taken in implementing the algorithm, to limit its message complexity, and still not violate its correctness. Edges which are found by local tokens to create cycles are deleted. Also, local and global tokens which find that other tokens of the same generation (or a higher generation) have preceded them, stop.

We now describe the algorithm in more details.

The Algorithm of a Local Token

(L.0) Let \((v,u)\) be an edge which leads from node \(v\) and which is already known to be non-faulty, but is not marked deleted or tree-edge-to-father or tree-edge-to-son. If no such edge exists then the local token waits until there is such an edge. (Note that it may wait forever.) Otherwise, the local token traverses edge \((v,u)\) to its other endpoint, \(u\). We now describe the following actions taken, according to the generation \(g_v\) and the tree root \(r_v\) of \(v\) (information carried in the local token), compared to the generation \(g_u\) and tree root \(r_u\) of node \(u\), and according to the marking of edge \((u,v)\).

Case (L.1) below is the case where \(g_v < g_u\). Case (L.3) is the case where \(g_v > g_u\), and edge \((u,v)\) is not marked deleted in \(u\). In Case (L.4) \(g_v = g_u\), but \(r_v \neq r_u\), and edge \((u,v)\) is not marked deleted in \(u\). Case (L.2) is the case where \(g_v \geq g_u\) but either edge \((u,v)\) is marked deleted in \(u\) or \(r_v = r_u\).

(L.1) \(g_v < g_u\):

Node \(v\)'s local token is destroyed, i.e. \(v\) stops trying to annex neighbors but does not notify any other node in its tree. (A new local token will be created in node \(v\) when \(v\) will be annexed by another tree.)

(L.2) \(g_v = g_u\) and nodes \(u\) and \(v\) belong to the same tree, or \(g_v \geq g_u\) and edge \((u,v)\) is marked deleted in node \(u\):

The local token marks edge \((u,v)\) deleted in \(u\) (in case it is not already marked deleted). Then it returns to node \(v\), and acts according to \(v\)'s generation:

(L.2.1) The generation of node \(v\) has increased since the local token has left it:

The local token is destroyed.

(L.2.2) The generation of node \(v\) is the same as the one carried in the token:

The token marks edge \((v,u)\) deleted.
(L.3) \( g_v > g_u \) and edge \((u,v)\) is not marked deleted

in node \( u \): Node \( u \) leaves its tree and becomes \( v \)'s son. It removes the old tree-edge-to-father and tree-edge-to-son marks (if such exist), marks the edge \((u,v)\) as the tree-edge-to-father, and creates a new local token for the new tree. If the old local token is waiting in \( u \) then it is destroyed. (Otherwise, if not already destroyed, the old local token of node \( u \) is destroyed on entering to node \( u \), since \( u \) now belongs to a higher generation tree.) Node \( v \)'s local token then returns to \( v \), and acts according to \( v \)'s generation:

(L.3.1) The generation of node \( v \) has increased since the local token has left it:

- The local token is destroyed.

(L.3.2) The generation of node \( v \) is the same as the one carried in the token:

If edge \((v,u)\) has not meanwhile been marked deleted then it marks edge \((v,u)\) as a tree-edge-to-son. (Otherwise \((v,u)\) stays deleted.)

(L.4)

Edge \((v,u)\) is not marked deleted in \( u \) and \( g_v = g_u \) but nodes \( v \) and \( u \) belong to two distinct trees: The main tokens of the two trees are destroyed, and a tree of a higher generation is formed. To do this the algorithm tries to arrange that the main tokens of the two trees meet in one of the roots, so that they can be destroyed at the same time a higher generation tree (main token) is created. This is the only operation which is not entirely local. The local token's next operation depends on \( r_v \) compared to \( r_u \):

(L.4.1) \( r_v < r_u \):

\( v \)'s local token is destroyed.

(L.4.2) \( r_v > r_u \):

Node \( v \)'s local token goes to the root \( r_u \) to fetch the single main token representing \( v \)'s tree.

The route to the root \( r_u \) is found by the local token, by first returning to node \( v \), and then using in each node the edge marked tree-edge-to-father. Each node; \( w \), which forwards the local token on this route records local.token.path\((g_v)\) which is the edge over which the local token has entered node \( w \).

