FAULT TOLERANT LEADER ELECTION WITH TERMINATION DETECTION, IN GENERAL UNDIRECTED NETWORKS

(Extended Abstract),

by

R. Bar-Yehuda and S. Kutten

Technical Report #409
April 1986
Fault Tolerant Leader Election with Termination Detection, in General Undirected Networks

(EXTENDED ABSTRACT)

Reuven Bar-Yehuda* and Shay Kutten
Computer Science Dept
Technion - Haifa 32000 Israel
*Visiting Computer Science Dept.
Duke University - Durham NC 27706 USA
Phone: (919)-684-3048, Net: rby@duke

Abstract

The problem of leader election in an asynchronous network with possible faulty edges (and nodes) is studied. In our model, a faulty edge is an edge which does not transfer messages in both directions. The election is needed in such cases in order to reorganize the network after failures have occurred. The goal is a fault tolerant algorithm with detection of termination. The common methods for the heavily studied termination detection cannot be applied in unreliable networks. Actually, when there is no additional knowledge; it is obvious that no fault tolerant algorithm can guarantee termination detection. However in real networks it is reasonable to expect that some global information, such as "how many nodes nodes form a majority", is known. For this model we present an $O(n^2 + m)$ messages complexity algorithm, and each message is $O(\log (MaxId))$ bits (where $n$ is number of nodes, $m$ the number of edges and MaxId is the maximum identity.) Before presenting the formal algorithm and proofs, we present an informal description in which we construct our algorithm step by step. The algorithm guarantees termination in a component with a majority of the nodes. This algorithm can be used in networks in which message transmission is not restricted to the FIFO discipline. Thus, the memory (or the time and messages) needed to simulate the FIFO discipline, is saved. The memory space needed in each node is only $O(MaxNodesDegree + \log (MaxId))$ (which, within a constant, is the best possible).
1. Introduction

1.1 The Basic Model

The model under investigation is a network of \( n \) processors with distinct totally-ordered identities, and \( m \) bidirectional communication lines connecting pairs of processors. Each processor knows the lines connected to itself. The communication is done by sending messages along communication lines. The network is asynchronous: the time to transmit a message from node to node 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 to a neighboring processor; (2) receiving a message from a pool of unserviced messages transmitted by a neighbor; and (3) processing information in local memory.

We view the communications network as an undirected graph, where nodes represent processors, and edges represent communications lines.

1.2 Measuring Efficiency

To measure the efficiency of a distributed algorithm, we use the common measure of the maximal possible number of messages transmitted by the node, where each message contains at most \( O(\log MaxId) \) bits (see e.g. [KKMB5, KMZB4, KRSB5].) We are also interested in the maximal possible number of bits needed in a node.

1.3 The Problem

In the problem of leader election (e.g. [GB2, GHSB3, KKM85, KMZB4, KRSB5]) it is required that the nodes co-operate to elect exactly one which will "know" that it has been elected leader. Electing a leader is fundamental in many distributed algorithms, and has been studied with various models, graphs and cost measurements. Except for [GB2], [KW85], and [SG85], it was always assumed that all edges and nodes are without faults. In [GB2] it is pointed out that the presence of faults is a strong motivation for election, as the elected leader can be used to reorganize the networks. However, in [GB2] it is assumed that faults are detectable, while we give up this restriction. [KW85] have solved the problem optimally for complete networks. They also suggest an
optimal algorithm for general networks if termination detection is not required. A similar solution for general networks without termination detection was suggested independently by [AG85].

1.4 Edges and Nodes with Faults.

We will consider the possibility that edges in the network may be faulty. A faulty edge is an edge that does not transmit messages in either direction. A faulty node is modeled by a node all of which edges are faulty. Because our basic model is asynchronous, it cannot be distinguished whether a message sent through an edge has simply been delayed or lost.

