Optimal Elections in Faulty Loop Networks and Applications*†

Bernard Mans‡ and Nicola Santoro§

Abstract

Loop networks (or Hamiltonian circulant graphs) are a popular class of fault-tolerant network topologies which include rings and complete graphs. For this class, the fundamental problem of Leader Election has been extensively studied assuming either a fault-free system or an upper-bound on the number of link failures.

We consider loop networks where an arbitrary number of links has failed and a processor can only detect the status of its incident links.

We show that a Leader Election protocol in a faulty loop network requires only $O(n \log n)$ messages in the worst-case, where n is the number of processors. Moreover, we show that this is optimal. The proposed algorithm also detects network partitions. We also show that it provides an optimal solution for arbitrary non-faulty networks with sense of direction.

keywords Loop Networks, Leader Election, Fault-Tolerance, Interconnection Networks, Distributed Algorithms, Sense of Direction.

*This research supported in part by N.S.E.R.C, grant#A2415, and by Macquarie Research Grant "Structural Information in Networks".

†Some of the results of this work have been presented at the 24th IEEE Annual International Symposium on Fault-Tolerant Computing (**FTCS'94**), and at the 14th IEEE International Conference on Distributed Computing Systems (**ICDCS'94**).

‡Dept. of Computing, School of Mathematics, Physics, Computing and Electronics, Macquarie University, Sydney, NSW 2109, Australia. Fax: 61-2-9850 9574. Email: bmans@mpce.mq.edu.au

§School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6 Canada. Email: santoro@scs.carleton.ca

1 Introduction

1.1 Loop Networks

A common technique to improve reliability of ring networks is to introduce link redundancy; that is, to have each node connected to two or more additional nodes in the network. With alternate paths between nodes, the network can sustain several nodes and links failures. Several ring networks, suggested in [3, 8, 27, 34, 40] are based on this principles. The overall topological structure of these *redundant rings* is always highly regular; in particular, the set of ring edges (*regular*) and additional edges (*bypass*) form a **Loop Network** (since they have at least one hamiltonian cycle).

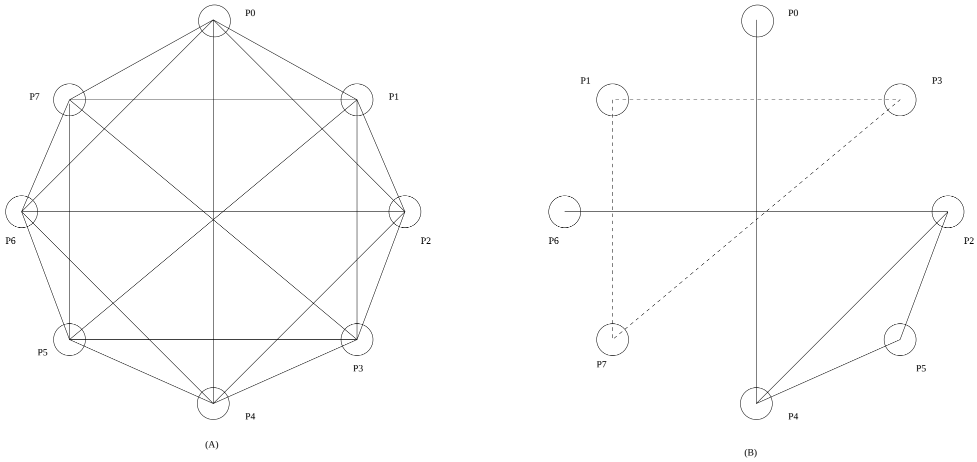


Figure 1: $\langle 2, 4\rangle$ Loop Network (a) with Faulty Links (b)

Loop Networks are particular cases of *Circulant Graph*. Because of an uncoordinated literature, numerous terms have been used to name this topology depending on the model; *Circulant Graph*, *Chordal Ring*, or *Distributed Loop Computer Networks* are the more common. A detailed survey of these topologies is presented in [5]. For sake of simplicity, we will use the term loop network in the remaining of this paper.

