# MAKING DISTRIBUTED SPANNING TREE ALGORITHMS FAULT-RESILIENT

(Extended Abstract)

Reuven Bar-Yehuda, Shay Kutten,
Yaron Wolfstahl and Shmuel Zaks[1]
Department of Computer Science, Technion, Haifa, Israel.

## ABSTRACT

We study distributed algorithms for networks with undetectable fail-stop failures, assuming that all of them had occurred before the execution started. (It was proved that distributed agreement cannot be reached when a node may fail during execution.) Failures of this type are encountered, for example, during a recovery from a crash in the network. We study the problems of leader election and spanning tree construction, that have been characterized as fundamental for this environment. We point out that in presence of faults just duplicating messages in an existing algorithm does not suffice to make it resilient; actually, this redundancy gives rise to synchronization problems and also might increase the message complexity. In this paper we investigate the problem of making existing spanning tree algorithms fault-resilient, and still overcome these difficulties. Several lower bounds and optimal fault-resilient algorithms are presented for the first time. However, we believe that the main contribution of the paper is twofold: First, in designing the algorithms we use tools that thus argued to be rather general (for example, we extend the notion of token algorithms to multiple-token algorithms). In fact we are able to use them on several different algorithms, for several different families of networks. Second, following the amortized computational complexity, we introduce amortized message complexity as a tool for analyzing the message complexity.

## 1. INTRODUCTION

The design of algorithms for distributed networks plays an important role in the study of distributed computing; particularly, a lot of research has been recently performed in models that assume certain faults in the networks. This study suggests a systematic design of fault-tolerant distributed algorithms for the basic problem of spanning tree construction, with the hope that extensions for general problems will follow.

### 1.1 Preliminaries

The basic model consists of a distributed network of $n$ identical processors, except their distinct identities, $k$ of which start the algorithm, and $m$ bidirectional communication links connecting pairs 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 link, (2) receiving a message from a pool of unserviced messages which arrived over links, and (3) processing information locally. Any subset of the processors may start the algorithm. No information is known to any processor, except for the information we explicitly

[1] The work of this author was supported in part by a Technion grant no. 121-641.

assume it has (e.g. that a node knows its own identity). We view the communication network as an undirected graph, where nodes represent processors and edges represent communication links. 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 $O(\log MaxId)$ bits (see e.g. [GHS83]).

In the *leader election* problem (e.g. [GHS83]) it is required that the nodes co-operate to distinguish one of them. Electing a leader, and the related spanning tree construction problem, are fundamental in many distributed algorithms, and have been studied for various models and cost measurements in reliable networks. *Termination detection* applies to a given distributed algorithm; it is required that some node will know that this algorithm has already computed its output, even though this output is distributed among the nodes. This problem is a generalization of the termination detection problem, which was also thoroughly investigated in reliable networks (e.g. [F80]).

Real systems, however, are not reliable. They are subject to faults of different types.

Consider the possibility that some edges or nodes in the network may be faulty. A faulty edge is an edge, every message which is transmitted over which, in either direction, is lost. A faulty node is a node all of its incident edges are faulty. Because our basic model is asynchronous, it cannot be distinguished whether a message is delayed or lost.

For the case where nodes can fail *during* the execution of an algorithm it was proved that no protocol for reaching distributed agreement (hence election) can be devised [FLP85]. Other types of faults are also hard or impossible to cope with [F83]. However, the presence of faults was pointed out as a motivation for leader election (as a method to reorganize the network) [G82]. (It was also pointed out there, that it is suffucient to deal with limited types of faults, since reliable hardware equipment makes failures of the more general type quite rare [G82].) We assume that no failure occurs during the execution of the algorithm.

Even with these assumptions about faults, it can be shown that no fault-tolerant leader-election algorithm can guarantee termination. Thus additional assumptions are needed. This includes, for example, knowledge about synchrony in the network ([G82]), its topology ([KW84,SG85]), or its size ([SG85]). Alternatively we can omit the requirement for termination detection and do with a spanning tree construction (rather than leader election).

We develop in this paper some new fault-resilient algorithms. However, our main contributions are the tools we use: First, we believe that our techniques to make existing algorithms fault resilient are rather general. In fact we are able to use them on several different algorithms, for several different families of networks. The applicability of our method to some other algorithms is straightforward (e.g. applying it to the algorithm of [HS80] for cycles will yield an algorithm which is resilient to many faults.) Second, we introduce a method we have found useful in analyzing the complexity of such algorithms.

