OPTIMAL DISTRIBUTED t-RESILIENT ELECTION IN COMPLETE NETWORKS[1]

Alon Itai, Shay Kutten[2], Yaron Wolfstahl and Shmuel Zaks[2]

Department of Computer Science
Technion, Haifa, Israel

ABSTRACT

We study the problem of distributed leader election 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 may be faulty, we present an algorithm that uses at most $O(n \log k + k\,t)$ messages. We prove that this algorithm is optimal. We also present an optimal algorithm for the case where the identities of the neighbors are known. It is interesting to note that the order of the message complexity of a t-resilient algorithm is not always higher than that of a non-resilient one. The t-resilient algorithm is a systematic modification of an existing algorithm for a fault-free network.

Index Terms: complete networks, distributed algorithms, election, fault-tolerance, message complexity.

---

[1] A preliminary version of this paper appeared as IBM research Report RC 12177, September 1986.

[2] Part of the work of this author was done while visiting IBM Thomas J. Watson Research Center.

1. INTRODUCTION

The problem of *leader election* in asynchronous distributed systems has been widely studied (e.g. [G, L77]). In this problem it is required that the nodes cooperate to elect one of them as the leader. The election problem, 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. Real systems, however, are subject to faults of different types, and this paper focuses on unreliable networks. A *t-resilient* algorithm is an algorithm that finds a leader when at most t nodes are faulty. In this paper, we develop t-resilient election algorithms. We believe, however, that our main contribution is to the understanding of the methods for making algorithms t-resilient.

The fault-free model consists of a distributed complete network of n identical processors, k of which start the algorithm spontaneously. Each processor has a unique identity, but no processor knows the identity of any other processor. Every pair of processors are 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. A node which does not start the algorithm spontaneously, joins the algorithm when it receives a message for the first time. We view the communication network as a complete undirected graph, where nodes represent processors and edges represent communication lines. To evaluate the efficiency of an algorithm, we use the usual measure of the maximal possible number of messages transmitted during any execution (see e.g. [GHS83]). Each message may contain at most $O(\log Max\_id)$

bits, where *Max_id* is the highest identity of a node in the network.

Note that the above assumptions are quite reasonable. Although in real-life networks not every two nodes are connected by a direct dedicated link, they are still connected somehow via the network. Moreover, in some networks the cost of routing a message between two nodes is about the same as that of a one-hop message, as long as the route between these nodes is known in advance. This route need not consist of identities of nodes. Instead, it may consist of numbers of links. Thus, it is also reasonable to assume that a node does not know the identities of its neighbors. We assume that each communication line satisfies the FIFO discipline. Note that this discipline can be achieved by using acknowledgements; *i.e.*, a node sends a message on a line only after receiving an acknowledgement for the previous message sent on this line.

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. For the general case where nodes can fail *during* the execution of an algorithm, no deterministic election protocol exists [FLP85, MW87]. Other types of failures are also hard or impossible to cope with [F83, FLM85]. (Fortunately, reliable hardware equipment makes failures of the most general type quite rare [G82].) Thus, additional assumptions are needed. These include, for example, knowledge about synchrony in the network [G82], its topology [KW84, SG86], or its size [SG86]. In our model, all faults are assumed to have occurred prior to the execution of the election algorithm (see also [BKWZ]).

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 $O(n^2)$ for this case). An $O(n \log n)$ upper bound for 1-resilient election in a ring (where neighbor identities are known and only an edge may fail) is found in [SG86]. The problem of designing resilient algorithms for graphs other than rings (using less than $O(n\,m)$ messages), and for more limited types of faults, is given there as an open problem.

We modify the election algorithm of [AG84], obtaining 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, where *k* is the number of nodes that started the algorithm. This bound is proved to be the best possible. Our algorithm improves on existing resilient algorithms (for the same fault model) in terms of message, bit, space and computational complexity measures (see last section of [FLP85], and [KW84]). Note that when *t* is $O(\frac{n \log k}{k})$ the message complexity is $O(n \log k)$, as in election algorithms 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 an optimal algorithm for the case where the identities of the neighbors are known.