A loop network $\mathcal{C}_n\langle d_1, d_2, ..., d_k\rangle$ of size n and k-chord structure $\langle d_1, d_2, ..., d_k\rangle$ is a ring R_n of n processors $\{p_0, p_1, ..., p_{n-1}\}$, where each processor is also directly connected to the processors at distance d_i and $n - d_i$ by additional incident chords. The link connecting two nodes is labeled by the distance which separates these two nodes on the ring, i.e., following the order of the nodes on the ring: the node p_i is connected to the node $p_{i+d_j \bmod n}$ through its link labeled d_j (as shown in Figure 1(a)). In particular, if a link, between two processors p and q, is labeled by distance d at processor p, this link is labeled by $n - d$ at the other incident processor q, where n is the number of processors. Note that both *rings* and *complete graphs* are circulant graphs, denoted as $\mathcal{C}_n\langle\ \rangle$ and $\mathcal{C}_n\langle 2, 3, \cdots, \lfloor n/2\rfloor\rangle$, respectively. It is worth pointing out that some designs for *redundant meshes* and *redundant hypercubes* are also circulant graphs, [7].

The distinction between regular and bypass links is purely a functional one. Typically, the bypass links are used strictly for reconfiguration purposes when faults are detected; in the absence of faults, only regular links are used. Special classes of loop networks have been widely investigated to analyze their fault-tolerant properties [3, 7, 8, 9, 27, 33] and solutions have been proposed for reconfiguration after links and/or node failures [30, 39]. In some applications (e.g., distributed systems), all the links (or *chords*) of a circulant graph are always used to improve the performance of a computation.

1.2 Election

In distributed systems, one of the fundamental control problem is the **Leader Election** [29]. Informally, election is the problem of moving the system from an initial situation where the nodes are in the same computational state, to a final situation where exactly one node is in a distinguished computational state (called *leader*) and all others are in the same state (called *defeated*). The election process may be independently started by any subset of the processors. The election problem occurs, for instance, in token-passing when the token is lost or the owner has failed; in such a case, the remaining processors elect a leader to issue a new token. Several other problems encountered in distributed systems can be solved by election; for example: *crash recovery* (a new server should be found to continue the service when the previous server has crashed), *mutual exclusion* (where values for election can be defined as the last time the process entered the critical section), *group server* (where the choice of a server for an incoming request is made through an election among all the available servers managing a replicated resource), etc.

Following failures, the network might be partitioned into several disconnected components (as shown in Figure 1(b)). With respect to the election process, a component will be called *active* if at least one processor in that component independently starts the election process. A leader election protocol must determine a unique element in each active component; such distinguished elements can then determine any additional information (e.g., size of component, etc.) which is needed for the particular application. The nature of such applications is irrelevant to the election process.

It is assumed that every processor p_i has a distinct id_i chosen from some infinite totally ordered set ID; each processor is only aware of its own identity (in particular, it does not know the identities of its neighbours). The processors all perform the same distributed algorithm. A distributed algorithm (or protocol) is a program that contains three types of executable statements: local computations, message send and message receive statements. We assume that the messages on each arc arrive with no error, in a unbounded but finite delay and in a FIFO order. The complexity measure is the maximum number of messages sent during any possible execution.

1.3 Election in a Faulty Loop Network

The Leader Election problem in loop networks has been extensively studied assuming that there are no failures in the systems [4, 18, 23, 24, 32]. The problem becomes rather more difficult if there are failures in the system. In asynchronous systems, in particular, the election problem is unsolvable (i.e., no deterministic solution protocol exists) if failures are undetectable and can occur at any time; this impossibility result holds even if just one processor may fail (in a fail-stop mode) and follows from the result of [10].

The research has thus focused on studying the problem in more restricted environments:

- (**r1**) failures are detectable,

Graph	# Faults (r3) Links	Nodes	(r1) Detectability	(r2) Occurrence	Termination
Arbitrary Loop Network	0	1	No	Arbitrary	Impossible [10]
Complete (r5)	0	$\leq t$	No	Prior	Possible [1, 16, 28, 37]
Complete (r5)	$\leq k$	$\leq t$	No	Prior	Possible [31]
Complete (r5)	$< N/2$ per node	0	No	intermittent	Possible [2, 38]
Ring	≤ 1	0	No	Prior	Possible [14, 41, 42]
Arbitrary Loop Network	unbounded	unbounded	Yes	Prior	Possible (**this paper**)

Table 1: Impossibility versus Possibility Results (k and t are constants bounding the number of Fail-Stop Faults).

- (**r2**) failure occurs prior to the execution of the election protocol,
- (**r3**) the number of failures is bounded by some constant,
- (**r4**) failures are fail-stop,
- (**r5**) every processor is directly connected to every processor.