### 1.2 Our Assumptions and Results

In spanning tree construction algorithms, trees are merged to eventually span the whole network. Every tree has a unique *center of activity*, which is a node, or a token, that decides on the tree expansion or merging. This method seems essential to prevent contradicting decisions (like two trees merging via two edges and creating a cycle). This center always 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 either 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 "One-expanssion-direction" 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 center of activity.

In our algorithms we use more than one token in a tree. Most decisions are done locally (without consulting the center), 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. Thus the center must be consulted only when this information does not suffice to make the decision. This information is a lower bound on a certain value that represents the status of the tree (and remains true even when the exact value is updated); this value may be the size of the tree (Section 2), its phase (Section 3.1) or some criterion to know which of the new non-faulty edges were already reported to the center (Section 3.2).

Using this method we also show that the complexity of a resilient algorithm is not always higher (in order of magnitude) than the complexity of a non-resilient algorithm for the same problem, even though duplicating messages is used.

A $t$-resilient algorithm is an algorithm that yields the correct answer when at most $t$ processors are faulty. We first derive an $\Omega(n \log k + k\, t)$ lower bound for the message complexity of any $t$-resilient election algorithm in complete networks, and show how to modify an existing election algorithm ([H84]) to achieve a $t$-resilient election algorithm (for any $t < \frac{n}{2}$) the complexity of which matches the lower bound. It also improves on existing resilient algorithms in terms of message, bit, space and computational complexity measures. Note that when $t$ is $O(log\ k)$ the message complexity is the same as for reliable networks. On the other hand, for a larger $t$ the message complexity must be higher than that for reliable networks (where it is $O(n \log k)$ [KMZ84]). We also consider the cases where edges may fail and where every node's identity is known to its neighbors. In order to analyse the message complexity of the algorithm we introduce the notion of *amortized* message complexity, derived from that of *amortized* computational complexity in [FT84]. It is suggested that this technique may prove useful in analyzing the complexity of distributed algorithms in similar situations. This is done in section 2.

Next we present (in section 3.1) an optimal resilient spanning tree construction algorithm for general graphs. For this we modify the non-resilient algorithm of [KKM85].

As explained above, the termination cannot be detected in this model without additional knowledge. We therefore add (in section 3.2) the assumption that $n$ is known, with which termination can be detected by the above algorithm, but with a higher message complexity. We then modify another non-resilient algorithm ([G]) and get a resilient version of it, improving the message complexity of the termination detection. In a complete network the message complexity of this algorithm is also optimal. As exemplified by this algorithm, termination detection in this model is a different (generalized) problem than the thoroughly investigated termination detection problem in reliable networks, and needs different solutions.

In [G82, M84] election in unreliable networks is investigated for the synchronous case, while we investigate the asynchronous case. Fault-resilient consensus in the presence of partial synchrony is discussed in

[DLS84]. An $\lceil n/2 \rceil - 1$-resilient consensus algorithm for a complete network is presented in [FLP85]. $O(n^2)$ messages are sent in any execution of this algorithm; however, since most messages contain $O(n \log MAX\_ID)$ bits, the bit complexity is $O(n^3 \log MAX\_ID)$ and the message complexity, in terms of our model, is $O(n^3)$ (our result implies an $O(n^2)$ algorithm for this case).

A (rather complicated) 1-resilient election algorithm for complete networks is presented in [KW84]. A method to generalize it for a larger $t$ is hinted, but for $t = \lceil \frac{n}{2} \rceil - 1$ the message complexity of a generalized algorithm would be $O((t \log t)(n \log n)) = O(n^2 \log^2 n)$. An $\Omega(n \log n)$ and $O(n \log n)$ lower and upper bounds for 1-resilient election in a ring are found in [SG85].

## 2. RESILIENT ALGORITHMS IN COMPLETE NETWORKS

### 2.1 Informal description of the t-resilient algorithm

Here we assume that the communication graph is complete, and only nodes may fail. In [KWZ86] we show that in the case where also edges may fail similar results hold. We modify the simple algorithm of [H84] (which is similar to that of [AG85a]) that elects a leader in a non-faulty complete network. In this algorithm some nodes are candidates for leadership, called *kings*. Each king tries to *annex* other nodes to its *domain* (initially containing only itself). An annexed king ceases to be a king, and stops trying to annex other nodes, but those already annexed by it remain in its own domain. The *size* of a node is the size of its domain, which is initially zero, becomes 1 when the node wakes up spontaneously, and may only grow. The size and identity of node $A$ are denoted by *size(A)* and *id(A)*, respectively. A node may belong to several domains, but it remembers the edge leading to its *master*, that is the last node by which it was annexed (a node that has not been annexed by another node is considered its own master). Each king owns one *token*, which is a process representing it, and carrying its size and identity, as well as an additional message, which is one of the following:

a. an ASK message, originated by the node that owns the token.

b. an ACCEPT message, originated by a node that was annexed by the token.