In networks where edges may fail at any time during execution of a protocol, the consensus task (and hence the election task) is unsolvable. [FLP85, MW85, DDS83, DLS84, FB3] Other types of failure are also hard or impossible to tackle [DDS83, DLS84, FLM85, LSP82, TPS85]. Consequently, some effort has been spent in devising algorithms for election in networks in which some processors may have failed before the algorithm execution begins. In this paper, we consider only the case that no edge will fail during the execution of the algorithm. Edges may recover during execution of the algorithm; however, when we speak of the set of faulty edges, we refer specifically to the set of faulty edges before the process begins.

1.5 Adding Knowledge to the Basic Model

Even with our simple assumptions about faults in the basic network, it is obvious that no fault-tolerant leader-election algorithm can guarantee termination. Thus additional assumptions are needed.

1.5.1 Examples of Additional Assumptions

(1) The number of faulty edges is zero: In [GHS83] an algorithm was described, (of \(O(m + n\log n)\) message complexity) that implies electing a leader with a model identical to our basic model.

(2) The graph is complete: [KW85].
(3) The graph is a circle: [SG85]

(4) Each node knows all of its neighbors: \( O(n \cdot m) \).

(5) All edges have distinct names, and each node knows the names of its edges: \( O(m^2) \).

(6) The network is synchronous: [G82].

1.5.2 Our Assumption

In addition to the basic model, we will assume that each node "knows" majority, which has a value of at least \( 1 + \frac{n}{2} \) and there is a non-faulty component (a component of the graph induced by the non-faulty edges) with a size (number of nodes) of majority or more. From the practical point of view, this is a realistic assumption: though the nature of "real life" network sizes is to change from time to time, in many cases it will be a long time before the variable majority must be updated.

1.6 Algorithm Performance

We present a fault-tolerant leader election algorithm that terminates with the additional assumptions: (1) the set of faulty edges does no increase during execution of the algorithm; (2) all nodes know majority; (3) there is a non-faulty component with a size of at least majority; and (4) at least one node in the majority component starts the algorithm. Our algorithm guarantees to find exactly one leader in the majority component and all nodes in the majority component will eventually know the leader's identity and a way to reach the leader. The message complexity of the algorithm is \( O(n^2 + m) \) \( O(n^2) \) if no parallel edges exist.) Node space complexity for our algorithm is \( O(\log \text{MaxId} + \text{MaxNodeDegree}) \).

1.7 Organization of this Paper

Section 2 contains a step-by-step development of the algorithm, as well as the motivation and "hand-waving" proof of each property desired. Section 3 contains the formal algorithm. Section 4 and 5 contain formal proofs of properties and formal complexity analysis respectively. This sections (4, 5) are omitted from this extended abstract.
2. Informal Algorithm Descriptions

We choose to present our algorithm in terms of tokens (see e.g. [KKM85]). A token can be viewed as a programmed process that is able to move from node to node through an edge. If the edge is faulty, then the token is lost. Each node operates as follows: it gives control to the next token available in order to perform the token algorithm. This algorithm may use the node's local memory. The token returns control to the node in one of these cases: when the token leaves the node; when the token stops; or when the token wants to wait for some condition of the node's local variables to occur. This procedure occurs every time there are tokens available and no token has control over the node. For this purpose, an available token is either one which is waiting as a message in some pool of unserviced messages, or a waiting token for which the token's desired condition is satisfied.

The reason we choose to present our algorithm in terms of the tokens' algorithm rather than in terms of a node's algorithm is that it simplifies understanding, analysis and proofs. The token algorithm can be simulated by a node algorithm as follows: (1) each node has a copy of the tokens' algorithms; (2) the process of a token travelling an edge is simulated by sending a message which contains all information that should be carried by this token, along with the state of the token algorithm.

We now informally develop the algorithm. First we describe the general scheme. Then we develop the algorithm step by step.