All the existing results for Election in faulty loop networks have been developed under assumptions (r2), (r3), (r4) and further assuming that the network is either a complete graph (r5) [1, 16, 28, 31, 37] or a ring [14, 41, 42] (see table 1). So far, without detectability, algorithms breaking free of the bounded number of failures assumption (r3) generate an expensive communication complexity ($O(n^2)$ messages of $O(n)$ bits, [19]).

In this paper, we consider the Election Problem in asynchronous arbitrary loop networks where an arbitrary number of links has failed and a processor can only detect the status of its incident links. That is, we make assumptions (r2) and (r4), and a relaxed version of assumption (r1). Thus, unlike all previous investigations, we do not restrict to complete graphs; we do not make any *a priori* restriction on the number of failures; we do however assume that a processor can detect the failure of its incidents links. Note that this assumption, the detectability assumption (r1), is required to cope with an unbounded number of faulty components (see table 1). We prove that, under these assumptions, a Leader Election protocol in a faulty loop network requires only $O(n \log n)$ messages in the worst-case, where n is the number of processors. Moreover, we show that this is optimal. In case the failures have partitioned the network, the algorithm will detect it and a distinctive element will be determined in each active component; depending on the application, these distinctive elements can thus take the appropriate actions.

Both processors and links may fail. In the following, we will assume that if a processor fails all its incident links fail. Thus, without any loss of generality, we can just consider link failures. We emphasize the fact that both regular and bypass links can fail (as shown in Figure 1(b)). A processor can only detect the failure of its incident links. Knowledge that a link is faulty can be either *off line* or *on line*. In the off line case, the hardware subsystem provides directly such a knowledge to the processors; thus, this information is *a priori* respect to the execution of the protocol. In the on line case, this knowledge can only be acquired upon an attempt to transmit on a link; if the link is operational, the message will be transmitted, otherwise an error signal will be issued by the system (see Figure 2).

From a computational point of view, the on line case is more difficult than the off line one. In particular, to transform it into *a priori* knowledge case (e.g., by a pre-processing phase where each active processor tests its incident links) would cost an additional m messages where $m = \Omega(e)$ is the number of non-faulty links. Thus, our $O(n \log n)$ solution, for the case where faults are only detected upon transmission attempt, is all the more important since fault-detection is performed only on these links which are used by the computation. Furthermore, this solution can obviously be applied with the same complexity to the case where there is *a priori* knowledge on the faulty links. Thus, in following, we will only concentrate on the more difficult case.

The algorithm presented here combines known techniques for election in non-faulty networks ([13, 17, 21]) and original routing paradigms based on structural information [12] in order to avoid the faulty components. The algorithm uses asynchronous promotion steps to merge rooted spanning trees.

2 Election Algorithm

We present an Election algorithm in loop network, where an arbitrary number of links have failed and where failure of a link is detectable only if an incident node attempts to transmit on it. The full algorithm is given in the Appendix (see also [26]). Any node can independently and spontaneously start the election process (we will model this by having such a node receive a WAKEUP message). If the network is not partitioned, the algorithm will detect it and will elect a leader. In case the failures have partitioned the network, a distinctive element will be determined in each active component and will detect that a partition has occurred; depending on the application, these distinctive elements can thus take the appropriate actions. We will now describe the algorithm as executed in each active component.

2.1 Description

In each active component, the algorithm builds a *Rooted Spanning Tree* or *Kingdom* by repeatedly combining smaller spanning trees; the final root of the spanning tree is the distinctive element of that component. In the following, we describe the algorithm as executed in one component.

The algorithm proceeds in phases and rounds. Initially, each node is a *king*, and does not know which of its links have crashed. At the end, all nodes are *citizen* except one which is still a *king*. During each intermediate phase of the algorithm, each *king* tries to expand its kingdom (a rooted directed tree) by attacking another kingdom. The attack is carried out by a particular node: the *warrior*.

Each kingdom is a tree with two distinguished nodes: the *king* and the *warrior*. Each king is assigned a *level*, initialized at zero. Each node p stores the identity $king_p$ and the level $level_p$ of its king, as well as the label of the outgoing chord to its king and to its warrior. If a node is attacked, it stores the label of the incoming chord from which the attack came. In the algorithm, each warrior p maintains a local view $List_p$ of all the others processors with the indication of which of them belong to the kingdom. An attack message is a request message defined by a request status $ReqStatus = (reqking, reqlevel, reqList)$ which contains such a local view $reqList$.