In order to annex a neighbor $B$, king $A$ sends its token to visit $B$ (with an ASK message). The token proceeds from node $B$ to $B$'s master $C$ (may be $B$ itself). The next actions taken by the token depend on *(size(A), id(A))* and the information it finds in $C$ and $B$, as follows:

Case 1. *((size(A), id(A)) > (size(C), id(C))* [2]:

Node $C$ ceases to be a king [3], but does not join $A$'s domain. The token returns to node $B$. If by now any token of another node $D$ has passed $B$ and *(size(D), id(D)) > (size(A), id(A))*, then the token (of $A$) is killed. Otherwise, $B$ joins $A$'s domain, and the token returns to $A$ (with an ACCEPT message), and *size(A)* is incremented (by 1).

Case 2. *((size(A), id(A)) < (size(C), id(C))*:

The token is killed.

A token that returns safely repeats the process of attempting to annex a new neighbor. The algorithm terminates when one node $A$ has $size(A) = n$.

---

[2] Lexicographically, namely $size(A) > size(C)$ or $size(A) = size(C)$ and $id(A) > id(C)$.

[3] Note that that a node may "cease to be a king" more than once; we use this convention throughout the paper.

In our $t$-resilient version each king owns $t+1$ tokens that are initially sent to different neighbors. In addition to the types of messages used above, there is a third type:

c. a REJECT message, originated by a node that refused to be annexed by the token. The actions taken by a token of node $A$ that arrives at a master $C$ of a neighbor $B$ of $A$, depend on *(size(A), id(A))* and the information it finds in $C$ and $B$, as follows:

Case 1. *((size(A), id(A)) > (size(C), id(C))*:

Node $C$ ceases to be a king, (but does not join $A$'s domain). The token returns to node $B$, and proceeds according to the following:

Case 1.1. A token of node $D$ has meanwhile passed $B$ and *(size(D), id(D)) > (size(A), id(A))*:

Node $A$'s token is not killed, but returns to $A$ (with a REJECT message), carrying *size(D)* and *id(D)*. *size(A)* may have increased while waiting for this token to return. If *(size(A), id(A))* is still smaller than *(size(D), id(D))* (note that for *size(D)* we use the lower bound carried in the token), then node $A$ ceases to be a king (without joining any other domain). We term this REJECT message a *relevant* REJECT. Otherwise *(size(A), id(A)) > (size(D), id(D))* and we say that this REJECT message is *irrelevant*. Node $A$ now starts a *war* with $C$ (unless it is currently involved in another war). Any of $A$'s tokens returning during a war are not sent again until the war is over (we term these tokens *suspended*). In the war $A$ sends again the token to $B$ (with an ASK message). The war with $C$ continues until the token returns from $B$ to $A$ with an ACCEPT message, or until $A$ loses (as a result of a relevant REJECT message or a leader announcement message). During this war, king $A$'s token may return several times from $B$ with irrelevant REJECT messages, in which case it will be sent back to $B$ (with the updated size of $A$). If $A$ wins it increments *size(A)* and all the currently suspended tokens are sent over unused edges, unless one of them is an irrelevant REJECT message, in which case another war starts.

Case 1.2. No token of another node $D$, such that *(size(D), id(D)) > (size(A), id(A))*, has passed $B$:

$B$ joins $A$'s domain, the token returns to $A$ (with an ACCEPT message), and *size(A)* is incremented.

Case 2. *((size(A), id(A)) < (size(C), id(C))*

The token returns to node $B$ and then to $A$ (with a REJECT message), carrying *size(C)* and *id(C)*, and proceeds according to the following:

Case 2.1. *(size(A), id(A)) < (size(C), id(C))* (using the lower bound for *size(C)* carried in the token):

Node $A$ ceases to be a king (without joining any other domain).

Case 2.2. *(size(A), id(A)) > (size(C), id(C))* (as carried in the REJECT message; this REJECT message is also called *irrelevant*):

Node $A$ starts a war with $B$ as in Case 1.1.

