Information Processing Letters 27 (1988) 299–307
North-Holland

13 May 1988

# OPTIMAL FAULT-TOLERANT DISTRIBUTED CONSTRUCTION OF A SPANNING FOREST *

Shay KUTTEN **

*Department of Computer Science, Technion, Haifa 32000, Israel*

Communicated by T. Lengauer
Received 1 April 1987
Revised 20 December 1987

We study the basic problem of constructing a spanning tree distributively in an asynchronous general network, in the 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. In the case the network has been partitioned we construct a spanning forest. This problem is fundamental in computer communication networks, for example for routing, and as a subroutine for other distributed algorithms. Since we do not assume that a node in the network has any global knowledge about the network, no fault-resilient algorithm can guarantee termination detection. We present for the first time an optimal (in the order of message complexity) fault-resilient spanning forest constructing algorithm for general networks. The algorithm eventually constructs a spanning tree in every component of the network that remained connected in which at least one node initiated the algorithm. Although our algorithm is fault-resilient, the order of the number of messages it uses is the same as that required by a nonresilient algorithm. For a network with $m$ communication lines and $n$ processors, $k$ of which initiate the algorithm spontaneously, the algorithm we present uses at most $O(n \log k + m)$ messages. Another major contribution of this paper is the approach we took in order to modify an existing tree construction algorithm for fault-free networks, to make it fault-resilient.

*Keywords*: Distributed algorithm, spanning tree, fault tolerance, asynchronous, network

## 1. Introduction

The model under investigation is a network of $n$ processors with distinct totally-ordered identities, and $m$ bidirectional communication lines connecting pairs of processors. Every line has two ports, one in each of the processors it connects. The ports in a processor form an array. A node that wishes to send a message puts it in one of its ports, and the message eventually arrives at the other side of the corresponding line. The network is *asynchronous*: the time to transmit a message between two adjacent nodes is unpredictable. The processors are identical in the sense that all processors working on some common task execute the same algorithm. Such an algorithm may include the operations of: (1) sending a message, (2) receiving a message, and (3) processing information in the local memory.

It is assumed that all the processors are initially 'asleep' (i.e., do not execute the algorithm) and that any nonempty set of $k$ processors may be awakened spontaneously (asynchronously) and initiate the algorithm; a processor that is not an initiator remains asleep until a message reaches it. A node may initiate the algorithm (or join the execution) at most once per execution of the algorithm.

* A preliminary version of the contents of this paper was presented at the *4th Ann. Symp. on Theoretical Aspects of Computer Science*, Passau, Fed. Rep. Germany, February 1987.

** Current affiliation: IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

We view the communication network as an undirected graph, where nodes represent processors, and edges represent communication lines. A *faulty edge* is an edge that does not transmit messages in either direction. A *faulty node* is modeled by a node all edges of which are faulty. In this paper, any number of nodes and edges may be faulty. Since our basic model is asynchronous, a node cannot distinguish whether a message sent through an edge is delayed or lost.

To measure the efficiency of a distributed algorithm, we use the accepted measure of the maximal possible number of messages transmitted by all nodes, where the number of bits in each message is logarithmic in the highest identity of a node in the network (see, e.g., [10]).

In the problem of spanning tree construction it is required that each nonfaulty node will eventually construct a list of some of its ports so that the collection of all the corresponding edges (in all nodes) forms a spanning tree. If nodes $v$ and $u$ are nonfaulty and $v$ considers edge $(v, u)$ a part of the tree, then $u$ must also consider this edge a part of the tree. Spanning trees are used for routing messages in communication networks, for efficient broadcast, for consultation among the computers, and for other distributed algorithms [7,8]. Being a basic tool in computer networks, their construction is a part of the network's self-organization. Their construction has been studied by many researchers with respect to various models and cost measures in reliable networks.

Real systems, however, are not reliable [6].