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 his tree have an associated KingdomId which is the KING's Id. The KING's role is to expand his "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, he starts a traversal "pulse." In each pulse, the king traverses only tested edges and eventually returns to his root. While the KING traverses these edges, he is rebuilding his territory tree. When the KING enters "enemy territory" (a node with a different KingdomId), the Ids of the tree and the KING are com-
pared in order to find out who wins. If the KING loses, he becomes a "loser," which eventually causes the KING's "death." Otherwise, the KING assigns his Id to the node's King-domId, thereby annexing the node to his "territory" tree.

It is obvious that any implementation of the above scheme will have a worst-case behavior of $O(n^2+m)$ messages complexity. Thus, our objective is to have no worse behavior than that, i.e. $O(n^2+m)$.

The most common algorithm used for distributed traversal of a graph is the Depth First Search (DFS), specifically that of Tremaux (see e.g. [E79]). The message complexity may be as bad as $O(m)$ per DFS; this is too costly for our purposes. To avoid this, every time a non-tree DFS edge is discovered, it is blocked. No KING will ever travel an edge that has been blocked. However, the edge may be travelled by other KINGS while it is in a process of being blocked. We will show how to avoid this in order to get:

$$\text{total messages for blocking edges bounded by } 2m$$  

Let us next consider the problem of termination. In order to have termination, we would like to have the following property:

once a KING has a majority, no other KING will ever have a majority  

In order to determine the kingdom's size, a KING carries with him a KingCounter which is set to zero at the beginning of each "pulse," and incremented with each node the KING occupies. But this is not sufficient to establish (2.2). When the KingCounter of one king indicates a majority, it may happen that other KINGS have annexed some of his territory, and hence achieved a majority. To solve this problem, and other problems which are discussed below, we add the concept of "freezing" nodes. So now a KING's pulse will consist of two DFS traversals: one in status FREEZE, and the second in status MELT. In status FREEZE, the KING "freezes" each node he occupies in order to get a frozen tree. If the KING's KingCounter indicates a majority, he becomes a "leader" and thereby the "strongest" KING. After completing the first DFS, the KING changes status to MELT. The role of this status is: (1) to update all relevant nodes' variables (e.g. Counter); (2) to unfreeze the frozen territory tree using reverse DFS by
each time unfreezing a leaf in the current frozen tree. This reverse DFS ends in the
tree's root, at which time the **KING** changes status to **SLEEP**. He will then "die" if he has
ever encountered a stronger kingdom, or he may start another "pulse" if and when he
finds a **SCOUT's** notification.

The idea is when a **KING** visits an "enemy" frozen node, he will wait until the node
is unfrozen. This may cause two problems: (1) deadlock -- there may be a "circle" of
**KINGS**, each waiting for another **KING**; (2) a "frozen" node may contain many waiting
**KINGS**. We do not allow this in order to limit the node's local memory. To solve the first
problem, the **KING** will wait only if he is "stronger"; otherwise, he retreats as a "loser."
This implies that no such deadlock can occur. The second problem can be solved by
observing that there is no need for more than one **KING** to wait in one node. If a **KING**
Enter a node with a waiting **KING**, the weaker **KING** immediately retreats as a "loser."
When a node becomes "unfrozen," the node's **Counter** is updated; thus, a waiting **KING**
may discover that he is no longer the 'stronger. For this, the definition of "stronger"
must be extended to include "leadership." This establishes property (2.2).

Besides achieving property (2.2), the freezing mechanism makes it trivial to estab-
lish property (2.1). When a **KING** in status **FREEZE**, say **A**, discovers a non-DFS tree edge
\( xy \) -- i.e. \( x \) and \( y \) are in his frozen tree -- he will block that edge. Observe that while the
**KING** is doing this, there may be no other **KINGS** on that edge. Otherwise, either \( x \) or \( y \)
cannot be in **A**'s territory. Thus, from now on we can consider each DFS to cost no
more than \( 2n \).