2. DESCRIPTION OF THE ALGORITHM

Leader election algorithms usually are viewed as each processor starting the algorithm by being its own king, and the algorithm advances by processors surrender to one another, and agreeing on a unique leader. Each processor knows, at any given time, the edge leading to its

current king; in other words, it might belong to certain 'kingdoms' during the execution of the algorithm. Each king contains certain information about its kingdom; this information contains at least the size of the kingdom.

These election algorithms can be thought of as *token algorithms*. Taking this point of view, every processor that starts the algorithm sends a token (carrying its name), that traverses the network, in an attempt to 'kill' all other tokens in the network. In the presence of faults, we extend this idea and use more than one token per processor; More precisely, in the presence of at most t faulty processors each processor sends $t+1$ tokens, in order to insure that at least one of them will be processed. Most decisions are done locally (without consulting the temporary leader), and contradicting actions are prevented by the following method: A token of processor A acts according to the global information it has about the state of A; Thus A must be consulted only when this information does not suffice to make the decision. This information is the identity of A and a bound on the size of the kingdom of A.

We now present the algorithm that elects a leader in a complete network with n nodes, at most t of which may have failed ($t < \frac{n}{2}$). The algorithm is a modification of the simple algorithm of [AG84], which elects a leader in a reliable complete network. We first describe the algorithm of [AG84]. In this algorithm some nodes are candidates for leadership, called *king*s. 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 if the node awakes spontaneously, and 0 otherwise. The value of *size* may only

increase. 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 node also remembers (*highest.size*, *highest.id*), the highest pair (*size*, *id*) known to it (as long as it is a king, this is the value of its own pair), and the communication line from which it was received. As explained above, the algorithm is described as if each king owns one *token*, which is a process representing it, and carrying its size and identity. In order to help the explanation it carries also an additional message, which is one of the following:

a. A **join** message, originated by the node that owns the token.

b. An **accept** message, originated by a node that was annexed by the token.

The *t*-resilient algorithm will also use the following message:

c. A **reject** message, originated by a node that refused to be annexed by the token.

In order to annex a neighbor *B*, the token of a king *A* is sent from *A* to *B* with a **join** message. The token proceeds from node *B* to *B*'s master *C*, which may be *B* itself.

The following actions taken by the token depend upon *(size_A, id_A)* and the information found at *C* and at *B*.

Case 1. $(size\_A, id\_A) \geq (highest.size\_C, highest.id\_C)$[3]

the *C*'s status becomes *killed* and $(highest.size\_C, highest.id\_C) := (size\_A, id\_A)$. (

[3] Lexicographically.

or

and

$size\_A > high$

$id\_A = high$

does not join A's domain.) The token returns to node B. If by now a token of another node D has passed B and therefore $(highest.size\_B, highest.id\_B) > (size\_A, id\_A)$, 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. $((size\_A, id\_A) < (highest.size\_C, highest.id\_C)$:

The token is killed.

A token that returns safely repeats the process of attempting to annex a new neighbor. The algorithm terminates when the token of a node A notices that $size\_A = n$.

We now present our algorithm. Each processor may be either a king, in states king_search, king_battle or king_defeated, or a subject, in states subject_relay or subject_waiting. The possible changes of states is depicted in Figure 1.