In the presence of faults, the network may be partitioned into several connected components. A *spanning forest* contains a spanning tree of every component in which at least one node initiated the algorithm spontaneously.

In networks where nodes may fail at any time during execution of a protocol, the election and consensus tasks (tasks related to spanning tree construction) are impossible [6,13]. Other types of failure are also hard or impossible to tackle [5,12,4]. In [11], an optimal algorithm was constructed to elect a leader and construct a spanning tree in a complete network where at most $t$ processors (or $t$ communication lines) may have failed before the execution of the algorithm. It is optimal for any $t < \frac{1}{2}n$ (for a larger $t$, the election problem trivially cannot be solved by any algorithm).

In [3], the results of [11] were generalized for the case when any number of nodes and communication lines may have failed. In that paper [3], an optimal spanning tree construction algorithm was presented for complete networks. When the number of nodes is known, the algorithm can also be applied in general networks. When a leader election is possible, this algorithm also elects a leader. However, leader election is possible only if the nonfaulty nodes and edges form a nonfaulty connected component of more than half the number of nodes.

We construct a spanning forest. However, recall that the faults are undetectable and that the topology and size of the network are not known. A message that has not been answered may have been lost. On the other hand, it may be slow so a reply is on its way. Thus, the termination of the algorithm may not be detectable. An algorithm which constructs a spanning forest, but does not detect its own termination is said to *weakly construct a spanning forest*. [1]

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 forest construction is $O(n \log k + m)$. In [10] it is shown that any global algorithm in a reliable network uses $\Omega(m)$ messages in the worst case. Also, Korach et al. [10] show that even when it is assumed that the graph is complete, $\Omega(n \log k)$ messages are used in the worst case by any global algorithm. Thus, the order of the message complexity of the fault-resilient algorithm presented in this paper is not higher than that of optimal nonfault-resilient algorithms.

Another main contribution of this paper (together with [11] and [3]) is the systematic conversion of algorithms which makes them fault-resilient. We make the algorithm of [9] fault-resilient. In the algorithm of [9], each tree root uses a distributed serial graph search—a traversal. In

[1] This problem has been independently investigated by Awerbuch and Goldreich [2]. Recently, another algorithm for this purpose was announced by Afek and Saks [1].

this paper, the process of annexing nodes to a tree is done in such a way that many nodes may join the same tree in parallel.

Finally, open problems are given in Section 4.

## 2. The algorithm

Let us now make the algorithm of [9] fault-resilient. Some definitions and preliminaries are now given. Initially, no edge is marked. Every node that enters the algorithm sends *test* messages over all its edges. Thus, if an edge is not faulty, its endpoints will eventually know it, and mark the edge *nonfaulty*. Each edge may receive the following marks in each of its endpoints (or in both): *nonfaulty, tree-edge-to-father, tree-edge-to-son*, and *deleted*. Every node is initially in its own *domain*. As long as no node of a domain has been annexed by another domain, the domain is a rooted tree. We refer to this root as the root of the domain. When a node of a domain $T_1$ is annexed to another domain, domain $T_1$ may become a directed forest. Each domain has an associated value called *phase*, which is initially $-1$, and is set to 0 if its root initiates the algorithm spontaneously. (The domain of a node which is not an initiator includes only that node, and only until the node is annexed to another domain.)

Each domain also has a single *main* token which usually resides in the root. Main tokens of domains are used to create new domains in such a way that the creation of a new domain with a new main token in phase $p+1$ involves the destruction of at least two main tokens in phase $p$. Thus, the number of domains in phase $p+1$ is at most half the number of domains in phase $p$.

Each node in a domain $T_1$ generates a *local* token, whose task is to annex the node's neighbors to the domain. The local token is considered to belong both to its creating node and to domain $T_1$, and it stops if it finds out that its creating node was annexed to another domain. (When the node is annexed, a new local token, which belongs to the new domain, is created in the node.) Several local tokens of a domain $T_1$ may simultaneously annex nodes to $T_1$ (by a method to be explained). At the same time, local tokens of other domains may annex nodes of $T_1$ to the other domains.