A token that returned safely repeats the process of attempting to annex a new neighbor. When a node $A$ has *size(A)* $> \frac{n}{2}$, it announces its leadership, and the algorithm terminates.

### 2.2 Correctness and complexity analysis

In [KWZ86] we prove the correctness:

**Theorem 1**: In every execution of the $t$-resilient algorithm in a network with no more than $t$ faulty processors, exactly one node declares itself a leader.

For the complexity analysis we define the *amortized message complexity* as the actual message complexity plus the *potential*s of the nodes, which are non-negative integers (defined in the sequel). Thus the amortized message complexity is an upper bound for the message complexity. In some events during execution the amortized message complexity does not change, although the actual one increases. This is because other components of the amortized message complexity decrease. Thus the amortized message complexity is a useful tool when we want to have an upper bound on the number of messages sent, even though the number in each case is not known. (This makes the method especially attractive when dealing with distributed algorithms, and especially in the presence of faults. In this environment the "uncertainty" is especially high.)

We use the following notations denoting values up to (and including) the $\tau$-th reception event in a node $A$ (including its treatment and the messages sent as a result), for a particular execution of the algorithm.

(1) $\#send(A,\tau)$ denotes the number of times a token has been sent by KING $A$. This is $A$'s part in the actual message complexity of a prefix of that execution of the algorithm.

(2) $\#wars(A,\tau)$ is the number of wars in which $A$ was involved.

(3) $\#susp(A,\tau)$ is the total number of times a token was suspended in $A$.

(4) $\#held(A,\tau)$ is the total number of currently suspended tokens.

(5) $\#inc(A,\tau)$ is the increment in the size of $A$ since the beginning of the current war, if such exists.

(6) $\#IRR(A,\tau)$ is the number of irrelevant REJECT messages received by $A$ from $B$ (the node with which it is currently involved in a war, if such exists) since the beginning of the current war.

(7) $SIZE(A,\tau)$ is the size of $A$'s domain.

(8) The $potential(A,\tau)$ of $A$ is $\#held+(SIZE-\#wars)+(SIZE-\#susp)+(\#inc-\#IRR)$

(we omit the $(A,\tau)$ notation in all terms).

**Lemma 1:** Let $A$ be any node that initiated the algorithm and is still a KING after its $\tau$-th reception in a particular execution. Then

$$3SIZE+t+1>\#send+potential \qquad (*)$$

**Sketch of Proof:** First we note that the following is clear from the definition:

$$\#inc-\#IRR\geq 0 \qquad (**)$$

We continue by induction on $\tau$.

**Induction Basis:** On initiating the algorithm, node $A$ sends $t+1$ tokens. By that time:

$3SIZE+t+1=4+t>(t+1)+0+1+1+0-0=\#send+\#held+(SIZE-\#wars)+(SIZE-\#susp)+(\#inc-\#IRR)$

and (*) holds.

**Inductive Step:** Assuming (*) holds after the $\tau$-th reception, we show that it holds after the $\tau+1$st reception. From the formal description of the algorithm we have to consider seven cases, only three of which are discussed here:

**Case 1:** A token that successfully accomplished a capture (i.e. a token with an ACCEPT) is received and $A$ is not involved in a war. The token is sent for another capture attempt, both $SIZE$ and $\#send$ are incremented (by 1), so (*) holds.

**Case 2:** A token with an ACCEPT message is received while $A$ is involved in a war. The token is suspended,

*SIZE* ,*#susp*, *#held* and *#inc* are incremented and (*) holds.

**Case 3:** A token has returned with an ACCEPT from some processor with which $A$ is in a war. If no other REJECTs are suspended then *SIZE* is incremented and all suspended tokens are now sent on unused edges. For each token sent, *#send* is incremented and *#held* is decremented. Also, $\#inc - \#IRR = 0$, so by (**) the right-hand side does not increase.

**Corollary 1:** The total number of ASK messages sent by a processor is bounded by $3 \cdot s + t$, where $s$ is the final size of the processor.

Now that we have bounded the number of tokens a KING sends, by the KING's final size, we can use a technique similar to the one used in e.g. [AG85a, G, H84], using the following lemma:

**Lemma 2:** If there are $l-1$ KINGs whose final size is larger than that of KING $A$, then the latter is bounded by $n/l$.