(L.4.2.1) If on its way to the root the local token enters a node which belongs to a higher generation tree, then the local token is destroyed.
(L.4.2.2) Also, if the local token enters a node passed before by another local token of the same generation, then the local token is destroyed. (The local token is no longer needed since the meeting of the main token with another main token is already taken care of by the other local token.)

(L.4.2.3) If the local token arrives at the tree root and the main token is still there, then the main token wakes up, and the local token is destroyed. (See the algorithm of the main token.)

A local token that returned safely to node \(v\) performs (L.0) again.

The main token acts as follows:

**The Algorithm of a Main Token**

(M.0) When the main token is woken by a local token of some node \(v\) which returned from \(v\)'s neighbor \(u\), it is routed to the root of \(u\)'s tree. To find this route, the main token first uses the edges recorded as `local.token.path(g_u)` to arrive at node \(u\), and then, from each node in \(u\)'s tree, the edge to the node's father (`tree-edge-to-father`).

(M.1) If on its way the main token enters a node which belongs to a higher generation tree, or has been visited by another main token of the same generation, then the main token is destroyed.

(M.2) If the main tokens of the two trees meet (in node \(r_u\)), then a new tree of higher generation is created, including initially only \(r_u\). Note that the two trees of the lower generation may still continue to grow, annexing other nodes. However their main tokens are destroyed, and thus they cannot create another tree of a higher generation. Eventually all their nodes will join trees of higher generations.

3. CORRECTNESS PROOFS AND COMPLEXITY ANALYSIS

The theorem at the end of this section summarizes the correctness proof and the complexity analysis. Lemmas 1 and 2 prove that the deletion of edges does not disconnect the graph. Lemmas 3 and 4 ensure that a progress is made by the algorithm as long as it is performed in a connected graph. Lemmas 5, 6, and 7 are used in the complexity analysis.
Definition: we say that edge \((v, u)\) belongs to a tree \(T\) at some time, if the following holds at that time: (1) One of its endpoints is marked \(\text{tree-edge-to-father}\) and the other is marked \(\text{tree-edge-to-son}\); and (2) this marking were made by local tokens of tree \(T\).

Lemma 1:
(a) An edge, \((v, u)\), which belongs to a tree \(T\) of some generation \(g\), cannot be marked deleted by a local token of a tree of a lower generation.
(b) Neither can edge \((v, u)\) be marked deleted by a local token of another tree of the same generation, \(g\).

Proof:
(a) By (L.2) in order that edge \((v, u)\) can be marked deleted by a local token of a tree \(T_1\), of a lower generation \(g_1\), the endpoints of this edge were first annexed by local tokens of tree \(T_1\). Since edge \((v, u)\) belongs to \(T\), both \(v\) and \(u\) were annexed also by tokens of \(T\) (see (L.3)). By (L.1) and the assumption that \(g > g_1\), the endpoints must have been annexed first by tree \(T_1\) and only later by tree \(T\). Let \(l_1\) be the local token of tree \(T_1\) which marked edge \((v, u)\) deleted. Without loss of generality assume that local token \(l_1\) was generated at node \(v\), sent to node \(u\), and marked edge \((v, u)\) deleted. By (L.1) it must have arrived at node \(u\) before node \(u\) was annexed by tree \(T\). Thus edge \((v, u)\) was already marked (at least in node \(u\)) deleted when node \(u\) was annexed by tree \(T\). Hence, by (L.0), (L.2), and (L.3), it could not have become a tree edge in-tree \(T\). A contradiction to the definition of edge \((v, u)\) as tree edge in tree \(T\).
(b) The proof is similar to that of (a).

Definition: Let \(G'\) be the subgraph of \(G\), containing all the non-faulty nodes, and all the non-faulty edges connecting these nodes.

Lemma 2: The removal of the edges which are marked deleted in an execution of the algorithm, does not disconnect any connected component of \(G'\).