Informally, the attack is carried out only by a warrior; the warrior will select randomly an outgoing link which leads to another kingdom (one connected to a processor which does not belong to its kingdom). It then attempts to transmit a REQUEST message on that link. If the link is faulty, a *failure detection* signal will notify the warrior of such a situation and the appropriate action (see below) will be taken; otherwise, the REQUEST message will carry the attack to the other kingdom, as shown in Figure 2.

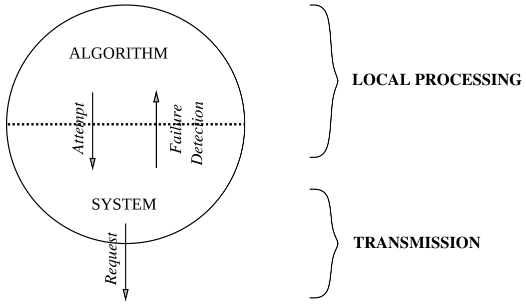


Figure 2: Local Failure Detection

The attacks by a kingdom follow a **Depth First Search** strategy. A state S_r for each chord is defined to specify if the chord is *unused* (initially), *branched* (is part of the spanning tree) or *failed* (determined after an attempt of transmission). For each branched chord a substate $SubS_r$ is introduced to specify if the chord is *closed* (is faulty or does not lead to another kingdom), or still *opened* (the incident node has not been completely explored and thus can lead to nodes which have not been reached yet). Initially, all non-faulty chords are *opened*. It is used to control the backtracking by closing a subtree whose visit has been completed. If a warrior j cannot reach any node outside the kingdom (this is locally determined by the state of its incident links and the local view $List_j$), then the state of *warrior*, together with $List_j$, is backtracked to its parent and the chord between them became *closed*. This strategy has the main advantage to limit the amount of backtracking after a combination compared to a *Breadth First Search* strategy. A state transition diagram of a chord is shown in Figure 3(a). Each node saves the label $\{W_{out}, W_{in}, K_{out}\}$ of the incident chord leading to $warrior_p$, the *warrior* attacking p, and $king_p$ respectively.

Define the $status_p$ of node p as $(level_p, king_p, List_p)$. Following a lexicographic total order, we say that $status_p > status_j$ iff:
- either (a) $level_p > level_j$
- or (b) $level_p = level_j$ and $king_p > king_j$.

Our algorithm obeys two main rules:

Promotion Rule. A warrior p can only successfully attack a kingdom with *status* less than its own. Let the attack by warrior p be successful. In case (a), each node in the kingdom which lost is informed of the identity of the new king $king_p$ and updates its level to $level_p$ (note that the value of $level_p$ is unchanged in the attacking kingdom). In case (b), each node in the attacked kingdom receives the identity $king_p$ of the new king and all nodes in both kingdoms increases their level by one (the *level* of a kingdom never decreases). After a successful attack by a warrior p to a warrior j, the warrior of the new kingdom is $warrior_j$. We say that a processor enters a new round when its level changes, (i.e., when its kingdom has been defeated or when its kingdom successfully attacked a kingdom of an identical level).

Asynchronous Rule (controls the number of messages during each phase): three different cases are theoretically possible when an attack from a warrior p reaches a node j in another kingdom:

1. $status_p < status_j$: the warrior is not strong enough to attack this kingdom and, thus, its attack fails: the message is killed and the attacking kingdom is just waiting to get attacked.

2. $status_p > status_j$: the attack from p must be forwarded to $warrior_j$. Any subsequent attack by other kingdoms, if not killed, is delayed until this attack is resolved at j (i.e., until j receives a new status).

   When forwarding an attack, if node i on the path to $warrior_j$ has a greater status (i.e., $status_i > status_p$), the request is killed. This situation occurs when the previously visited nodes have not yet been informed that they have become part of a greater kingdom (i.e., the level has increased).

   When the attack reaches warrior j, if it still has a lower status, then a surrender message is sent back to $warrior_p$ and each node on the path waits for the new status.

3. $status_p = status_j$: as proved later, this case (i.e., an attack within the same kingdom) cannot occur during the execution of the algorithm.

If warrior p receives a message of surrender, it broadcasts the new status to the absorbed kingdom or to both kingdoms, depending on the promotion rule. The new local view *List* is obtained by merging the two Lists. The initial local view is a list of bits $List[0..(n-1)]$. Since initially the processor sees only itself and is at chord distance 0, the list is initialized to 10^* (i.e., all bits are set to 0 except $List[0]$ which is set to 1).

