Optimal Distributed t-Resilient Election in Complete Networks

ALON ITAI, SHAY KUTTEN, YARON WOLFSTAHL, AND SHMUEL ZAKS

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 of which might be faulty, we present an algorithm that uses at most $O(n \log k + n + kt)$ 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 nonresilient one. The t-resilient algorithm is a systematic modification of an existing algorithm for a fault-free network.

Index Terms—Complete networks, distributed algorithms, fault-tolerance, leader election, message complexity.

I. Introduction

THE problem of *leader election* in asynchronous distributed systems has been widely studied (e.g., [8], [15]). 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 measures 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 line, 2) receiving a message from a pool of unserviced messages that arrived over lines, 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., [10]). Each message may contain at most $O(\log Max_id)$ bits, where Max_id is the highest possible 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 communication line, 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 identities of communication lines. 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 acknowledgments; i.e., a node sends a message on a line only after receiving an acknowledgment for the previous message sent on this line.

Consider the possibility that some nodes in the network may be faulty. We assume that the only type of faults is that in which a faulty node stops sending messages (these failures are known as *fail-stop* or *crash* failures; see [7]). In our model we also assume that all faults have occurred prior to the execution of the election algorithm (see also [4]). For the general case where nodes can fail *during* the execution of an algorithm, no deterministic election protocol exists [7], [16]. Other types of failures are also hard or impossible to cope with [5], [6]. (Fortunately, reliable hardware makes failures of the most general type quite rare [9].) Thus, additional assumptions are needed. These include, for example, knowledge about synchrony in the network [9], its topology [13], [17], or its size [17].

An $\lceil n/2 \rceil$-1-resilient consensus algorithm for a complete network is presented in [7]. $O(n^2)$ 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 $O(n^2)$

Manuscript received May 25, 1988; revised November 14, 1989. Recommended by F. B. Bastani. Part of this work was performed while S. Kutten and S. Zaks were visiting the IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

The authors are with the Department of Computer Science, Technion—Israel Institute of Technology, Haifa 32000, Israel.

IEEE Log Number 8933741.

0098-5589/90/0400-0415$01.00 © 1990 IEEE

for this case). An $O(n \log n)$ upper bound for 1-resilient election in a ring (where neighbor identities are known and only one edge may fail) is given in [17]. More general topologies were considered in [18].

We modify the election algorithm of [11], obtaining a t-resilient election algorithm (for any $t < n/2$). The resulting algorithm uses at most $O(n \log k + n + kt)$ 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 [7] and [13]). Note that when t is $O(n(\log k)/k)$ the message complexity is $O(n \log k)$, as in election algorithms for reliable networks [12]. On the other hand, for $n(\log k)/k = o(t)$ 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.

II. Description of the Algorithm

A. General Remarks

Leader election algorithms are usually viewed as each processor starting the algorithm by being its own king; the algorithm advances by processors surrendering to one another, and agreeing on a unique leader. Each processor knows, at any given time, the edge leading to its current master; in other words, it might belong to several "kingdoms" during the execution of the algorithm. Each king contains certain information about its kingdom; this information contains, for example, the size of the kingdom.

These election algorithms can be thought of as *token algorithms*. An originator of a message sends a token, that is a message carrying the originator's identity, and it traverses the network, trying to increase its orginator's kingdom. A processor receiving a token of another processor can modify some information (in either its local variables or in the message itself) and send it to other processors, but it does not change the identity of the originator of that message. 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 ensure that at least one of them will be processed.

B. Humblet's Algorithm

Our algorithm elects a leader in a complete network with n nodes, at most t of which may be faulty ($t < n/2$). It is a modification of the following algorithm of [11], which elects a leader in a reliable complete network (this algorithm is similar to the one in [2]).

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 becomes a subject, 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. The value of *size* may only increase. The size and identity of node A are denoted by $size_A$ and id_A, respectively. At different times 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). As explained above, the algorithm is described as if each king owns one *token*, which is a process representing it, and carrying its size, its identity and an additional message, which is either a **join** message (originated by the node that owns the token), an **accept** message or a **reject** message[1] (both originated by a node that was 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. When the token (**join**, id_A, $size_A$) arrives at C, it compares ($size_A$, id_A) to ($size_C$, id_C).