The message complexity can now be computed:

**Theorem 2:** The number of messages used by the algorithm is $O(n \log k + k\, t)$.

**Sketch of Proof:** Let $s$ be a final size of a KING that has initiated the algorithm. By Corollary, 1 the number of times this KING has sent a token with an ASK message does not exceed $3 \cdot s + t$. Each time the token used at most 4 messages (2 to arrive at the neighbor's master and 2 to come back). The final size of the nodes that were never awakened by the high-level protocol is one. Other $n-1$ messages are needed for the LEADER announcement. Thus by Lemma 2 the total number of messages is bounded from above by $pn - 1 + 4(3n\,(1+1/2+1/3+...+1/k) + k\, t) = O(n \log k + k\, t)$.

The following theorem implies that our algorithm is optimal:

**Theorem 3:** The message complexity of election in complete networks containing at most $t$ faulty processors is at least $\Omega(n \log k + k\, t)$.

The proof of Theorem 3 appears in [KWZ86]. For the case where every node knows its neighbors' identities, a variation of the above proofs can be used to prove (see [KWZ86]):

**Theorem 4:** The message complexity of election in complete networks containing at most $t$ faulty processors where every node's identity is known to the node's neighbors is $\Theta(k\, t)$.

## 3. RESILIENT ALGORITHMS IN GENERAL NETWORKS

### 3.1 Constructing a spanning tree in general networks

We make the algorithm of [KKM85] resilient and optimal. As in the case of complete networks, the idea is to make decisions (as long as possible) using information which may not be up-to-date. Here this policy leads to enabling many nodes of a tree to locally decide in parallel, to add neighbors to the tree. Thus a tree may annex new leaves, while other nodes of that tree (even the root) are being captured by other trees. We give here a brief description of the algorithm. (This problem was independently investigated in [AG85b]. Another algorithm for this problem was lately announced [AS86].)

First we explain the basic principle of the algorithm. Initially every node 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 following marks in each of its endpoints (or in both): *non-faulty*, *tree-edge-to-father*, *tree-edge-to-son*, and *deleted*. Initially no edge is marked. Every node is a part of a rooted tree, initially containing only itself. Each tree has an associated value called *generation* (or *phase*), initially 0, and is set to 1 if its root node spontaneously starts the algorithm.

Each tree has also a single *main* token which usually resides in the root. Main tokens of trees are used to create new trees in such a way that the creation of a new tree (main token) in generation $g+1$, involves the destruction of at least two main tokens in generation $g$. We defer the explanation of the exact method by which such a new main token is created. For now it suffices to note that since each tree has only one main token, the number of trees in generation $g+1$ is at most half the number of trees in generation $g$.

Each node in a tree generates a *local* token, whose task is to annex the node's neighbors to the node's current tree. Several local tokens of a tree $T$ may simultaneously annex nodes to the tree (in a method to be explained). 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 (as long as the node doesn't join another tree). The tree information that the local token may not have is the answer to the question "does the main token still exist?" For example it may happen that the root has been annexed by a tree in a higher generation, and the main token has thus been destroyed in the root. Thus the local token must consult the tree root only when the local token arrives at a node that belongs to another tree of the same generation. In this case the local token must consult the tree root, since the main tokens of the two trees may be used to create a tree in a higher generation.

We now outline the algorithm. A more detailed description appears in [K86].

The main token of each tree waits in the tree root until it is awakened by a local token of the same tree. The local token generated by 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. (Otherwise the arrangement is the "responsibility" of the other tree.) To arrange the meeting it goes to its tree root and wakes the main token. The main token then travels to the root of the other tree. 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. The tree of the new main token first contains only one node- the root of the second mentioned tree. A local token is also generated at that node, and the new tree starts to grow as described above. Some time must pass until all the nodes of the two trees of the lower generation are captured by trees of higher generations (not necessarily this new tree). Each of 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. When a local token arrives at a node which was annexed by another local token of the same tree, it "deletes" that edge from the graph. (It is shown in [K86] that such an edge is not needed to maintain the connectivity.) Also, a local token is destroyed when it finds that another token of the same generation (or a higher generation) has preceded it on the way to the root. Also, a main token is destroyed when it finds that another main token of the same (or higher) generation has preceded it on its way to meet another main token.

In [K86] we prove the correctness of the algorithm and its optimality:

**Theorem 5:** Let $G$ be an undirected graph, containing faulty nodes and edges. Let $G'$ be the graph of the non-faulty nodes and edges of $G$. The above algorithm weakly constructs a spanning tree in any connected component of $G'$, using $O(m + n \log k)$ messages.