Each local token always carries $p$, the phase of its domain, and $i$, the identity of the domain's root. Thus, a local token must consult the domain's root only when the local token arrives at a node that belongs to another domain of the same phase. At such a time, the local token must consult the domain's root, since the main tokens of the two domains may be used to create a new domain in a higher phase.

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

A node that joins the algorithm sends test messages over all its edges. An edge over which such a message arrives is marked *nonfaulty*. The main token of each domain waits in the tree root until it is awakened by a local token of the same domain. The local token generated by a node $v$ in a domain $T_1$ tries to annex $v$'s neighbors to $T_1$. To annex a neighbor, the local token travels to the neighbor over an edge which is already marked *nonfaulty* (carrying the phase of domain $T_1$ and the identity of its root). If the phase of the neighbor's domain is lower, then the neighbor is annexed and becomes $v$'s son. In this case, the local token returns to $v$ and tries to annex another neighbor of $v$. If the phase of the neighbor's domain is higher, then $v$'s local token stops. (However, neither $v$ nor the other nodes in $T_1$ are aware of this fact.) Otherwise, the phases are equal. If the neighbor belongs to another domain $T_2$, then a meeting between the main tokens of the two domains is arranged. Only a local token of the domain whose root has the higher identity is responsible for arranging the meeting. Assume, without loss of generality, that the identity of $T_1$'s root is higher than that of the root of $T_2$. The local token of $v$ goes to its domain root and wakes the main token (if it is still there). The main token then travels to the root of the other domain. If it arrives and finds the other main token, then the two main tokens stop (are destroyed), and a main token of a higher phase is created. (If it does not arrive, or does not find the other main token, for reasons to be explained, then it stops). The domain $T_3$ of the new main token contains at first only one node—the root of domain $T_2$. This node has no sons (in the new domain). A local token is also generated at that node, and the new domain

$T_3$ starts to grow as described before. It may take some time until all the nodes of the two domains of the lower phase are annexed to domains of higher phases (not necessarily this new domain). During that time, nodes may still consider themselves as belonging to domains $T_1$ and $T_2$. Also, local tokens of $T_1$ and $T_2$ may continue at that time to annex nodes to their domains. Eventually, only one domain remains in every connected component.

Some care must be taken in implementing the algorithm, to limit its message complexity. When a local token arrives at a node which was annexed by another local token of the same domain, it 'deletes' that edge from the graph. (It will be shown later that such an edge is not needed to maintain connectivity.) Also, a local token stops when it finds that another token of the same phase has preceded it on the way to the root. It also stops if it finds that there already exists a token in a higher phase. Similarly, a main token stops when it finds out that another main token of the same phase has preceded it on its way to meet another main token. It also stops if it finds that there exists a token in a higher phase.

Let us now describe the algorithm in more detail. Initially no edge is marked. The algorithm starts when a nonempty set of nodes create (not necessarily synchronously) a main token and a local token in phase 0, each of which has the identity of its creating node. Such nodes are considered to belong each to its own main token. Nodes that have not been awakened spontaneously belong to tokens in phase $-1$. Every node that initiates the algorithm, or joins it, sends test messages over all its edges. An edge over which such a message arrives is marked *nonfaulty*.

### *The Algorithm of a Local Token in Node $v$*

**(L.0)** If there is no edge $(v, u)$ which (a) leads from node $v$, (b) is already marked *nonfaulty*, and (c) is not marked *deleted, tree-edge-to-father*, or *tree-edge-to-son*, then the local token waits until there is such an edge. (Note that it may wait forever.) If there is such an edge, then the local token traverses edge $(v, u)$ to its other endpoint, $u$.

The actions to follow depend on $v$'s phase $p_v$ and its domain's root $r_v$ (information which the local token carries), compared to $u$'s phase $p_u$ and domain root $r_u$, and the marking of edge $(u, v)$.

