OPTIMAL DISTRIBUTED t-RESILIENT ELECTION
IN COMPLETE NETWORKS

by

S. Kutten, Y. Wolfsthal and S. Zaks

Technical Report #430
August 1986
OPTIMAL DISTRIBUTED \( t \)-RESILIENT ELECTION IN COMPLETE NETWORKS

Shay Kutten, Yaron Wolfstahl, and Shmuel Zaks

Department of Computer Science
Technion, Haifa, Israel

ABSTRACT

We study the problem of electing a leader distributively in an asynchronous complete network, in presence of faults that occurred prior to the execution of the election algorithm. Failures of this type are encountered, for example, during a recovery from a crash in the network. For a network with \( n \) processors, \( k \) of which start the algorithm and at most \( t \) processors might be faulty, we present an algorithm that uses at most \( O(n \log k + k \cdot t) \) messages. We prove that this algorithm is optimal. We also present optimal results for the cases where neighbor's identities are known, and where edges may fail. It is interesting to see that the order of the message complexity of a \( t \)-resilient algorithm is not always higher than the complexity of a non-resilient algorithm. The \( t \)-resilient algorithm is a systematic modification of an existing algorithm for a fault-free network. In the complexity analysis we use a new technique that we believe will prove helpful in analyzing fault-tolerant algorithms.
1. INTRODUCTION

The problem of leader election in asynchronous distributed systems is widely studied (e.g. [G, L77, KKM85, KMZ83]). In this problem 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.

The basic model consists of a distributed complete network of \( n \) identical processors (except for their having distinct identities), \( k \) of which start the algorithm. Every pair of processors is connected by a bidirectional communication line. 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. We view the communication network as a complete undirected graph, where nodes represent processors and edges represent communication lines. To measure the efficiency of an algorithm, we use the common measure of the maximal possible number of messages transmitted during any execution, where each message contains at most \( O(\log \text{MaxId}) \) bits (see e.g. [GHS83]).

Real systems, however, are subject to faults of different types.

Consider the possibility that some nodes in the network may be faulty. A faulty node is a node which never transmits a message, and every message transmitted to it is lost. Because our basic model is asynchronous, it cannot be distinguished whether a message is delayed or lost.

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 failures are also hard or impossible to cope with [F83, FLM85]. Actually, the presence of these faults motivates the leader election, as a method to reorganize the network [G82]. (Reliable hardware equipment makes failures of the most general type quite rare [G82].) We assume that no failure occurs during the execution of the algorithm. Nodes may recover during execution of the algorithm.

Even with these assumptions about faults, it is obvious that no fault-tolerant leader-election algorithm can guarantee termination in the case of a general graph. Thus additional assumptions are needed. This includes, for example, knowledge about synchrony in the network ([G82]), its topology ([KW84, SG85]), its size ([SG85]), or requiring a spanning tree construction (and not leader election), without termination detection [K85]. In this paper we assume that the network is complete, namely, every two processors are connected by a communication line.
We develop fault-resilient algorithms. However, we believe that our main contributions are the following: (1) the tools we used to make an existing algorithm fault resilient are rather general, and can be applied in similar situations; and (2) we introduce a method we have found useful in analyzing the complexity of such algorithms.

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

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

In presence of faults we choose to use more than one token in a tree. Most decisions are done locally (without consulting the temporary leader), and contradicting actions are prevented by the following method: each token acts according to the global information it has about the state of the tree. Thus the temporary leader must be consulted when this information alone does not suffice to make the decision. This information is the identity of the temporary leader of the tree and a bound on the size of the tree. 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, even though duplicating messages is used.