**3.2 Leader election with termination detection in general networks**

In order to elect a leader and detect termination, we now assume that every node knows $n$. (Thus, like in the case of complete graphs, termination can be detected when more than $\frac{n}{2}$ nodes are captured by some node.) We also assume that there are non-faulty edges which, together with the non-faulty nodes, form a connected non-faulty component which contains more than $\frac{n}{2}$.

A relatively simple (but having a high message complexity) method is to use the algorithm of Section 3.1 and add some a subroutine that counts the nodes. Each node which joins a tree can send a message to the tree root and when there are more than $\frac{n}{2}$ nodes the root becomes a leader. One problem with this scheme is that nodes may also leave trees. However, even if this problem is solved, the message complexity could be $O(n)$ for each node to join a tree. Since there are $n$ nodes, and each may join $O(\log n)$ trees, the message complexity is $O(n^2 \log n)$. By modifying the algorithm in [G] we derive an $O(n^{2})$ fault-resilient algorithm. This matches the lower bound of $\Omega(m + n \log k)$ messages in the case of dense graphs, and is therefore not worse than that of existing non-resilient algorithms. The description of the algorithm follows. (The formal proofs appear in [BK86].)

We start with a main token which traverses its tree in order to find an edge leading to a node which belongs to another tree. This must be an edge which is known not to be faulty, otherwise the token may be lost, and the algorithm will be deadlocked. Once the token traverses such an edge, the conventional methods can be used in order to decide what happens to the two trees; for example, they can be merged, as in Section 3.1. The problem is to find such an edge using a small number of messages.

Each node initially contains a "sleeping" *king* token identified by the node's *Id*, and is considered as a one-node rooted "territory" tree. The *king* and its tree have an associated *IdOfKingdom* which is the *king*'s *Id*. The *king*'s role is to expand its "territory" tree in order to occupy a majority of the nodes. Each edge end-point contains a *scout* token. The role of a *scout* token is to test its edge and notify a *king*. When a *king* is notified by a *scout*, it starts a traversal "battle." In each "battle", the king traverses only tested edges and eventually returns to its root. While the *king* traverses these edges, it is rebuilding its territory tree. When the *king* enters an "enemy territory" (a node with a different *IdOfKingdom*), the *Id*s of the enemy tree and the *king* are compared in order to find out who wins. If the *king*'s *Id* is lower, it becomes a "loser," which eventually causes the *king*'s "death." Otherwise, the *king* assigns its *Id* to the node's *IdOfKingdom*, thereby annexing the node to its "territory" tree.

In refining the above scheme we aimed to guarantee:

(1) Once a king has a majority, no other king will ever have a majority. (This is needed to prove the partial correctness.)

(2) In every "battle" traversal the king visits an enemy territory. (This is needed to prove termination.)

(3)

(3a) In its "battle" traversal a king blocks edges that are not needed for connectivity. Once an edge is blocked by a king, no other kings will ever traverse it.

(3b) For each king's "battle" traversal, every node forwards at most one scout on its way to notify the king. (These are needed in order to reduce the messages complexity.)

Each "battle" consists of two traversals. In the first the king *freezes* every node it annexes. A king which enters an enemy frozen node, waits until either the node is unfrozen, or the king has become a "loser". When the king counts a *Majority* nodes it has frozen in this "battle", it stops after broadcasting a leadership message. (This guarantees (1).) In the second traversal of its "battle", the king unfreezes all the nodes it has frozen, and returns to sleep in its initiating node.

The traversals are performed according to the Depth First Search (DFS) strategy, specifically, that of Tremaux (see e.g. [E79]). The king traverses only edges which are tested and not blocked. An edge is blocked by the king when it is a DFS *back edges*. (This, together with the freezing mechanism, guarantees (3a).)

The scout makes a round trip on the edge it tests. according to the information it finds in both endpoints it decides whether the king should be notified. If so the scout climbs up the tree built by the king. It stops when its notification is guaranteed to reach the king (either by this scout, or by another). (This, together with the two-traversals mechanism, will be shown to guarantee (2) and (3b).)

## The Formal Algorithm

Each node contains:

| | |
|---|---|
| Constants: | *Id* = the identity of the node; {The only difference between nodes' programs}<br>*Majority* = any integer greater than $n/2$; {$n$ - total number of nodes} |
| Variables: | *IdOfKingdom* initially *Id* ;<br>*Counter* : initially 1 ;<br>*Date* : initially 0 ; |
| Token: | *king* : initially in status *Sleep* , and associate 4-touple:<br>(*KingId* , *KingCounter* , *KingShouldWake* , *KingHasLost*)<br>initially ( *Id* , 1, *false* , *false* ) ; |

Each edge entry in a node has:

| | |
|---|---|
| Variable: | *mark*: (*CLOSE* , *OPEN* , *USED* , *UNUSED UP* , *BLOCK*) initially *CLOSE* ; |
| Token: | *scout*: initially in status *test*, and associate variables:<br>*ClimberDate* and *CompareId*, initially undefined; |

When a node wakes up {spontaneously or by a token's arrival} each *scout* token is sent through its entry.