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

**(L.1)** ***$p_v < p_u$:*** Node $v$'s local token stops. Note that this is not known to $v$, nor to any other node in $v$'s domain. However, this means that $v$ stops trying to annex neighbors. (A new local token will be created in node $v$ when $v$ is annexed by another domain.)

**(L.2)** ***Either $p_v = p_u$ and nodes $u$ and $v$ belong to the same domain, or $p_v \geq p_u$ and edge $(u, v)$ is marked deleted in $u$:*** The local token marks edge $(u, v)$ *deleted* in $u$ (if it is not already marked *deleted*). Then it returns to node $v$, and acts according to $v$'s phase:

**(L.2.1)** The phase of node $v$ has not increased since the time the local token has left it: The token marks edge $(v, u)$ *deleted*.

**(L.2.2)** The phase of node $v$ has increased since the time the local token has left it: The local token stops.

**(L.3)** ***$p_v > p_u$ and edge $(u, v)$ is not marked deleted in node $u$:*** Node $u$ stops being in its current domain, and becomes $v$'s son in the domain to which the local token belongs. The local token removes all the *tree-edge-to-father* and *tree-edge-to-son* marks (if such exist) in node $u$. Then, it marks edge $(u, v)$ as *tree-edge-to-father* in $u$. A new local token for the new domain is then created in node $u$. If the old local token is waiting in $u$, then it stops. (Otherwise, if not already stopped, the old local token of node $u$ stops on entering node $u$, since $u$ now belongs to a higher phase domain.) Similarly, if the old main token is waiting in $u$, then it stops. Node

$v$'s local token then returns to $v$, and acts according to $v$'s phase:

**(L.3.1)** The phase of node $v$ has increased since the local token has left it: The local token stops.

**(L.3.2)** The phase of node $v$ has not increased since the local token has left it: It marks edge $(v, u)$ as a *tree-edge-to-son.*

**(L.4)** ***Edge $(v, u)$ is not marked deleted in $u$ and $p_v = p_u$ but nodes $v$ and $u$ belong to two distinct trees***: The main tokens of the two domains will stop, and the algorithm tries to create a domain of a higher phase. To do this, the algorithm tries to arrange that the main tokens of the two domains meet in one of the roots, so that they can stop at the same time a domain (main token) of higher phase 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 stops.

**(L.4.2)** $r_v > r_u$: Node $v$'s local token goes to the root $r_v$ to fetch the main token representing $v$'s domain. The route to the root $r_v$ 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*$(p_v)$, i.e., 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 has (meanwhile) been annexed by a higher phase domain, then the local token stops.

**(L.4.2.2)** Also, if the local token enters a node passed before by another local token of the same phase (on its way to the main token), then the local token stops. (The local token is no longer needed since the other local token will travel to arrange the meeting of the main token with another main token.)

**(L.4.2.3)** If the local token arrives at the domain root and the main token is still there, then the main token wakes up. (See the algorithm of the main token.) The local token stops (even if it arrived at the domain root and the main token was no longer there).

*A local token that returned safely to node $v$ (see* **(L.2)** *and* **(L.3)**) *performs* **(L.0)** *again.*

The main token acts as follows.

***The Algorithm of a Main Token***

**(M.0)** When the main token is awakened by a local token of some node $v$ which returned from $v$'s neighbor $u$ (see **(L.4)**), it is routed to the root of $u$'s domain. To find this route, the main token first uses the edges recorded as *local.token.path*$(p_v)$ to arrive at node $u$. Then, it traverses the edges marked *tree-edge-to-father* until it is in the root of $u$'s domain.

**(M.1)** The main token stops if on its way it enters a node which belongs to a higher phase domain. Once it has left its own domain, it also stops when it enters a node that has been visited by another main token of the same phase.