A $t$-resilient algorithm is an algorithm that yields the correct answer when at most $t$ nodes are faulty. We modify an existing election algorithm ([AG85]) (and similar algorithms such as [AG84, H84]) to achieve a $t$-resilient election algorithm (for any $t < \frac{n}{2}$). The resulting algorithm uses at most $O(n \log k + k t)$ messages during any possible execution, and we prove that bound to be the best possible. Our algorithm improves on existing resilient algorithms in terms of message, bit, space and computational complexity measures (see [FLP85, KW84]). Note that when $t$ is $O(\frac{n \log k}{k})$ the message complexity is $O(n \log k)$, the same as for reliable networks [KMZ83]. On the other hand for $t > \Omega(\frac{n \log k}{k})$ the message complexity of every $t$-resilient algorithm is higher than the message
complexity of election in reliable networks. We also present optimal algorithms for the cases where edges may fail and where the identities of the neighbors are known.

In order to analyze 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.

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 $O(n \log \text{MAX}_ID)$ bits, the bit complexity is $O(n^3 \log \text{MAX}_ID)$ and the message complexity, in terms of our model, is $O(n^3)$ (our result implies an $O(n^2)$ for this case). An $\Omega(n \log n)$ and $O(n \log n)$ lower and upper bounds for 1-resilient election in a ring is found in [SG85].

2. INFORMAL DESCRIPTION OF THE $t$-RESILIENT ALGORITHM

In this section we describe an algorithm that elects a leader in a complete network with $n$ nodes, at most $t$ of which may have failed ($t < \frac{n}{2}$). We modify the simple algorithm of [AG85] (which is similar to the ones in [H84, AG84]) that elects a leader in a nonfaulty 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 1 and may only grow. The size and identity of node $A$ are denoted by $\text{size}(A)$ and $\text{id}(A)$, 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.

c. a REJECT message, originated by a node that refused to be 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 $(\text{size}(A), \text{id}(A))$ and the information it finds in $C$ and $B$, as follows:
Case 1. \(((\text{size}(A), \text{id}(A)) > (\text{size}(C), \text{id}(C)))^2\):

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 \((\text{size}(D), \text{id}(D)) > (\text{size}(A), \text{id}(A))\), 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. \(((\text{size}(A), \text{id}(A)) < (\text{size}(C), \text{id}(C)))\):

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.

In our \(t\)-resilient version each king owns \(t+1\) tokens that are initially sent to different neighbors. The actions taken by a token of node A that arrives at a master C of a neighbor B of A, depend on \((\text{size}(A), \text{id}(A))\) and the information it finds in C and B, as follows:

Case 1. \(((\text{size}(A), \text{id}(A)) > (\text{size}(C), \text{id}(C)))\):

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 \((\text{size}(D), \text{id}(D)) > (\text{size}(A), \text{id}(A))\):

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 \((\text{size}(A), \text{id}(A))\) is still smaller than \((\text{size}(D), \text{id}(D))\) (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 \((\text{size}(A), \text{id}(A)) > (\text{size}(D), \text{id}(D))\) 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

---

\(^2\) Lexicographically, namely size(A) > size(C) or size(A) = size(C) and id(A) > id(C).

\(^3\) Note that a node may "cease to be a king" more than once; we use this convention throughout the paper.
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.

### 3. PROOF OF CORRECTNESS

We now prove the correctness of the algorithm, summarized in Theorem 1. Lemmas 1, 2 and 3 ensure that no deadlock will be caused by the faulty processors. Lemma 4 and Corollary 1 ensure that at least one king remains. Lemma 5 helps to prove that only one king remains. Corollary 2 is used to guarantee termination detection. Henceforth we consider a given execution of the algorithm.

**Lemma 1:** At any given time during the execution, let A be a king that has more than \(r\) edges leading to nodes from which it has not received an ACCEPT message. If A is waiting for a message, and has not sent a leader announcement message, then eventually either A will receive a leader announcement message, or one of its tokens will return.