## The Algorithm of a Scout *x*-->*y*

*test* Status:

Travel from $x$ through edge $(x,y)$ ;
When arrives to $y$: Mark edge $(x,y)$ *OPEN* if marked *CLOSE* ;
Go to *compare* Status ;

*compare* Status:

*CompareId* := *IdOfKingdom* {of $y$} ;
Carry with you *CompareId*, travel back to $x$ ;
{Compare} When arrives $x$: If *IdOfKingdom* {of $x$} = *CompareId* {of $y$} then stop {vanish} ;
{Wait} Wait Until $x$ is not frozen ;
{Check} If edge $(x,y)$ in $x$ is not marked *OPEN* then stop {vanish} ;
{Set} *ClimberDate* := *Counter* {of $x$} ;
Go to *Climber* Status ;

```
climber Status:
                        While ClimberDate = Counter and ClimberDate > Date {in x} do begin
{UpDate}                    Date := ClimberDate ;
{Arrived? Stop!}            If x is frozen or no edge in x is marked UP then stop {vanish} ;
{Climb up}                  Travel the edge (x,y) marked UP, and assign x:=y ;
                        end {While}
                        Stop {vanish} ;
```

## The Algorithm of a King initiated in node r

```
Sleep Status: {Initially and after Unfreeze}:
                        If KingHasLost then stop {vanish} ;
{Should wake}           If KingShouldWake then Go to Freeze Status ;
{Sleep}                 Wait Until r is frozen or Date = Counter ;
{Wake}                  If r is frozen then stop {vanish} else Go to Freeze Status ;
Freeze Status: {From status Sleep}
{Initiate}              Freeze r; KingCounter:=1; x := r;
{Main}                  Repeat
{Go up}                     If x ≠ r then travel the edge (x,y) marked UP, and assign x:=y ;
{Deep down}                 While exist an OPEN or UNUSED edge in x do begin
{Freeze edge}                       Let (x,y) be such an edge. Mark (x,y) USED and travel to y ;
{Old node?}                         If y is frozen and belongs to your kingdom then begin
{Block}                                 Mark (x,y) in y BLOCK ;
{Go back}                               Travel back to x ; Mark (x,y) in x BLOCK ; end {if}
{New node}                          else begin
{Wait}                                  Wait Until y is not frozen or a higher identity is known to y ;
{Lost?}                                 If a higher identity is known to y
{Lost!}                                 then begin
{Back off}                                  Travel back the edge (x,y) ; Mark (x,y) in x UNUSED
                                            ;
{Give up}                                   KingHasLost := true ; Go to Unfreeze Status ; end
                                            {Lost!}
{Take over}                             else begin
{Freeze node}                               Freeze the node and increment KingCounter ;
{Majority?}                                 If KingCounter ≥ Majority then Go to Leader Status ;
{Update father}                             If an edge in y is marked UP then mark it UNUSED ;
                                            If y ≠ r then mark (x,y) UP ;
                                            Assign x:=y ;
                                        end {Take over}
                                    end {New Node}
                            end {while}
{Till finish}               Until ( x = r )
                            Go to Unfreeze Status ;
Unfreeze Status: {From status Freeze}
{Main}                  repeat
{Melted by you?}            If x is not frozen then
{Unfreeze edge}               Travel the edge (x,y) marked UP and Mark (x,y) in y UNUSED ;
{Deep down}                 While exist a USED edge in x do
```

Let $(x,y)$ be a *USED* edge: Travel to $y$, and assign $x:=y$ ;

{Should wake?} If *Date* = *Counter* then *KingShouldWake* :=*true*

{Update node} *Counter* :=*KingCounter* ; Unfreeze $x$ ;

{Till finish} Until ( $x=r$ )

Go to *Sleep* Status ;

*Leader* Status: {From status *Freeze* when *KingCounter* =*Majority* }

{Election} The current node is the leader.

{Notify}

Perform a broadcast over non-*BLOCK*ED EDGES, to notify all nodes.