Now, we would like to save DFS's. If a **KING**'s "pulse" is caused each time an edge is
tested successfully, then the total number of one **KING**'s messages may be as bad as
\( \Omega(n \cdot m) \) \( \Omega(m) \) edges cause \( \Omega(n) \) DFS messages.) In order to avoid this, we would like
to have the following property:

\[ O(1) \text{ of } \text{KING's DFS's cause a death of at least one KING} \quad (2.3) \]

This will help us to show that the death of \( k-1 \) **KINGS** costs \( O(kn) \) messages, which
implies a total cost of \( O(n^2) \) non-blocking DFS messages.
Because of the mechanism of "freezing" we have described, we can even establish:

\[ \text{every KING's pulse causes a death of at least one KING} \]  \hspace{1cm} (2.3.1)

Thus we need for every SCOUT's notification to a KING not to cause a wasted "pulse." In order to establish this, the first idea is that the SCOUT will compare the KingdomId of its start node \( x \) with the neighbor \( y \). If the SCOUT finds that the neighbor \( y \) is in the same territory as \( x \), then he will not notify the KING. We face a difficulty when the SCOUT discovers that the KingdomId of \( y \) is not the same as the KingdomId that he brought with him from \( x \). The difficulty is that, there is no way to compare \( x \)'s and \( y \)'s KingdomIds at the same time. It may happen that while the SCOUT is travelling from \( x \) to \( y \), the KingdomId of \( x \) (and maybe, also of \( y \)) are being changed; when the SCOUT gets to \( y \), he may be carrying an irrelevant KingdomId. For clarity, we partition the different actions of a SCOUT into three statuses: TEST, COMPARE, and CLIMB, which tests an edge, compares kingdoms, and climbs to the tree's root, respectively. In order to establish (2.3.1), we would like to have the following property:

at the time a SCOUT with status CLIMB leaves a node \( y \),

\[ \text{it is guaranteed that} \]

\( y \) is included in a larger tested component than \( y \)'s Counter indicates.

To establish this, the TEST status is responsible for marking the edge "tested" at both end points. This is done by a round trip from \( x \) to \( y \) and back. This then guarantees that the edge will be explored by a complete "pulse" if one should occur before the SCOUT in status CLIMB leaves \( y \). When the SCOUT ends status TEST, it starts status COMPARE, which is as described above. The COMPARE status ends in the node \( y \) either by the SCOUT vanishing when it finds the kingdoms of \( x \) and \( y \) are the same, or else by changing status to CLIMB. At this point, if \( y \) is frozen, a SCOUT in status CLIMB will wait until \( y \) is unfrozen, in order to allow the possibility for edge \( xy \) to be explored. If and when the node becomes unfrozen, the SCOUT checks the mark on \( xy \) to see if it has been explored. If \( xy \) was explored, then the SCOUT vanishes. Otherwise, the SCOUT leaves \( y \), climbing toward the root. This establishes property (2.3.2). Assume the contrary, i.e. a SCOUT in status CLIMB leaves a node \( y \) when the Counter of \( y \) has already
been updated by the KING in status MELT to include \( x \) in the kingdom while the edge \( xy \) remains unexplored. As (by the nature of DFS) \( xy \) must have already been marked tested when the KING was at node \( x \), and should have been explored, unless the KING became a "loser." The node \( z \), which caused the KING to become a "loser," was in the same component as \( y \) though not considered in \( y \)'s Counter. Contradiction.

This does not yet imply (2.3.1). The problem we face now has the same nature as the problem we faced before: the SCOUT may have been so slow that he notifies the KING after the KING has already completed at least one more "pulse." Thus the SCOUT may cause a wasted "pulse." In order to solve this problem, the SCOUT in status CLIMB carries with him ClimberDate, which is set to \( 1 + \text{Counter} \) in his start node \( y \). (2.3.2) implies the following property:

\[
\text{ClimberDate is a lower bound for } y \text{'s tested component size. (2.3.3)}
\]

The notification the KING gets in the root is the value of ClimberDate. Only if this value is greater than the KingCounter will the KING start a "pulse," knowing that he will visit an "enemy" node. This completely establishes property (2.3.1), thus also establishing the complexity desired for KING's messages.