Concurrency. The number of concurrent incoming attacks in a kingdom must be limited in order to guarantee a message complexity of $O(n)$ for each round. A substate $Substate_p$ for each node p is introduced to specify if the node is *WaitingForSurrender* (has forwarded an attack message), is *WaitingForStatus* (has forwarded a surrender message and is waiting for its new level), or is *Regular* (is ready to receive an attack). The state transition diagram of a processor is shown in Figure 3(b).

Some substates are introduced to deal with two specific situations which may occur due to the inherent concurrency of the model.

First of all, if a citizen j has forwarded an attack to $warrior_j$ a subsequent attack with a greater status will be delayed (wait at j), but not killed (asynchronous rule 2).

Secondly, an incoming attack can be received before knowing that the kingdom has already absorbed (or been absorbed by) another kingdom: the level may have increased.

In both cases, the *citizen* knows afterwards (when it receives the new status) if the forwarded attack was successful. At this time, if the status of the forwarded attack is smaller than the new received status, the attack will be killed; thus, the *citizen* can go

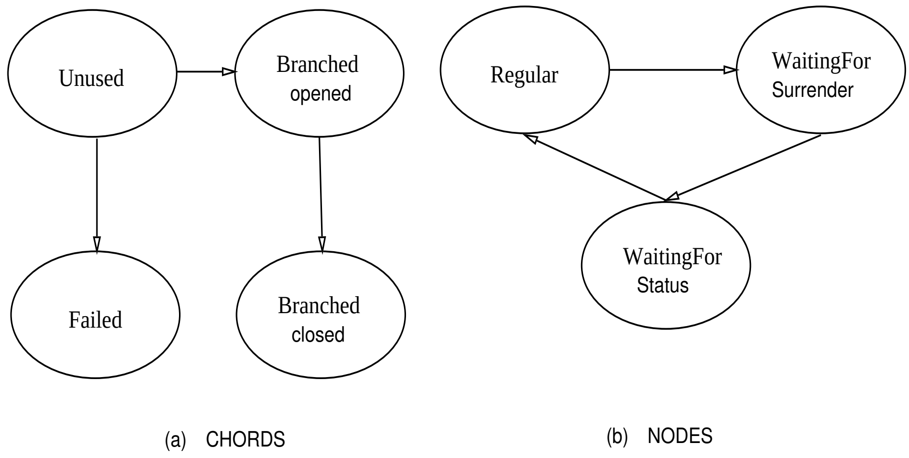


Figure 3: State Transition Diagrams

back to *regular* substate. Otherwise, the current attack status is still legal; thus, the inhibition waiting substate must be kept.

Progress. The problem occurs if a warrior q receives a surrender message from a warrior p when it is already engaged in a wait for status process from a warrior w (q has been attacked by w while attacking p). Consistently with the asynchronous rule, the warrior q has to wait for the new status of warrior w before it can send the new status to the warrior p. The extreme case occurs if - a more complicate scenario involving more nodes can be deduced - w is waiting for p (p has attacked w): a deadlock situation. As proved later in Theorem 2.1, the total lexicographic order on the *status* forbids the creation of such a waiting cycles.

Structural Information. The knowledge of the size of the network, the topology, a globally consistent assignment of labels (or, labelings) to interconnection nodes and communication links is used to reduce the communication cost. Since the loop network is a node-symmetric graph (all its nodes are similar to one another), each node can represent the other nodes by their relative distance along the cycle. This is actually available with the edge labeling and can be used to pass the knowledge of the processors (represented by their distances) that have been already reached: when node p_1 receives a message from node p_2 by the incident chord labeled d_1, it can unambiguously "decode" the information about other nodes contained in the message. Namely, if the message contains information about the node linked to p_2 by a chord d_2, then this information refers to the node at distance $(d_1+d_2) \bmod n$ from p_1 in the ring ordering. This fact will be used to determine whether an unused chord (i.e., on which no messages have been sent) is outgoing or not (that is connected to a different kingdom or not). This function combined with the local view of a processor provides the message with a consistent representation of the kingdom which can be passed from processor to processor. This decoding function corresponds to a circular bit shift by the length of the chord, denoted as `transpose` (the exact code of the function is given at the end of the algorithm).

Termination and Partitioning. The algorithm terminates when the kingdom includes all nodes in its connected non-faulty subgraph. The determination of this event may differ depending on whether the network is disconnected or not. Consider first the case of a partitioned network. Once all reachable nodes have become part of the kingdom, the king will become warrior (because of the bactracking inherent to the depth first search strategy) and all its incident chords will be *closed* (there is no outgoing link towards a node which does not belong to the kingdom). At this point, it will detect termination; from its local view, it will also determined the size of its kingdom and that a disconnection has occurred.