If $(size_A, id_A) > (size_C, id_C)$—lexicographically, namely, either $size_A > size_C$, or ($size_A = size_C$ and $id_A > id_C$)—then C's status becomes defeated and the token returns to node B; B joins A's domain, the token returns to A with an **accept** message, and $size_A$ is incremented (by 1). Otherwise—that is, $(size_A, id_A) < (size_C, id_C)$—the token is returned to B with a **reject** message (so that B can continue its algorithm) and is then destroyed (note that since the id's are distinct, there cannot be an equality).

A token that returns safely repeats the process of attempting to annex a new neighbor. The algorithm terminates when one processor A notices that $size_A = n$ (actually, $\lceil (n + 1)/2 \rceil$ suffices).

C. The t-Resilient Algorithm

As mentioned above, our algorithm is a modification of Humblet's. It differs from Humblet's original algorithm in the following respects.

1) Each processor owns $t + 1$ tokens instead of one; this ensures that at least one token arrives at a nonfaulty node.

2) All tokens return to their originators, carrying either an **accept** or a **reject** message.

3) Suppose a token was sent from A to B and from there to B's master C and upon examining the token C sees that its own status was larger than A's. It returns the token with a **reject** message. However, by the time the token returns to A, A's status may have increased and it may even become higher than C's. A therefore enters into a *war* with C: it resends the token (with A's new status) and does not read any of its other tokens. (These tokens will be *suspended*.) Another consequence is that A's status cannot change until the token returns from C for the second time. If the token carries an **accept** message then A increases its size and reads its suspended tokens. Otherwise the token carries a **reject** message, indicating that C also changed its size and its status is greater than A's fro-

[1] In Humbet's algorithm the reject messages are not returned to the originator.

zen status. Thus A becomes defeated. (It may destroy all of its tokens.)

We now present our algorithm. Each processor may be either a leader, or 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 are depicted in Fig. 1.

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 algorithm is *message-driven*, in the sense that each processor, after sending its first $t + 1$ messages, acts in steps, each consisting of reading a message from its pool, performing some computation, and either sending a message or returning the message to the pool. As long as no leader announcement is made, the processor continues to react to messages. The model assumes that a message that is sent to a processor eventually arrives at the pool. To simplify the exposition we assume the following.

Eventuality Property: Each message in the pool is eventually read.

This property can be easily achieved by letting the processor receive the messages in a round-robin fashion (without this property a message that is returned to the pool could possibly be read infinitely often, blocking all other awaiting messages, and resulting in a deadlock situation).

Processor A uses the following:

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

***Data structures*:**
A pool of unprocessed messages,
Variables: *id*, *size*, *state*, *edge_to_master*, *waiting_edge*.

***Messages*:**
(**join**, *id*, *size*, *hop_length*) (*hop_length* is either 1 or 2)
(**accept**, *id*)
(**reject**, *id*, *size_B*, *id_B*)
(**leader**, *id*).

Following is the algorithm to be performed by processor A.

```
state := king_search;
edge_to_master := nil;
size := 0;
if the node woke up spontaneously
then begin
        size := 1;
        send (join, id, size, 1) on t + 1 edges
end;
while the pool contains a message m (from edge e) do
begin
    remove m from the pool;
    if m = (leader, id) then stop
    else act as follows, according to the type of the message and
    your state (the actions—e.g., <1>—are detailed below):
```

	king_search	king_battle	king_defeated	subject_relay	subject_waiting
(**join**, *id_B*, *size_B*, 1)	<1>	<1>	<1>	<2>	<8>
(**join**, *id_B*, *size_B*, 2)	<1>	<1>	<1>	<1>	<1>
(**accept**, *id_B*)	<3>	if from *waiting_edge* then <3> else <8>	<9>	<9>	<4>
(**reject**, *id*, *size_B*, *id_B*)	<6>	if from *waiting_edge* then <7> else <8>	<9>	<9>	<5>