**Proof:** King A can send a leader announcement message only when its size increases. This implies that one of its token will return. Thus we may assume that no such message will ever be send by A. We can also assume that no leader announcement is ever received by A (otherwise the lemma trivially holds). Thus no leader announcement is ever sent by any node in the whole execution. Hence no token will be stopped in any node as a result of a leader
announcement message received by that node. If node A is involved in a war with some node B (in Cases 1.1 and 2.2), then one of A’s tokens has already returned once from B and B’s master; hence they are both nonfaulty, and from Case 1.1 in the algorithm the token will return from B in any case. Otherwise A’s t+1 tokens are sent to t+1 distinct nodes, at most t of which are faulty. Therefore at least one token arrives at a nonfaulty node B. If B is its own master then the token returns to A. Otherwise it proceeds to B’s master which is also nonfaulty (since it sent at least one message in order to annex B). The token then returns to A through B and the lemma follows.

Lemma 2: Let A be a king which started a war with some node C because one of its tokens returned with an irrelevant REJECT message from some node B. The war eventually terminates, and at that time either A ceases to be a king, or this token returns from B with an ACCEPT message, causing size(A) to be incremented.

Proof: If king A receives a leader announcement message, or a relevant REJECT message (from any node) then it ceases to be a king, and the lemma clearly holds. Thus consider the first irrelevant REJECT message which is received by A from node B. The next token sent by A to B starts a war with B. A REJECT message which is received from B during this war can be irrelevant only if size(A) has increased since it sent the last ASK message to node B (Cases 1.1. and 2.2). However, size(A) may increase only when a token returns to A with an ACCEPT message, but such a token returning to king A during the war with B is suspended until the war with B is over. Hence the number of irrelevant REJECT messages received by A from B during the war is bounded by t, and the total number of REJECT messages received by a king from any node is t+2.

Lemma 3: Every king eventually either has its size > n/2 or it ceases to be a king.

Proof: Assume, to the contrary, that node A remains a king with size(A) ≤ n/2. Consider the time A reaches its final size s, by receiving a returning token with an ACCEPT message. If king A is now involved in a war, then, by Lemma 2, this war will eventually end, and either A either will cease to be a king or its size will increase, a contradiction. Therefore A is not involved in a war and all the edges over which it has received answers leads to nodes in its domain. Since A is always in its own domain, and since size(A) ≤ n/2, it follows that the number of these edges is at most n/2 - 1. There are at most (actually, exactly) t < n/2 other edges over which A sent token that has not yet returned. Thus the total number of edges over which A has sent its tokens is less than n - 1. Hence it has at least one edge over which it has not yet sent one of its tokens, and the returning token is sent over this edge. By Lemma
1. king $A$ must either receive a leader announcement message or one of its $t + 1$ tokens must return. If either a leader announcement message or a relevant REJECT message is received, then $A$ ceases to be a king, a contradiction. If an ACCEPT message is received then $size(A)$ is incremented, a contradiction. If an irrelevant REJECT message is received then a war starts, and by Lemma 2 this war must end and then either $A$ will cease to be a king or its size will increase, a contradiction. This completes the proof.

**Lemma 4**: Assume king $A$ ceases to be a king as a result of a message originated by king $C$. Then $(size(C), id(C)) > (size(A), id(A))$ from the time this message is received by $A$.

**Proof**: From the time this message is received by $A$, $size(A)$ remains unchanged while $size(C)$ may only increase, and the lemma follows.

**Corollary 1**: At least one node always remains a king.

**Proof**: Assume, to the contrary, that all nodes cease to be king, and consider the final sizes. The node $A$ for which $(size(A), id(A)) > (size(B), id(B))$ for any other node $B$, clearly contradicts Lemma 4.

**Corollary 2**: At least one king has eventually its size $\geq \frac{n}{2}$.

**Proof**: Follows from Corollary 1 and Lemma 3.

**Lemma 5**: For every $l \geq 2$, if there are $l - 1$ kings whose final size is larger than that of king $A$, then $size(A) \leq \frac{n}{l}$.