**(M.2)** If the main tokens of the two domains meet (in node $r_u$), then a new domain of a higher phase is created, initially including only $r_u$. A new local token for the new domain is then created in node $u$. All the *tree-edge-to-son* marks are removed. Note that the two domains of the lower phase may still continue to grow, annexing other nodes. However, their main tokens no longer exist; thus, they cannot create more domains of higher phase. Eventually, all their nodes will join domains of higher phases.

## 3. Correctness proofs and complexity analysis

Theorem 3.9 at the end of this section summarizes the correctness proof and the complexity analysis. Lemmas 3.1 and 3.3 prove that the deletion of edges does not disconnect a connected component. Lemmas 3.4 and 3.5 ensure that the

algorithm does not deadlock in a connected component. Lemmas 3.6, 3.7, and 3.8 are used in the complexity analysis.

Let us say that edge $(v, u)$ *belongs* to a domain $T$ at some time if the following holds at that time: (a) one of its endpoints is marked *tree-edge-to-father*, and (b) this marking was made by a local token of domain $T$.

**3.1. Lemma.** *An edge which at some point in time belongs to the tree of domain $T$ is never marked **deleted** by a local token of another domain of the same or of a lower phase.*

**Proof.** We prove the lemma for another domain of a lower phase. The proof for another domain of the same phase is similar. By **(L.2)**, in order that edge $(v, u)$ can be marked *deleted* by a local token of a domain $T_1$, of a lower phase $p_1$, the endpoints of this edge were first annexed by local tokens of domain $T_1$. Since edge $(v, u)$ belongs to $T$, both $v$ and $u$ were also annexed by tokens of $T$ (see **(L.3)**). By **(L.1)** and the assumption that $p > p_1$, the endpoints must have been annexed first by domain $T_1$, and only later by domain $T$. Let $l_1$ be the local token of domain $T_1$ 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 domain $T$. Thus, edge $(v, u)$ was already marked *deleted* (at least in node $u$) when node $u$ was annexed by domain $T$. Hence, by **(L.0)**, **(L.2)**, and **(L.3)**, it could not have become a tree edge in domain $T$; a contradiction to the definition of edge $(v, u)$ as a tree edge in domain $T$. □

**3.2. Definition.** $G'$ is the subgraph of $G$ that contains all the nonfaulty nodes, and the nonfaulty edges connecting them.

**3.3. Lemma.** *Every connected component of $G'$ remains connected after the removal of the **deleted** edges.*

**Proof.** Assume the contrary, and remove every *deleted* edge. Let $R$ be the set of those *deleted* edges whose two endpoints are not connected by a path of non*deleted* and nonfaulty edges. For every phase $p$, let $R_p \subseteq R$ include the edges in $R$, the marking of which was done by a local token of phase $p$. (If the two endpoints of an edge were marked *deleted* by two local tokens, let us consider here only the highest phase local token to mark it.) Let $\bar{p}$ be the highest $p$ such that $R_{\bar{p}}$ is not empty and let edge $(v, u)$ be in $R_{\bar{p}}$. Let us now show that there is a path of non*deleted* and nonfaulty edges connecting $v$ and $u$. This will be a contradiction to the definition of $(v, u)$.

Edge $(v, u)$ was marked *deleted* when a local token of a domain $\bar{T}$ of phase $\bar{p}$ found that nodes $v$ and $u$ belonged to domain $\bar{T}$. Recall that while trying to annex (see **(L.0)**), local tokens are not sent over an edge which belongs to their domain. This, together with **(L.2)**, implies that edge $(v, u)$ did not belong to domain $\bar{T}$. Thus, there existed a nonfaulty path (of domain $\bar{T}$'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 domain $T_1$ of phase $p_1$. By Lemma 3.1, $p_1 > \bar{p}$. Thus, by the definition of $\bar{p}$ and Lemma 3.1, there exists a path of non*deleted* and nonfaulty 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. □