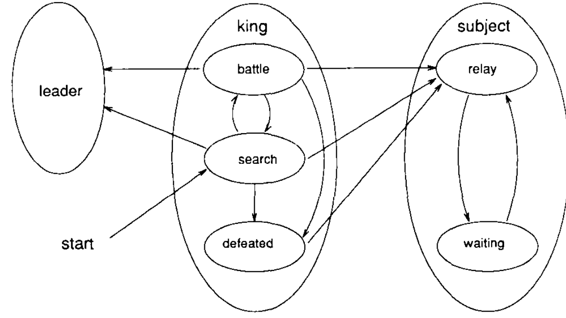

Fig. 1. The states of a processor.

```
<1> if (size, id) > (size_B, id_B) {Invariant: id ≠ id_B}
    then send (reject, id_B, size, id) on e
    else begin
        send (accept, id_B) on e;
        if hop_length = 1
        then begin
            state := subject_relay;
            edge_to_master := e
            end
        else if state = king_. . . then state := king_defeated
        end;
<2> send (join, id_B, size_B, 2) to master;
    state := subject_waiting;
    waiting_edge := e;
<3> size := size + 1;
    if size > n/2
    then begin
        state := leader;
        send (leader, id) to all processors {you are the leader}
        end
    else begin
            send (join, id, size, 1) on a new edge;
            state := king_search
        end;
<4> state := subject_relay;
    if id ≠ id_B
    then begin
        send m on waiting_edge;
        edge_to_master := waiting_edge
        end;
<5> state := subject_relay;
    if id ≠ id_B
    then send m on waiting_edge;
<6> if (size, id) < (size_B, id_B)
    then state := king_defeated
    else begin
            send (join, id, size, 1) on e;
            waiting_edge := e;
            state := king_battle
        end;
<7> state := king_defeated;
<8> Return m to the pool {the message is suspended};
<9> no action {the message is destroyed};
end.
```

III. Proof of Correctness

In this section we prove the correctness of the protocol. We first show that at least one node remains a king (Lemma 1). We then show (Lemma 2) that, as long as no

leader is elected, there is no deadlock in the network, that the algorithm terminates and that exactly one node remains as a leader (Lemma 3, Lemma 4, and Theorem 5).

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.

Fact (1): If a node B has a master then both B and its master are nonfaulty.

Fact (2): When a king becomes a leader, it sends a **leader** message to all processors, who subsequently terminate the algorithm.

Fact (3): A token can be destroyed only be its originator. A king in state king_defeated and a subject destroy all of their own messages (**accept** or **reject**).

Fact (4): As explained above, a step during the (message-driven) execution might result in *suspending* a message, i.e., returning it to the pool. The only possible suspensions are as follows:

a) 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 entail a change of its master's status, or even the change of its master) before it transmits any messages.

b) A king in state king_battle suspends an **accept** or a **reject** message (unless it belongs to its current war). **accept** messages are suspended in order to make sure that the message from the *waiting_edge* is still valid, thus each war involves sending of at most one more message. **reject** messages are suspended in order to prevent wars within wars.

Note: A subject in state subject_relay and a king in state king_search do not suspend any messages. Also, a message of type (**join**, *, *, 2) is never suspended.

Fact (5): Only in state subject_waiting a processor may receive an **accept** or **reject** message that is addressed to another processor.

Lemma 1: At least one node always remains an undefeated king.

Proof: Assume, to the contrary, that each processor became either a subject or a defeated king (after which its size did not increase). Consider the set T of all pairs ($size$, id) sent in tokens to nonfaulty processors (for each processor consider a pair with the largest possible $size$). These pairs are clearly totally ordered. Let $(size_A, id_A) = \max_{V \in T}(size_V, id_V)$. In order for processor A to become either a subject or a defeated king, it must have seen a token of another processor B such that $(size_B, id_B) > (size_A, id_A)$, a contradiction. □

Lemma 2: Consider any time τ during the execution. Eventually, either all processors receive a leader announcement message, or one of the tokens is returned to its originator.

Proof: Following Fact (2) we assume that no **leader** message is sent during the execution. At time τ, consider for each king the last token it sent (and has not yet returned) in state king_search or king_battle to a nonfaulty processor, and consider the set T of all such tokens (by Lemma 1, T is not empty). Each token in T carries the values *size* and *id* of its sender, and these values are totally ordered. Let $(size_A, id_A) = \max_{V \in T}(size_V, id_V)$. This token of A arrives at a node B, and by the **eventuality property** it will eventually be processed by B. 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 Fact (4) above—a king never suspends a **join** message).

Case 2: B is in a subject state. Both B and its master are nonfaulty [Fact (1)].

Case 2.1: B is in state subject_relay. Then B does not suspend the message [Fact (4)] and, according to the protocol, it forwards the message (**join**, *, *, 2) to its master, and, by Fact (4) and the **eventuality property**, this message is eventually returned to B and then to A.