{termination} When a node forwards this broadcasts it records the edge over which it has received the broadcast, and stops all algorithm activity.

In [BK86] we prove:

**Theorem 6:** Let $G$ be an undirected graph, containing faulty nodes and edges. Let $G'=(V',E')$ be a connected subgraph of $G$ containing only non-faulty nodes and edges, s.t. $|V'| > \frac{n}{2}$. The above algorithm elects a leader in $G$ using $O(n^2)$ messages.

## REFERENCES


[AG85a] Afek, Y., and Gafni, E., Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks, *4-th ACM Symposium on Principles of Distributed Computing*, Minaki, Canada, August 1985, pp 186-195.

[BK86] Bar-Yehuda, R., and Kutten, S., Fault-Tolerant Leader Election with Termination Detection, in General Undirected Networks, Technical Report #409, Computer Science Department, Technion, Haifa, Israel, April 1986, Revised August 1986.

[AG85b] Awerbuch, B., and Goldreich, O., private communication.

[AS86] Afek, Y., and Saks, M., An Efficient Fault Tolerant Termination Detection Algorithm (draft), unpublished.

[DLS84] Dwork, C., Lynch, N., and Stockmeyer, L., Consensus in the Presence of Partial Synchrony, *3th ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1984, pp 103-118.

[F83] Fischer, M. The Consensus Problem in Unreliable Distributed Systems (a Brief Survey), YALE/DCS/RR-273, June 1983.

[FLM85] Fischer, M.J., Lynch, N.A., and Merritt, M., Easy Impossibility Proofs for Distributed Consensus Problems, *4-th ACM Symposium on Principles of Distributed Computing*, Minaki, Canada, August 1985, pp. 59-70.

[FLP85] Fischer, M., Lynch, N., Paterson, M., Impossibility of Distributed Consensus with One Faulty Process, *JACM*, Volume 32(2), April 1985.

[F80] Francez, N., Distributed Termination, *ACM-TOPLAS*, January 1980.

[FT84] Fredman, M.L., and Tarjan, R.E., Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, *25th FOCS* Singer Island, Florida, October 1984.

[G] Gallager, R.G., Finding a Leader in a Network with $O(|E|)+O(n \log n)$ messages, Internal Memo, Laboratory for Information and Decision Systems, MIT, undated.

[G82] Garcia-Molina, H., Election in a Distributed Computing System, *IEEE Trans. on Computers*, Vol. c-31, No 1, 1982.

[GHS83] Gallager, R.G., Humblet, P.M., and Spira P.M., A Distributed Algorithm for Minimum-Weight Spanning Trees, *ACM TOPLAS*, January 1983, Vol. 5, No. 1.

[H84] Humblet, P., Selecting a Leader in a Clique in $O(n \log n)$ Messages, internal memo., Lab. for Information and Decision Systems, M.I.T., February, 1984.

[HS80] Hirshberg, D.S., and Sinclair, J.B., Decentralized Extrema-Finding in Circular Configurations of Processes, *CACM*, November 1980.

[K86] Kutten, S., Optimal Fault-Tolerant Distributed Spanning Tree Weak Construction in General Networks, Technical Report #432, Computer Science Department, Technion, Haifa, Israel, August 1986.

[KKM85] Korach, E., Kutten, S., and Moran, S., A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms, *4-th ACM Symposium on Principles of Distributed Computing (PODC)*, Minaki, Canada, August 1985, pp. 163-174.

[KMZ84] Korach, E., Moran, S, and Zaks, S, Tight Lower and Upper Bounds For Some Distributed Algorithms for a Complete Network of Processors, *3th ACM Symposium on Principles of Distributed Computing (PODC)*, Vancouver, B.C., Canada, August 1984, pp. 199-207.

[KW84] Kutten, S., and Wolfstahl, Y., Finding A Leader in a Distributed System where Elements may fail, *Proceeding of the 17th Ann. IEEE Electronic and Aerospace Conference (EASCON)*, Washington D.C., September 1984, pp. 101-105.

[KWZ86] Kutten, S., Wolfstahl, Y., and Zaks, S., Optimal Distributed $t$-Resilient Election in Complete Networks, Technical Report #430, Computer Science Department, Technion, Haifa, Israel, August 1986.

[M84] Merritt, M., Election in the Presence Of Faults, *3th ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1984, pp. 134-142.

[SG85] Shrira, L., and Goldreich, O., Electing a Leader in the Presence of Faults: a Ring as a Special Case, to appear in *Acta Informatica*.