**3.4. Lemma.** *Once a main token of phase $p$ is created in some node $i$ in a maximal connected component $G''$ of $G'$, one of the following must eventually occur: either* (a) *every node in $G''$ is annexed to domain $T$ of this main token, or* (b) *a local token of domain $T$ enters a node which belongs to another domain of the same or higher phase.*

**Proof.** If a local token of $T$ enters a node (perhaps its creating node) which at that time belongs to another domain in the same or a higher phase, then the lemma holds. Thus assume that no local token of $T$ every enters a node of another domain in the same or a higher phase. By **(M.2)** when the main token is created in node $i$, the tree edge markings are removed, and a local token starts to annex $i$'s neighbors. This also happens, by **(L.3)**,

in every node which is annexed. These local tokens do not stop, since they never enter a node which belongs to another domain in the same or a higher phase. The lemma now follows from Lemma 3.3. □

**3.5. Lemma.** *If there is more than one main token at a certain phase p, in a maximal connected component of $G'$, then a main token at phase $p+1$ is eventually created.*

**Proof.** Assume the contrary. Let $G''$ be a maximal connected component of $G'$ for which the lemma does not hold. Then there is a phase $p$ such that there are at least two main tokens at phase $p$ in $G''$, but no main token at phase $p+1$ is ever created. It will be shown that this is impossible.

Denote a main token of phase $p$, created in node $i$, by $(p, i)$. Let $(p, i)$ be a main token in phase $p$ with the maximum possible $i$ in $G''$. Since $p$ is the highest phase in $G''$, no local token of $(p, i)$'s domain can enter a node which belongs to a domain of a higher phase. Also, by the assumption that there is another domain of the same phase in $G''$ and by **(L.4)**, not every node in $G''$ is annexed by the domain of main token $(p, i)$. Thus, by Lemma 3.4, a local token, $l$, of $(p, i)$'s domain eventually enters a node $u$, which belongs to another domain of the same phase $p$. By **(L.4.2)** and the maximality of $i$, the local token is then routed to node $i$. Since $p$ is the largest phase, the local token cannot be stopped in some node, $w$, due to a token of a higher phase 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 domain 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 3.1, some local token of this domain must arrive at node $i$.

Without loss of generality assume that local token $l$ was the first to arrive at node $i$. Main token $(p, i)$ is then routed to $r_u$, the root of $u$'s domain. Since $p$ is the largest phase, main token $(p, i)$ cannot be stopped in some node $w$, due to a token of a higher phase which passed in node $w$ (see **(M.1)**). If another main token $(p, j)$ comes to visit $(p, i)$, then $j > i$. Thus, $(p, i)$ does not go to visit $(p, j)$ (see **(L.4)**). Thus, if $(p, i)$ visits a node $w$ that was previously visited by $(p, j)$, then $w$ belongs to $(p, i)$'s domain. Hence, $(p, i)$ does not stop in $w$ because of $(p, j)$ (see **(M.1)**). Once token $(p, i)$ leaves its domain, it, and maybe other main tokens of the same phase, is routed to node $r_u$ over a tree. Thus, and by Lemma 3.1, some main token of phase $p$ must arrive at node $r_u$.

So far it has been established that some domain root of phase $p$ was visited by a main token of another domain. Let $h$ be the smallest identity among all the roots of domains of phase $p$, which were likewise visited by a main token of another domain of phase $p$. By a similar argument as above, if main token $(p, h)$ had gone to visit the root of a domain of another main token $(p', h')$, then this root $h'$ would have been visited, contrary to the definition of $(p, h)$. Thus, and by the maximality of $p$, main token $(p, h)$ was still in node $h$ when another main token of the same phase visited node $h$. This, together with **(M.2)**, implies that the two main tokens have stopped and a main token of a higher phase was created, contrary to the assumption that $p$ was the highest phase in $G''$. □

**3.6. Lemma.** *The number of phases is bounded by* $\log k + 1$ *where k is the number of nodes which initiated the algorithm spontaneously.*

The proof of this lemma is trivial.