**Proof**: If a node $B$ of the domain of $C$ joins the domain of $A$, then $C$ ceased to be a king and from that time $(size(C), id(C)) < (size(A), id(A))$ (see Lemma 4). Thus domains of equal sizes (even viewed at different times) are disjoint. The lemma follows.

(A similar lemma appears in [G, H84, AG85].)

**Theorem 1**: Every execution of the $t$-resilient algorithm in a complete network with no more than $t$ faulty nodes terminates, and exactly one node declares itself a leader.

**Proof**: By Corollary 2 some king must eventually have its size larger than $\frac{n}{2}$. By Lemma 5 no other king will reach
such size. This unique king then sends a leader announcement message over all its edges, and the execution terminates.

4. COMPLEXITY ANALYSIS

For the complexity analysis we define the amortized message complexity as the actual message complexity plus the potentials 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 an 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. As in the previous section, we assume a particular execution of the algorithm.

Define a reception event at node A to include a reception of a returning token of node A at node A, the local computation which node A then performs, and the message (or messages) which A sends as a result. We use the following notations denoting quantities up to (and including) the τ-th reception event at a node A.

1. \( \#send(A, \tau) \) is the number of times a token of node A was sent by A. This is A's part in the actual message complexity of a prefix of that execution.

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

3. \( \#susp(A, \tau) \) is the number of times a token has returned to A and was suspended as a result of a war in which A was involved.

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

5. \( inc(A, \tau) \) is the increment in \( size(A) \) 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 the node with which it is currently involved in a war since the beginning of this war (if such exists).

7. \( size(A, \tau) \) is \( size(A) \) after the \( \tau \)-th reception event.

8. \( potential(A, \tau) = \#held + (size - \#wars) + (size - \#susp) + (inc - IRR) \)

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

Lemma 6: Let A be a node that initiated the algorithm and is still a king after its \( \tau \)-th reception event. Then
Proof: First note that

\[ \text{size} + t + 1 > \#send + \text{potential} \quad (*) \]

since each time an irrelevant REJECT is accepted, the size has been incremented at least once since this token was sent. We continue by induction on \( t \).

Induction Basis: \((t = 0)\) On initiating the algorithm, node \( A \) sends \( t + 1 \) tokens. By that time:

\[ \#send = t + 1; \text{size} = 1; \#wars = \#susp = \#held = \text{inc} = \#IRR = 0, \]

and \((*)\) holds.

Inductive Step: Assuming \((*)\) holds after the \( t \)-th reception event in node \( A \), we show that it holds after the \( t + 1 \)-st reception event. Note that only ACCEPT and irrelevant REJECT messages received in \( A \) need to be considered.

The cases where an ACCEPT or a REJECT message is received while \( A \) is not involved in a war are Case 1 and 3 (below), respectively. If \( A \) is involved in a war then, the message it receives may be received either from the node with which it is involved in a war (say node \( B \)), or from some other node. Cases 2 and 7 below deal with the cases that an ACCEPT or REJECT message are received from a node other than \( B \). Case 4 deals with the case where a REJECT message is received from node \( B \). In cases 5 and 6 ACCEPT messages are received from \( B \), but in case 6 some other irrelevant REJECT message is suspended in \( A \), while in case 5 no other irrelevant REJECT message is suspended in \( A \).

Case 1: A token that successfully annexed a neighbor (i.e. a token with an ACCEPT) is received and \( A \) is not involved in a war.

In this case the token is sent for another annexing attempt, both \text{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.

In this case the token is suspended, \text{size}, \#sus, \#hel and inc are incremented and \((*)\) holds.

Case 3: \( A \) is not involved in a war and for some node \( B \), an irrelevant REJECT message is received by \( A \) from \( B \) for the first time.

In this case a war starts, the token is immediately sent back towards \( B \) and both \#wars and \#send are incremented.

Case 4: A token with an irrelevant REJECT message is received from a processor with which \( A \) is involved in a war.