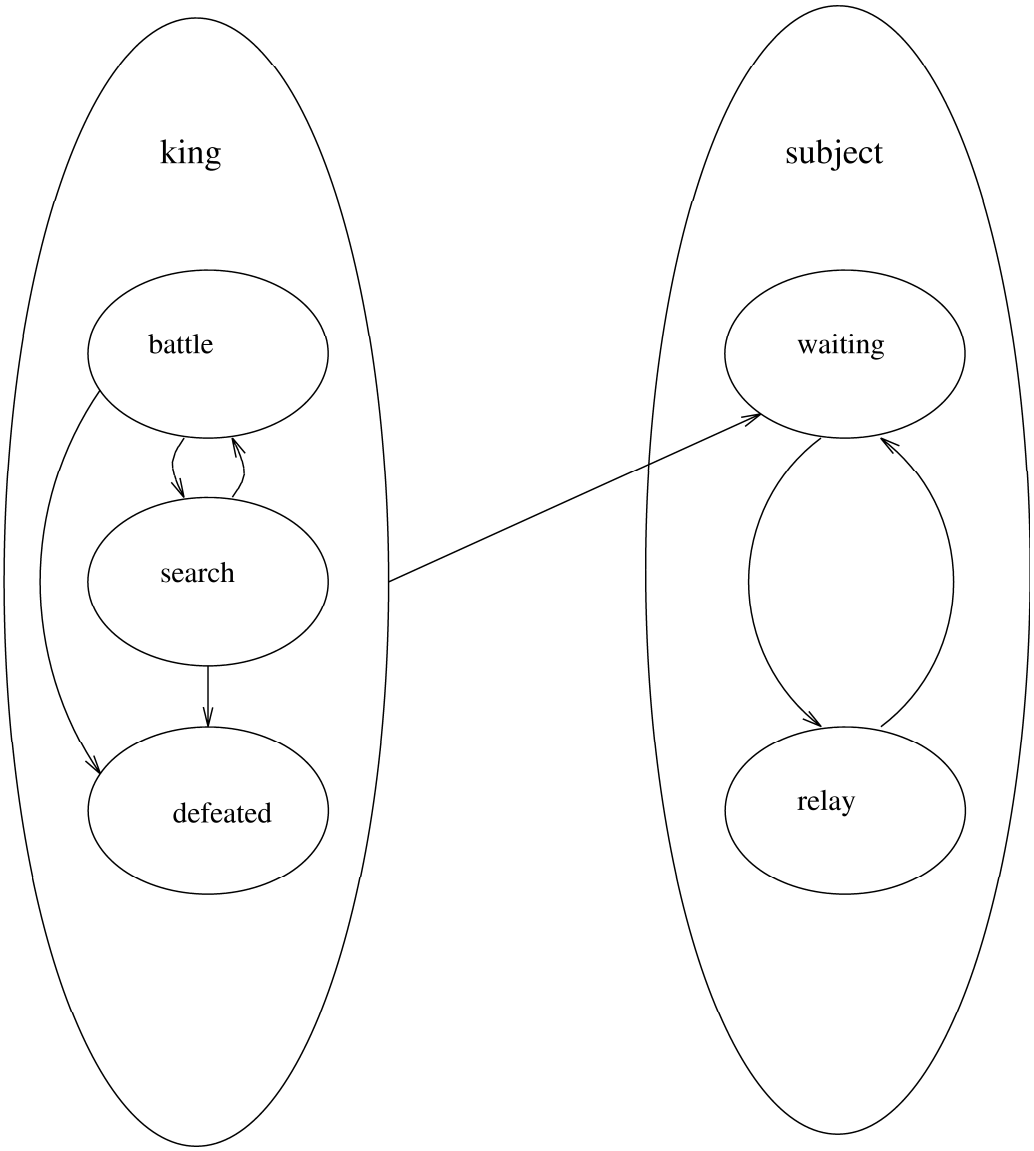

Figure 1: The states of a processor

The messages are of four types: **join**, **accept**, **reject** or **leader**. Each king owns $t+1$ tokens that are initially sent to different neighbors. The algorithms is message- driven, in the sense that each processor, after sending its first messages, waits until any message arrives at the pool, fetches any of these messages, and its reaction to this message may be doing some local

computation and sending other messages, and so on. To simplify matters we assume that a message that is sent to a processor eventually arrives at the pool, and the scheduler is fair, in the sense that each message in the pool is eventually read (for example, a round-robin scheduler); this assumption is made because, in certain cases, when a message is read, the processor decides to return it to the pool of messages, thus without this assumption this message could possible be read infinitely. Though this situation can be easily avoided by the processor (actually, simulating some kind of a scheduler via its local data structures), we believe these details can be left to the reader.