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

(1) *Fault-testing*: These are the messages that each node broadcasts when it begins its part in the algorithm, to enable its neighbors to detect the nonfaulty edges.

(2) *Local annexing*: These are the messages of the local tokens, except for the voyage of a local token to its domain root to fetch the main token.

(3) *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 domain of the same phase.

**3.7. Lemma.** *The number of messages of the 'main meeting' category is bounded by 4n for each phase.*

**Proof.** In order to arrive at its domain root, a local token must first return to its creating node. This uses one message for each local token, i.e., at most $n$ for a phase. Except for the above message, the routes of the local tokens form a forest for each phase, and are disjoint. Thus, they use at most $n-1$ additional messages. In addition, each node $w$ forwards at most two main tokens in the same phase: one main token that comes to visit $w$'s domain root, and the main token of $w$'s domain. The lemma now follows. □

**3.8. Lemma.** *The number of messages of the 'local annexing' category is bounded by* $4m + 5n \log k + O(n)$.

**Proof.** Consider a local token $l$ of node $v$ that is sent to node $u$.

(1) If $l$ annexes $u$, then it returns to $v$. This causes node $u$ to increase its phase. Thus, by Lemma 3.6, the total number of such messages is bounded by $2n \log k + O(n)$.

(2) If $l$ finds a higher phase in $u$, then it stops in $u$. Thus, the total number of such messages is bounded by the number of local tokens. The latter, however, is bounded by Lemma 3.6 by $n \log k + O(n)$.

(3) If $l$ marks edge $(v, u)$ *deleted* in $u$ and returns to $v$ before $v$'s phase has increased, then no other token will ever be sent from node $v$ over this edge. Thus, the number of such messages is at most $4m$.

(4) It may happen that $l$ has marked edge $(v, u)$ *deleted* in $u$, but the phase of $v$ increased before $l$ returns to $v$. In this case, another local token $l'$ may have been sent from $v$ over $(v, u)$ before $l$ arrives back at $v$. In this case, the phase of $v$ is higher than that of $l$. Thus $l$ stops, and the number of messages used in this case is bounded by twice the total number of local tokens. The lemma now follows. □

Note that the edge deleted in cases (3) and (4) of the proof of Lemma 3.8 is actually redundant. The deletion mechanism bounds the number of times such edges are used by $O(m + n \log k)$. The first term in this bound follows from case (3) and the second from case (4).

**3.9. Theorem.** *Let* $G = (V, E)$ *be an undirected graph, possibly containing faulty nodes and edges. Let* $G'$ *be the subgraph induced by the nonfaulty nodes of* $G$, *such that all the faulty edges are removed. The above algorithm weakly constructs a spanning tree in every connected component of* $G'$ *in which at least one node has initiated the algorithm, using* $O(m + n \log k)$ *messages, where* $n = |V|$, $m = |E|$, *and* $k$ *is the number of nodes that initiated the algorithm spontaneously.*

**Proof.** The correctness part follows from Lemmas 3.1, 3.3, 3.4, and 3.5. The complexity part follows from Lemmas 3.6, 3.7, and 3.8. □

## 4. Open problems

If global information of certain types is known, then an algorithm can make a commitment (final output). Such global information can be, for example, the number of nodes, distinct names for the edges, etc. Using such information, an algorithm can detect that certain global conditions hold in the network (e.g., the majority of the nodes agree on some node as a root [3]), and thus the algorithm can output (e.g., that this unique node is the leader). It is interesting to find tight upper and lower bounds for detecting these conditions in such cases (in the above example, to efficiently count the nodes in a growing tree, so that its root will eventually know that the tree contains the majority of the nodes). It is especially interesting to find out whether the message complexity of this task too is not higher than in reliable networks.