Proof: Assume the contrary, and remove every deleted edge. Let \(R\) be the set of deleted edges, the two endpoints of each are not connected by a path of non-deleted and non-faulty edges. For every generation \(g\) let \(R_g \subset R\) include the edges in \(R\) the marking of which was done by a local token of generation \(g\). (If the two endpoints of an edge were marked deleted by two local tokens, we consider here only the first local token to mark it.) Let \(g\) be the highest \(g\) such that \(R_g\) is not empty and let edge \((v, u)\) be in \(R_g\). We now show that there is a path of non-deleted and non-faulty edges connecting \(v\) and \(u\). A contradiction to the definition of \((v, u)\).
Edge \((v,u)\) was marked *deleted* when a local token of a tree \(\overline{T}\) of generation \(g\) found that nodes \(v\) and \(u\) belonged to tree \(\overline{T}\). Recall that while trying to annex (see (L.0)), local tokens are not sent over an edge which belongs to their tree. This, together with (L.2), implies that edge \((v,u)\) did not belong to tree \(\overline{T}\). Thus there existed a non-faulty path (of tree \(\overline{T}\)'s edges) between nodes \(v\) and \(u\), not passing edge \((v,u)\). This path could have been disconnected only if some edge \((w,x)\) on this path has been marked *deleted* by a local token of some tree \(T_1\) of generation \(g_1\). By Lemma 1, \(g_1 > g\). Thus, by the definition of \(\overline{g}\) and Lemma 1, there exists a path of non-*deleted* and non-faulty edges between every two nodes which once belonged to \(T_1\), including nodes \(w\) and \(x\). Thus the removal of edge \((w,x)\) does not disconnect node \(v\) from \(u\). A contradiction.

Lemma 3: Once a main token of generation \(g\) is created in some node \(i\) in a maximal connected component \(G''\) of \(G'\), one of the following must eventually occur. Either every node in \(G''\) is annexed to the tree of this main token, or a local token of this tree enters a node which belongs to another tree of the same or higher generation.

Proof: The proof follows from Lemma 2 and from (L.0) and (L.3).

Lemma 4: If there is more than one main token at a certain generation \(g\), in a maximal connected component of \(G'\), then a main token at generation \(g+1\) is eventually created.

Proof: Assume the contrary. Let \(G''\) be a maximal connected component of \(G'\). Then there is a generation \(g\) such that there are at least two main tokens at generation \(g\) in \(G''\), but no main token at generation \(g+1\) is ever created. We shall show that this is impossible.

Denote a main token of generation \(g\), created in node \(i\), by \((g,i)\). Let \((g,i)\) be a main token in generation \(g\) with the maximum possible \(i\) in \(G''\). Since \(g\) is the highest generation in \(G''\), no local token of \((g,i)\)'s tree can enter a node which belongs to a tree of a higher generation. Also, by the assumption that there is another tree of the same generation in \(G''\) and by (L.4), not every node in the \(G''\) is annexed by the tree of main token \((g,i)\). Thus, by Lemma 3, a local token, \(l\), of \((g,i)\)'s tree eventually enters a node, \(u\), which belongs to another tree of the same generation \(g\). By (L.4.2) and the maximality of \(i\), the local token is then routed to node \(i\). Since \(g\) is the largest generation, the local token cannot be stopped in some node, \(w\), due to a token of a higher generation which passed in node \(w\) (see (L.4.2.1)). Thus if the local token has stopped in node \(w\) on its way to node \(i\), it is due to some other local token of the same tree which passed there on its way to \(i\) (see (L.4.2.2)). Since the local tokens are routed to \(i\) over a tree, and by Lemma 1, some local token of this tree must arrive at node \(i\).
Without loss of generality assume that local token \( l \) was the first to arrive at node \( i \). Main token \((g,i)\) is then routed to node \( u \) (by reversing the path taken by local token \( l \), see (M.0)). From node \( u \) main token \((g,i)\) is routed to \( r_u \), the root of \( u \)'s tree. Since \( g \) is the largest generation, main token \((g,i)\) cannot be stopped in some node, \( w \), due to a token of a higher generation which passed in node \( v \) (see (M.1)). Thus if main token \((g,i)\) has stopped in node \( w \) on its way to node \( r_u \), it is due to some other main token of the same generation which passed there on its way to \( r_u \) (see (M.1)). Since the main tokens are routed to node \( r_u \) over a tree, and by Lemma 1, some main token of generation \( g \) must arrive at node \( r_u \). Without loss of generality assume that main token \((g,j)\) was the first to arrive at node \( r_u \).