We now present the algorithm to be performed in processor A.

States:
king_search, king_battle, king_defeated, subject_relay, subject_waiting.

Messages:
(**join**, *id, size, hop_length*)
(**accept**, *id*)
(**reject**, *id, size_B, id_B*).

Data structures:
Each processor A has a pool of unprocessed messages,
variables: *id, size, state, master, waiting_edge*.

procedure *surrender(id_B, e)*;
begin
state := subject_waiting;
master := *e*;
send (**accept**, *id.B*) on *e*
end.

procedure *response(id.B,size_B,hop_length, e)*;
begin

```
        if (size, id) > (size_B, id_B) then send (reject, id_B, size, id) on e
                        else if hop_length = 1 then surrender(id_B, e)
                                        else begin
                                                send (accept, id_B) on e;
                                                if state= king_... then state:= king_defeated;
                                end;

end.
```

Program for processor A

```
begin
  Initialization:
      size := 1;
        on k+1 edges send the message (join, id, size, 1);
        state := king_search;
        master := nil;

while the pool contains a message m from edge e
do begin
  case state of
  king_...:
        m = (accept,id) and (state = king_search or (state = king_battle and e = waiting_edge)):
                size := size + 1;
                if size > n/2 then send a leader message to all processors {you are the leader}
                else send (join , id, size, 1) on a new edge.

        m = (join, id_B, size_B, hop_length):
            {Invariant:id ≠ id_B}
              response(id_B,size_B,hop_length,e);

        m = (reject, id, size_B, id_B):
              if (size, id) < (size_B, id_B) then state :=  king_defeated
                        else if state = king_search
                        then begin
                                send (join, id, size, 1) on e;
                                waiting_edge := e;
                                state := king_battle
                        end
                        else if state = king_battle
                        then begin
```

```
                    return the message to the pool of
                            messages;
                    end;

  subject_relay:
            m = (join, id_B, size_B, 1):
                    send (join, id_B, size_B, 2) to master;
                    state := subject_waiting;
                    waiting_edge := e;

            m = (join, id_B, size_B, 2):
                  response(id_B,size_B,hop_length,e);

  subject_waiting.
            m = (accept, id_B) or (reject,id_B, size_C, id_C):
                    send m  on waiting_edge;
                    state := subject_relay;
                    if message_type = accept then master := waiting_edge;

            m = (join, id_B, size_B, 2):
              response(id_B,size_B,*,e);
end.
```

3. PROOF OF CORRECTNESS

In proving the correctness of the protocol, we consider a given execution of the algorithm. The following facts follow immediately from the protocol and our assumptions about the model:

(0) If a node B has a master then both B and its master are non-faulty.

(1) When a king becomes a leader, it sends a **leader** message to all processors, who subsequently terminate the algorithm.

(2) The algorithm is message-driven, thus, after the sending of the first messages, the operations taken by A between fetching one message and fetching the next one can be regarded as one atomic operation. Moreover, as explained above, each message is eventually read by the processor.

(3) The result of such an atomic operation might be *suspending* of this message, which results in returning the message to the pool. (Note: in some of the cases where we suspend messages we could actually destroy them, but we decided to leave it this way in order to simplify matters.)

(4) A subject A in state subject_waiting suspends the message (**join**,*id_B,size_B,1*). The reason for this is that A has already forwarded a message to its master, so it waits for a response (which might be the change of its master's status, or even the change of its master) before it transmits more messages.

(5) A king in state king_battle suspends an **accept** message (unless it belongs to its current war), and a **reject** message from a processor with higher status (which might cause a new war later).

(6) A king in state king_defeated suspends all its own messages (**accept** or **reject**; actually, they could be destroyed).

(7) A subject in state subject_relay and a king in state king_search do not suspend any message.

Lemma 2: Let A be a king which started a war with some node C as one of its tokens returned with a **reject** message from some node B.

(1) The war eventually terminates, with no additional messages sent from A to B.

(2) When the war terminates such that A does not receive a leader announcement, either A's status becomes subject_waiting, or this token returns from B with an **accept** message, causing *size_A* to be incremented.