Case 2.2: B is in state subject_waiting. B returns the token to the pool. Moreover, B is in this state since it forwarded a message $m' =$ (**join**, id_A', $size_A'$, 2) to its master, and, by Fact (4) and the **eventuality property**, m' is eventually returned to B and then to A'. □

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

Proof: When a **leader** announcement message is sent, its originator has $size > n/2$, and all other processors will receive this message and cease to be kings (if they are still kings). If no **leader** announcement is sent, then, by Lemma 2, at least one token will be returned from a processor C (via B) to its originator A. If A is a defeated king, then the token is destroyed. Applying this argument again, and using Lemma 1, we get that at least one token will be returned from a processor C (via B) to its originator A, which is a nondefeated king.

Now, consider all kings to which tokens are returned while being nondefeated kings. If a token is returned not on the *waiting_edge* of a king in state king_battle, then it is suspended, and we can apply the argument again. Therefore, eventually a token will return to its originator, that is a king in state king_search, or a king in state king_battle (and the token is returned via the waiting edge).

We proceed by induction on the number i of such kings in state king_search to show that the size of some king increases (actually, we use induction on the total number of nondefeated kings, and within it we use induction on the number of them in state king_search).

The case $i = 0$: if the token carries an **accept** message, then the processor will enter a king_search state and its size will be increased and we are done, and if it carries a **reject** message, then the processor will enter a king_defeated state, and we can apply the previous argument again, with a smaller number of nondefeated processors.

The induction step follows from the fact that if a token returns to a processor in state king_search (that is, $i > 0$), then the processor either becomes defeated (in which

case we apply the previous argument), or it enters a king_battle state (in which case we use the induction hypothesis), or its size is increased (in which case we are done). □

Lemma 4 [3], [11]: For every $l \geq 1$, there are at most n/l kings that ever reach size l.

Proof: If a node B of the domain of C joins the domain of A, then C ceases to be a king and from that time $(size_C, id_C) < (size_A, id_A)$. Thus domains of equal size (even viewed at different times) are disjoint. The lemma follows. □

Theorem 5: During any execution of the algorithm, exactly one leader is elected.

Proof: According to Lemma 1 at least one node will remain as a candidate for leadership, and Lemmas 3 and 4 assure the termination of the algorithm and the uniqueness of the leader. □

IV. Complexity Analysis

We now analyze the message complexity of the algorithm. Recall that throughout the discussion n denotes the number of processors in the network, t is an upper bound for the number of faulty processors, and k is an upper bound for the number of initiators. Lemma 4, that was used above in proving the correctness of the protocol, implies the following.

Theorem 6: The number of messages used by the t-resilient algorithm is $O(n \log k + n + kt)$.

Proof: The number of messages used for the leader announcement is $n - 1$. The total number of tokens sent in the beginning is $k(t+1)$. A token of processor A traverses at most eight steps before either it is destroyed or A's size increases (this is so because it might take up to four steps until the token returns and A enters a war situation, and it now takes at most four more messages for A to resolve this war). Since only the k processors that woke up spontaneously originate tokens, they are the only ones whose *size* is positive. Following Lemma 4, the total number of messages is bounded by $n - 1 + k(t+1) + 8n(1 + 1/2 + 1/3 + \cdots + 1/k) = O(n \log k + n + kt)$. □

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

Theorem 7: The message complexity of election in complete networks containing at most t faulty processors is $\Omega(n \log k + n + kt)$.

Proof: The term $\Omega(n \log k + n)$ follows from the lower bound of $\Omega(n \log k + n)$ messages for the problem of election in complete reliable networks [12], and from the fact that the number of nonfaulty processors $n - t$ is larger than $n/2$. For the lower bound of $\Omega(kt)$ 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. □

Note: The result presented above can be extended for the case where every node knows its neighbors' identities. In fact, Theorem 8 is easily verified.

Theorem 8: The message complexity of election in complete networks containing at most t faulty processors where all identities are known to all nodes is $\Theta(kt)$.