One final modification is needed in order to save SCOUT messages. Status TEST and COMPARE obviously cost no more than \( 8m \) (each of the \( 2m \) endpoints may send two TEST messages, and one COMPARE message.) The problem lies in the SCOUT's status CLIMB algorithm, as described above. \( O(m) \) edges (not necessarily from different kingdoms) may cause \( O(n) \) SCOUTs in status CLIMB, which leads to a total cost of \( O(mn) \). This is solved as follows: each node contains a local variable, \( Date \), initially 1. When a SCOUT in status CLIMB visits a node, the node's \( Date \) is tested; if it is not greater than ClimberDate, the SCOUT will vanish. Otherwise, \( Date \) is updated with ClimberDate. (2.3.3) implies the invariant that a node's \( Date \) is a lower bound for the tested component in which the node is included. Thus the notification for a KING will come from the \( Date \) of the root. This implies that (2.3.1) remains valid, but this leads us to an interesting message-complexity analysis. By the definition of our \( Date \) and ClimberDate mechanism, we get:
a node $y$ sends a \texttt{CLIMB} message only if its $Date$ increases \hfill (2.4)

Observe that $Date$ must have values in the range 1, 2, \ldots, $n$, and can never decrease.

Thus, each of the $n$ nodes sends less than $n$ "climber" messages. Hence, the total number of \texttt{CLIMB} messages is less than $n^2$.

Finally, consider node's space complexity. Under the model definition and the nature of the problem, a node's space memory is at least:

$O(\log \text{MaxId} + \text{MaxNodeDegree})$. So, the best we can hope for is $O(\log \text{MaxId} + \text{MaxNodeDegree})$. It is not hard to implement our algorithm to have this. $O(1)$ of \texttt{KINGs}, \texttt{Id}, \texttt{KingdomId}, \texttt{Counter}, \texttt{ClimberDate} and $Date$, each need no more than $\log \text{MaxId}$ bits, and each edge in a node needs only $O(1)$ bits for edges' marks and additional one bit for a possible waiting \texttt{CLIMB} token.
3. The Formal Algorithm.

Each node contains:

Constants: \( \text{Id} \) = the node identification ; \{The only line which differs from node to node\}

\[ \text{Majority} = 1 + \left\lfloor \frac{n}{2} \right\rfloor ; \{n - \text{total number of nodes}\} \]

Variables: \( \text{KingdomId} \) initially \( \text{Id} \) ;
\( \text{Date} : 1 .. n \) initially 1 ;
\( \text{Counter} : 1 .. n \) initially 1 ;
\( \text{Flag} : ("", \"Looser", \"Leader\") \) initially "" ;

Tokens: \( \text{KING} \) with status \( \text{SLEEP} \) and associate triple \( (\text{KingFlag}, \text{KingId}, \text{KingCounter}) \) initially \( (\"", \text{Id}, 1) \) ;

Each edge entry in a node has

Variable: \( \text{mark} : (\text{danger}, \text{unexplored}, \text{frozen}, \text{melted}, \text{blocked}) \) initially \( \text{danger} \) ;

Token: \( \text{SCOUT} \) initially with status \( \text{TEST} \) ;

When node wakes up \{spontaneously or by token's arrival\}, it sends over \( \text{SCOUT} \) token over the edge with which it is associated;

\( \text{SCOUT} \) \{A \( \text{SCOUT} \) has 3 statuses: (i) \( \text{TEST} \) (ii) \( \text{COMPARE} \) (iii) \( \text{CLIMB} \) \{When the \( \text{SCOUT} \) changes its status, it starts to execute the algorithm for the new status.\} \{Sent by a node when wakes up. Initial status is \( \text{TEST} \)\}

(i) \( \text{TEST} \) \{initially\}:
Let the \( \text{SCOUT} \) be originated in node \( x \), and sent to \( y \).
When arrives to \( y \), Mark \( xy \) \( \text{unexplored} \) if marked \( \text{danger} \);
Return back to \( x \), when arrives \( x \), Mark \( xy \) \( \text{unexplored} \) if marked \( \text{danger} \);
Change status to \( \text{COMPARE} \) ;