In this case the token is sent back there immediately to continue the war, and both \#send and \#IRR are incremented.
Case 5: A token returns with an ACCEPT from some processor with which \(A\) is in a war, and no other REJECTs are suspended. In this case 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, \(inc-#IRR = inc-#IRR = 0\), so by (**), the right-hand side does not increase.

Case 6: A token returns with an ACCEPT from some processor with which \(A\) is in a war, and some other irrelevant REJECT message from some node, \(C\), is suspended in \(A\).

In this case, the token returned from \(B\) is suspended and the one returned from \(C\) is sent there again. size, \#susp\ and \#wars \#send\ are all incremented. Also, \(inc-#IRR = 0\)

Case 7: An irrelevant REJECT is received while \(A\) is involved in a war with another processor. Both \#susp\ and \#held\ are incremented, and (***) still holds.

Corollary 3: 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 we can use a technique similar to [G, H84, AG85], using Lemma 5, and get:

Theorem 2: The number of messages used by the \(t\)-resilient algorithm is \(O(n \log k + k \cdot t)\).

Sketch of Proof: Let \(s\) be a final size of a king that initiated the algorithm. By Corollary 3 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 return from it). 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 5 the total number of messages is bounded by \(n-1 + 4 \cdot (3 \cdot n \cdot (1+1/2+1/3+\ldots+1/k) + k \cdot t) = O(n \log k + k \cdot t)\).

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 \(\Omega(n \log k + k \cdot t)\).

Proof: The term \(\Omega(n \log k)\) follows from the lower bound of \(\Omega(n \log k)\) messages for the problem of election in complete reliable networks [KMZ83] and from the fact that the number of the nonfaulty processors \(n-t\) is larger than \(\frac{n}{2}\). For the lower bound of \(\Omega(k \cdot t)\) consider a node which initiated the algorithm. It must send at least \(t+1\) messages, as it may occur that it is the only node to wake up and that the first \(t\) messages were sent to faulty nodes.
Assume now that actually there is no faulty node, and that \( k \) nodes initiate the algorithm. Since an adversary can delay all the messages, each of these \( k \) nodes must act as if it alone initiates the algorithm.

5. RESULTS FOR OTHER MODELS

Our results can be extended to other models. For the case where every node knows its neighbors' identities it follows immediately that:

**Theorem 4:** The message complexity of election in complete networks containing at most \( t \) faulty processors where all identities are known to all nodes is \( \Theta(k) \).

In the algorithm achieving the upper bound of \( O(k) \) we let the kings compete by capturing only \( t + 1 \) nodes out of the \( 2t + 1 \) nodes with the highest identities, instead of half of the nodes.

For the case where edges may fail (instead of nodes), we have:

**Theorem 5:** The message complexity of election in complete networks containing at most \( t \) faulty edges (and nodes' id is not known to neighbors) is \( \Theta(MIN(n \log k + k t, t n)) \).

**Proof:** The proof of the lower bound is exactly like in Theorem 3. The upper bound when \( t \geq n - 2 \) follows from the [BK86]. (They assume that even if there are more than \( n - 2 \) faulty edges, there is a connected nonfaulty component containing at least \( \frac{n}{2} \) nodes.) For the case where \( t < n - 2 \) the algorithm follows. As in Section 2, a node which wakes up sends \( t + 1 \) tokens. A king, \( A \), which annexes at least one other node, \( B \), proceeds as in the algorithm for the case of faulty nodes. However the unused edges used to resend tokens which return with an ACCEPT, are both \( A \)'s edges and \( B \)'s edges. A node may thus receive \( A \)'s token first from \( A \), and then from \( B \). In the second case the token returns from the node to \( B \) (with neither ACCEPT nor REJECT) and proceeds to annex another node. The details are left to the reader. Note that \( A \) and \( B \), together with their neighbors to which nonfaulty edges lead, always form a majority of the nodes.

REFERENCES


1986.


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