We have so far established the fact that some tree root of generation \( g \) was visited by a main token of another tree. Let \( h \) be the smallest id among all the roots of tree of generation \( g \), which were likewise visited by a main token of another tree of generation \( g \), By (L.4.1) the main token \((g,h)\) could not have been routed to visit a root of another tree. Thus, and by the maximality of \( g \), main token \((g,h)\) was still in node \( h \), when another main token of the same generation visited node \( h \). This, together with (M.2) implies that the two main tokens were destroyed, and a main token of a higher generation was created, contrary to the assumption that \( g \) was the highest generation in \( G'' \).

Lemma 5: The number of generations is bounded by \( \log k + 1 \), where \( k \) is the number of nodes which started the algorithm spontaneously.

Proof: The number of distinct main tokens of generation \( g \) created in an execution of the algorithm is at most \( k \cdot 2^{g-1} \). This is clear from the fact that any main token at generation \( g > 1 \) is created by destroying two tokens at generation \( g-1 \) (see (M.2)). The lemma follows.

Let us now divide the messages sent by the algorithm to three categories:

- **Faults-testing:** these are the messages broadcasted by every node, when it starts its part in the algorithm, to enable its neighbors to detect the non-faulty edges.

- **Local annexing:** these are the messages of the local tokens, except for the voyage of a local token to its tree root to fetch the main token.

- **Main meeting:** these are the messages used by the local tokens to fetch the main tokens, and the messages used by the
main tokens to go and visit a root of another tree of the same generation.

Lemma 6: the number of messages of the "main meeting" category is bounded by $3n$ for each generation.

Proof: In order to arrive at its tree root, a local token must first return to its generating node. This uses one message for each local token, i.e. at most the size of the tree. The total number of messages used by the local tokens at generation $g$, on their way from their generating nodes to the tree root is smaller than the size of the tree. A main token which visits another tree's root is routed first on a path in its own tree, and then on a path which belongs to another tree of the same generation. It is easy to see that any two trees of the same generation are nodes disjoint. The lemma follows.

Lemma 7: The number of messages of the "local annexing" category is bounded by $4|E| + 5n \log k + O(n)$.

Proof: A local token which has annexed a node, $u$, has used 2 messages, and caused node $u$ to increase its generation. Thus, by Lemma 5 the number of such messages is bounded by $2n \log k + O(n)$. A token which did not succeed to annex node $u$, as node $u$ belonged to a higher generation, has used 1 message to arrive at node $u$ and be destroyed. Thus the number of such messages is bounded by the number of local tokens. The latter however is bounded by Lemma 5 by $n \log k + O(n)$. If a local token which marked an edge deleted and returned to its generating node, $\nu$, has arrived at node $\nu$ before another local token was sent by node $\nu$ over this edge, then no other token will ever be sent from node $\nu$ over this edge. Thus the number of such messages is bounded by $4|E|$. If, however, another local token had been sent over an edge by node $\nu$ before $\nu$'s previous local token returned to node $\nu$, then the other local token must have been of a higher generation. Thus the returning local token is destroyed, and the number of such messages is bounded by twice the total number of local tokens. The lemma follows.

Theorem: Let $G$ be an undirected graph, possibly containing faulty nodes and edges. Let $G'$ be the subgraph induced by the non-faulty nodes of $G$, such that all the faulty edges are removed. The above algorithm weakly constructs a spanning tree in any connected component of $G'$, using $O(m + n \log k)$ messages.

Proof: The correctness part follows from Lemmas 1 to 4. The complexity part follows from lemmas 5 to 7.

Q.E.D.

There is a more delicate analysis of the algorithm's complexity, which yields a smaller constant factor. The constant factor can be further reduced, if one is willing to somewhat complicate the algorithm.
4. OPEN PROBLEMS

If some global information is known, then the termination of the algorithm can be detected. Such global information can be e.g. the number of nodes, distinct names for the edges, etc. It is interesting to find tight upper and lower bounds for termination detection in such cases. It is especially interesting to find out whether the message complexity of this task too is not higher than in reliable networks.

REFERENCES


[K85] Kutten, S., Constructing a Spanning Tree in General Networks using \(3n \log k + 2m + O(n)\) messages, unpublished.