## Acknowledgment

It is a pleasure to thank Baruch Awerbuch, Reuven Bar-Yehuda and Oded Goldreich for helpful discussions, Yishay Mansour, Ayla Matalon, Shlomo Moran and Shmuel Zaks for reading and commenting, and an anonymous referee whose comments were also very helpful.





## References

[1] Y. Afek a
tion detect
*ples of Dis*
109–124.
[2] B. Awerbu
1985.
[3] R. Bar-Y
commitme
[4] M. Fische
*uted Syste*
273, June
[5] M.J. Fisc
bility proc
*ACM Sy*
Minaki, C
[6] M.J. Fisc
ity of dis
*ACM* **32**
[7] R.G. Gal
uted algo
*Trans. Pr*
[8] P.A. Hu
weight di
**COM-31**

.) be an undirected
ulty nodes and edges.
uced by the nonfaulty
the faulty edges are
n weakly constructs a
ed component of $G'$ in
nitiated the algorithm,
iges, where $n = |V|$,
ber of nodes that ini-
usly.

ollows from Lemmas
nplexity part follows
.8. □

rtain types is known,
a commitment (final
ition can be, for ex-
, distinct names for
information, an al-
in global conditions
the majority of the
a root [3]), and thus
g., that this unique
resting to find tight
etecting these condi-
above example, to
a growing tree, so
know that the tree
odes). It is especially
r the message com-
not higher than in

Baruch Awerbuch,
ded Goldreich for
ansour, Ayla Mata-
el Zaks for reading
mous referee whose
ful.



## References


[1] Y. Afek and M. Saks, An efficient fault tolerant termination detection algorithm, *Proc. 6th ACM Symp. on Principles of Distributed Computing*, Vancouver, Canada (1987) 109–124.

[2] B. Awerbuch and O. Goldreich, Private communication, 1985.

[3] R. Bar-Yehuda and S. Kutten, Fault tolerant majority commitment, *J. Algorithms* (1988) to appear.

[4] M. Fischer, *The Consensus Problem in Unreliable Distributed Systems — A Brief Survey*, Rept. YALE/DCS/RR-273, June 1983.

[5] M.J. Fischer, N.A. Lynch and M. Merritt, Easy impossibility proofs for distributed consensus problems, *Proc. 4th ACM Symp. on Principles of Distributed Computing*, Minaki, Canada (1985) 59–70.

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

[7] R.G. Gallager, P.M. Humblet and P.M. Spira, A distributed algorithm for minimum-weight spanning trees, *ACM Trans. Programm. Lang. & Systems* **5** (1) (1983) 67–77.

[8] P.A. Humblet, A distributed algorithm for minimum weight direction spanning trees, *IEEE Trans. Commun.* **COM-31** (6) (1983) 756–762.

[9] E. Korach, S. Kutten and S. Moran, A modular technique for the design of efficient distributed leader finding algorithms, *Proc. 4th ACM Symp. on Principles of Distributed Computing*, Minaki, Canada (1985) 163–174.

[10] E. Korach, S. Moran and S. Zaks, *Tight Lower and Upper Bounds For Some Distributed Algorithms for a Complete Network of Processors*, Tech. Rept. #298, Dept. of Computer Science, Technion, Haifa, Israel, November 1983; also in: *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, Vancouver, B.C., Canada (1984) 199–207.

[11] S. Kutten and Y. Wolfstahl, *Tight Upper and Lower Bounds on the Message Complexity of t-Resilient Election in Complete Networks*, Internal memorandum, Technion, Israel, November 1985.

[12] L. Lamport, R. Shostak and M. Pease, The Byzantine general problem, *ACM Trans. Programm. Lang. & Systems* **4** (3) (1982) 382–401.

[13] S. Moran and Y. Wolfstahl, *An Extended Impossibility Result for Asynchronous Complete Networks*, Rept. RC 11896 (#53579), IBM T.J. Watson Research Center, 1986; also appeared as: Extended impossibility results for asynchronous complete networks, *Inform. Process. Lett.* **26** (3) (1987/88) 145–151.