(ii) \( \text{COMPARE} \) \{from status \( \text{TEST} \)\}:
\( \text{CompareId} := \text{KingdomId} \text{of} \ x \); Carry with you. \( \text{CompareId} \), travel again to \( y \);
When arrives \( y \), If \( \text{KingdomId} = \text{CompareId} \) then stop \{vanish\} ;
Change status to \( \text{CLIMB} \) ;

(iii) \( \text{CLIMB} \) \{From status \( \text{COMPARE} \)\}:
\{Initiate\} While \( y \) is frozen do Wait ;
\ If edge \( xy \) in \( y \) is not marked \( \text{unexplored} \) then stop \{vanish\} ;
\( \text{ClimberDate} := 1 + \text{Counter} \) ;
\{Climb up\}
\ While \( \text{ClimberDate} > \text{Date} \{\text{in} \ y \} \) do begin
\( \text{Date} := \text{ClimberDate} \);
\ If no edge in \( y \) marked \( \wedge \text{up} \) then stop \{vanish\} ;
\ Travel the-edge \( yz \) marked \( \wedge \text{up} \);
\( y := z \) ;
\ end \{While\}
Stop ;
KING

{A KING has one of the statuses: (i) SLEEP (ii) FREEZE (iii) MELT}

(i)  SLEEP {Initially and after MELT}:
While Date ≤ Counter and Flag ≠ "Looser" and KingdomId = KingId do
Wait;
If Flag ≠ "Looser" and KingdomId = KingId then Change status to FREEZE else stop {vanish}

(ii) FREEZE {From SLEEP status}:
{Initiate} Let r be the root of kingdom KingdomID, x:=r ; KingCounter:=0
{Main} As long as your status is FREEZE do begin
{Take over} FREEZE the node and increment KingCounter;
If KingCounter ≥ Majority then KingFlag := "Leader" ;
If x ≠ r then mark the edge you came from by ↑p ;
{Search} Repeat
{No way?} While no edge in x marked unexplored or melted do begin
{Finish?} if x = r then change status to MELT ;
{Go up} Travel the edge xy marked ↑p ;
x:=y ;
end ; {While}
{Down} Let xy be an unexplored or melted edge, Mark xy frozen and travel to y ;
{Old node?} If y is frozen and belong to your kingdom then begin
{Block} Mark xy in y by blocked ;
{Go back} Travel back to x ; Mark xy in x by blocked end {if}
{New node} else x:=y
{Til new} until x:=y
{War} Definition: Kingdom with (Flag1, Id1 ) is stronger than a kingdom with (Flag2, Id2 ) iff (Flag1="Leader" or (Id1>Id2) and Flag2≠"Leader")
{Wait} While x is frozen and no stronger kingdom around do Wait ;
{Lost?} If x is stronger or contains stronger king then begin
{Back} Travel the edge xy you came from ; Mark xy in y melted ;
{Unfreez} KingFlag := "Looser" ; Change to MELT ;
end {if} {else go to beginning of the loop to take over the node x}
end {Main loop}

(iii) MELT {From FREEZE status}:
{Main} As long as status is MELT do begin
{No way?} While no frozen edges in x do begin
{Update node} Counter:=KingCounter ; Flag:=KingFlag ;
If Date < Counter then Date:=Counter ;
{Unfreez node} Unfreez x ;
{Finish? Sleep!} if no edge marked ↑p then change status to SLEEP ;
{Up} Travel the edge xy marked ↑p ;
{Unfreez edge} Mark xy in y melted ; x:=y ;

Technion - Computer Science Department - Technical Report CS0409 - 1986
end; {While}
{Down}    Let $xy$ be a frozen edge, Travel to $y$; $x := y$
end {Main loop}

Acknowledgement

We would like to thank Charles R. Martin for his linguistic help.

REFERENCES