In the algorithm achieving the upper bound of $O(kt)$, 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 had appeared [14], Abu-Amara independently developed an algorithm similar to ours [1]. However, his algorithm deals with the case when there are n processors and f faulty communication lines, where $1 \leq f \leq \lfloor n/2 \rfloor - 3$, whereas we deal with the failure of t processors. Although this can be simulated by the failure of $t(n-1)$ edges, Abu-Amara's algorithm cannot be applied in this case, due to the restriction on the number of faulty edges. Moreover, his algorithm deals with faults that occur *during* the course of the algorithm, while we require all failures to occur *prior* to the execution. As explained in the Introduction, this is necessary; in other words, no version of Abu-Amara's algorithm will work in the presence of $n - 1$ or more faulty edges, since this will violate the impossibility result of [7].

Acknowledgment

We would like to thank the referees for many helpful suggestions.

References

[1] H. H. Abu-Amara, "Fault tolerance distributed algorithm for election in complete networks," *IEEE Trans. Comput.*, vol. 37, no. 4, pp. 449–453, 1988.

[2] Y. Afek and E. Gafni, "Simple and efficient distributed algorithms for election in complete networks," in *Proc. 22nd Annu. Allerton Conf. Communication, Control, and Computing*, Allerton House, Monticello, IL, Oct. 1984, pp. 689–698.

[3] —, "Time and message bounds for election in synchronous and asynchronous complete networks," in *Proc. 4th ACM Symp. Principles of Distributed Computing*, Minaki, Canada, Aug. 1985, pp. 186–195.

[4] R. Bar-Yehuda, S. Kutten, Y. Wolfstahl, and S. Zaks, "Making distributed algorithms fault resilient," in *Proc. 4th Annu. Symp. Theoretical Aspects of Computer Science (STACS 87) Lecture Notes in Computer Science*, vol. 247. New York: Springer-Verlag, Feb. 1987, pp. 432–445.

[5] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," Rep. YALE/DCS/RR-273, June 1983.

[6] M. J. Fischer, N. A. Lynch, and M. Merritt, "Easy impossibility proofs for distributed consensus problems," in *Proc. 4th ACM Symp. Principles of Distributed Computing*, Minaki, Canada, Aug. 1985, pp. 59–70.

[7] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 373–382, Apr. 1985.

[8] R. G. Gallager, "Finding a leader in a network with $O(|E|) + O(n \log n)$ messages," Lab. Inform. Decision Syst., M.I.T., Internal Memo, undated.

[9] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Trans. Comput.*, vol. C-31, no. 1, 1982.

[10] R. G. Gallager, P. M. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, Jan. 1983.

[11] P. M. Humblet, "Selecting a leader in a clique in $O(n \log n)$ messages," Lab. Inform. Decision Syst., M.I.T., Internal Memo, Feb. 1984.

[12] E. Korach, S. Moran, and S. Zaks, "Tight lower and upper bounds for some distributed algorithms for a complete network of processors," Dep. Comput. Sci., Technion, Haifa, Israel, Tech. Rep. 298, Nov. 1983; also *Proc. 3rd ACM Symp. Principles of Distributed Computing*, Vancouver, B.C., Canada, Aug. 1984, pp. 199-207.

[13] S. Kutten and Y. Wolfstahl, "Finding a leader in a distributed system where elements may fail," in *Proc. 17th IEEE Annu. Electronics and Aerospace Conf.*, Washington, DC, Sept. 1984, pp. 101-105.

[14] S. Kutten, Y. Wolfstahl, and S. Zaks, "Optimal distributed t-resilient election in complete networks," IBM, Res. Rep. RC 12177, Sept. 1986.

[15] G. Le-Lann, "Distributed systems—Towards a formal approach," in *Information Processing 77*, B. Gilchrist, Ed. Amsterdam, The Netherlands: North-Holland, 1977, pp. 155-160.

[16] S. Moran and Y. Wolfstahl, "An extended impossibility result for asynchronous complete networks," *Inform. Processing Lett.*, vol. 26, pp. 145-151, 1987/1988.

[17] L. Shrira and O. Goldreich, "Electing a leader in a ring with link failures," *Acta Inform.*, vol. 24, pp. 79-91, 1987.

[18] A. Itai and M. Rodeh, "The multi-tree approach to reliability in distributed networks," *Inform. Computat.*, vol. 79, no. 1, pp. 43-59, Oct. 1988; also preliminary version in *Proc. 25th Symp. Foundations of Computer Science*, Oct. 1984.

Alon Itai, photograph and biography not available at the time of publication.

Shay Kutten, photograph and biography not available at the time of publication.

Yaron Wolfstahl, photograph and biography not available at the time of publication.

Shmuel Zaks, photograph and biography not available at the time of publication.