Proof:

[]

Lemma 3: Let A be a king at any time τ during the execution. Eventually either A receives a leader announcement message, or one of the tokens sent has returned.

Proof: Following fact (1) we assume that no **leader** message is sent during the execution. at time τ, consider for each king the last token that has been sent (and not yet returned) in state king_search to a non-faulty processor, and consider the set T of all such tokens. Each token in T carries the values *size* nad *id* of its sender, and these values are clearly totally ordered. Let $size\_A, id\_A) = \max_{V \in T}(size\_V, id\_V)$. If this token of A arrives at a node B, it will eventually be treated by B (see fact (2)). When this is done, B might be either in a king state or in a subject state.

Case 1: B is in a king state.

The token is returned to A (by facts (5),(6) and (7)).

Case 2: B is in a subject state.

Both B and its master are non-faulty (fact (0)).

Case 2.1: B is in state subject-waiting. Then B does not suspend the message (fact (7)) and, according to the protocol, it forwards the token to its master C. When the master treats the message, it might be either a king or a subject.

Case 2.1.1: C is in a subject state.

In either state subject_relay or subject_waiting, it returns the token to B, and the token is then forwarded by B (in state subject_waiting) to A.

Case 2.1.2: C is in a king state.

Either a **reject** or an **accept** message will be returned to B, and then to A.

Case 2.2: B is in state subject-waiting. B returns the token to the pool.

[]

Lemma 4: Every king eventually has size $> \frac{n}{2}$ or it ceases to be a king.

Proof: Assume, to the contrary, that node A remains a king forever with *size_A)* $\leq \frac{n}{2}$. Consider the time A reaches its final size s, by receiving a returning token with an **accept** message. If one of A's suspended tokens now starts a war, then, by Lemma 2, this war will eventually end, and either A 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 lead to nodes in its domain. Since A is always in its own domain, and since *size_A* $\leq \frac{n}{2}$, it follows that the number of these edges is at most $\frac{n}{2} - 1$. There are at most $t < \frac{n}{2}$ other edges over which A has sent tokens that have 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 such an edge. By Lemma 3 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 a **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 5: 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*) at any time after the time this message is processed by A.

Proof: After the time this message is received by A, no more nodes are annexed to A's domain. Some of its tokens may still mark in its neighbors that they belong to A's domain, but when they return they are killed before doing anything (and thus we do not consider this an annexing). Thus, *size_A* remains unchanged while *size_C* may only increase, and the lemma follows.

[]

Corollary 6: At least one node always remains a king.

Proof: Assume, to the contrary, that all nodes cease to be kings, and consider the final sizes. Let A be the node that *((size_A, id_A) > (size_B, id_B)* for any other node B. Let C be the node that originated the message that caused A to cease to be a king. By Lemma 5 *(size_C,id_C) > (size_A,id_A)*, contradicting the definition of A.

□

Corollary 7: At least one king has eventually its size > $\frac{n}{2}$.

Proof: Follows from Corollary 6 and Lemma 4.

□

Lemma 8 [H84,AG85] : For every $l \geq 2$, if there are $l-1$ kings whose final size is not smaller than that of king A, then $size(A) \leq \frac{n}{l}$.

Sketch of 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 5). Thus domains of equal sizes (even viewed at different times) are disjoint. The lemma follows.

□

Theorem 9: 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 7 some king must eventually have its size larger than $\frac{n}{2}$. By Lemma 8 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

Lemma 10: Let A be a king and s its final size. The total number of times a token of A was sent by A (with an **join** message) is bounded by $2 \cdot s + t$.

Proof: When A initiates the algorithm it sends $t+1$ messages. Every other **join** message it sends follows the reception of either an **accept** or a **reject** message. The number of **accept** messages it receives as a king is $s-1$. The number of **reject** messages it receives as a king is bounded by s by Lemma 2. The lemma follows.

Now we can employ a technique similar to [G, H84, AG85], using Lemma 8, and get:

Theorem 11: The number of messages used by the t-resilient algorithm is $O(n \log k + k\,t)$.

Proof: The number of messages used for the leader announcement is $n-1$. The total number of the other messages sent during an execution is bounded by four times the number of **join** messages sent by a king. (At most two messages are used to arrive at the neighbor's master and at most two to return from it.) Let s be a final size of a king that initiated the algorithm. By Lemma 10 the number of times this king has sent a token with an **join** message does not exceed $2 \cdot s + t$. Nodes that did not wake spontaneously, have never been kings. Thus by Lemma 8 the total number of messages is bounded by $n-1+4\,(2\,n\,(1+1/2+1/3+...+1/k)+k\,t) = O(n \log k + k\,t)$.

□

The following theorem implies that the t-resilient algorithm is optimal:

Theorem 12: The message complexity of election in complete networks containing at most t faulty processors is $\Omega(n \log k + k\,t)$.

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 $\Omega(k\,t)$ consider a node which initiated the algorithm. It must send at least $t+1$ messages as it may be the only node to wake up, and the first t messages may have been sent to faulty nodes. Assume now that there is actually 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 A RESULT FOR ANOTHER MODEL

The result presented in Sections 3-5 can be extended for the case where every node knows its neighbors' identities. In fact, it is easily verified that

Theorem 13: The message complexity of election in complete networks containing at most t faulty processors where all identities are known to all nodes is $\Theta(k\,t)$.

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

Remark

After the preliminary version of this paper has appeared [KWZ86], Abu-Amara has independently developed an algorithm similar to ours [A87].

REFERENCES

[A87] Abu Amara, H. H., "Fault Tolerant Distributed Algorithm for Election in Complete Networks", *2nd International Workshop on Distributed Algorithms,* Technical Report RUU-CS-87-10, Dept. of CS, University of Utrecht, The Netherlands, July 1987.

[AG84] Afek, Y., and Gafni, E., Simple and Efficient Distributed Algorithms for Election in Complete Networks, *22-th Annual Allerton Conference on Communication, Control, and Computing,* Allerton House, Monticello, Illinois, October 1984, pp. 689-698.

[AG85] Afek, Y., and Gafni, E., Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks, Proceedings of the *4-rd ACM Symposium on Principles Of Distributed Computing,* Minaki, Canada, August 1985, pp. 186-

195.

[BKYZ] Bar-Yehuda, R., Kutten, S. Wolfstahl, Y., Zaks, S., "Making Distributed Algorithms Fault Resilient", Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 87), Lecture Notes in Computer Science 247, Springer- Verlag, February 1987, pp. 432-445.

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

[FLM85] Fisher, M.J., Lynch, N.A., and Merritt, M., Easy Impossibility Proofs for Distributed Consensus Problems, Proceedings of the *4-rd 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, pp. 373-382.

[L77] Le-Lann, G., Distributed Systems- towards a Formal Approach, *Information Processing 77 (ed. B. Gilchrist),* pp. 155-160, North-Holand 1977.

[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 Transactions on Programming Languages and Systems,* 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.

[KMZ83] Korach, E., Moran, S, and Zaks, S, Tight Lower and Upper Bounds For Some Distributed Algorithms for a Complete Network of Processors, *TR #298,* Department of Computer Science, Technion, Haifa, Israel, November 1983, and Proceedings of the *3-rd ACM Symposium on Principles Of Distributed Computing,* 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, *17-th IEEE Annual Electronics and Aerospace Conference,* Washington D.C. September 1984, pp. 101-105.

[KWZ86] Kutten, S., and Wolfstahl, Y., and Zaks, S., Optimal Distributed *t*-resilient Election in Complete Networks, IBM research Report RC 12177, September 1986.

[MW] Moran, S., and Wolfstahl, Y., An Extended Impossibility Result for Asynchronous Complete Networks, *Information Processing Letters* 26 (1987/88), pp. 145-151.

[SG86] Shrira, L., and Goldreich, O., Electing a Leader in the Presence of Faults: a Ring as a Special Case, *Acta Informatica,* 1986.