If the network is not disconnected, the termination detection can occur earlier: as soon as a warrior determines, by its local view, that the kingdom includes all the nodes in the network (the list is full, i.e., set to 1^*).

In both cases, the warrior (which is possibly the king) broadcasts along the tree the termination message. Since this message contains the view of the warrior upon termination, every node in the component can determine whether or not the graph is disconnected as well as which other nodes are in this component. In the case of a disconnection, depending on the application, the king can take the appropriate action.

An example of an attack is shown in Figure 5, where the kingdom K has a greater status than the kingdom K' (the corresponding loop network $\mathcal{C}_{16}\langle 3, 8\rangle$ is shown in Figure 4). The result of the successful attack is shown in Figure 6.

Messages Used:

- *(REQUEST,Status)*: it is an attack by a warrior, and is forwarded to its adversary. This message is also considered as the first *ATTEMPT* on the chord, and provides the failure detection if the chord is faulty,
- *(SURRENDER,Status)*: it is sent by a defeated warrior to inform the winner of its success,
- *(NEWSTATUS,Status)*: it is broadcast by the winner on the appropriate tree (depending on promotion rule),
- *(BRANCH)*: it is sent by a successful warrior on the chord connecting the two trees,
- *(BACKTRACK,Status)*: it is sent by the warrior to its parent when all its chords have been closed, that is when all the nodes reachable through this chord are part of the kingdom or are faulty,
- *(MOVEWARRIOR,Status)*: it is sent by the warrior to one of its opened chords after a backtracking,
- *(TERMINATION)*: it is broadcast by the sole remaining warrior of the connected component to terminate the execution of the algorithm.

Any number of processors can spontaneously start the execution of the algorithm; this is modeled by the reception of a WAKEUP message. The active components are those where at least one processor spontaneously start the algorithm (i.e., it receives a WAKEUP message).

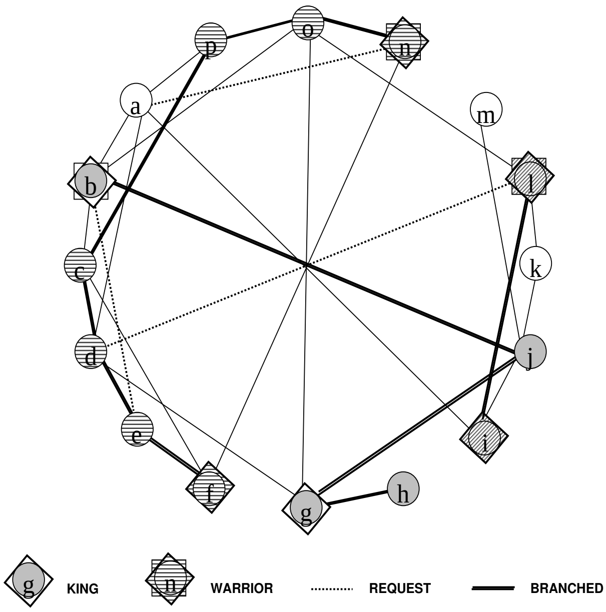


Figure 4: Kingdoms in $\mathcal{C}_{16}\langle 3, 8\rangle$

2.2 Correctness

The protocol is fully asynchronous, the messages received by each processor and the order in which each processor receives its messages depends on the initial input but is non-determinist. However the algorithm is event-driven with messages processed in first-in-first-out order, the order in which each processor processes its communication relies on tree structures and on the asynchronous and progress rules.

The correctness follows after establishing the *safety* (a warrior never attacks a node of its kingdom), the *progress* (eventually a tree spans all the nodes of a connected component), and the *appropriate termination* (there is exactly one elected node in a connected component of the network). In the following, numbers between parentheses refer to corresponding sections of the algorithm in the Appendix.

Lemma 2.1 *A request message is initiated by a warrior through an unused opened chord. The request message only traversed citizen nodes and branched chords leading to the warrior of the kingdom traversed.*

Proof The warrior sends the request (if the attempt is successful) through an unused opened arc (4, 5, 7, and procedure attempt at the end of the description of the algorithm). A citizen (or king) can send a request only upon receipt of a request (1) to forward it to its warrior through links labeled W_{out}, that is, a used chord of a citizen. □

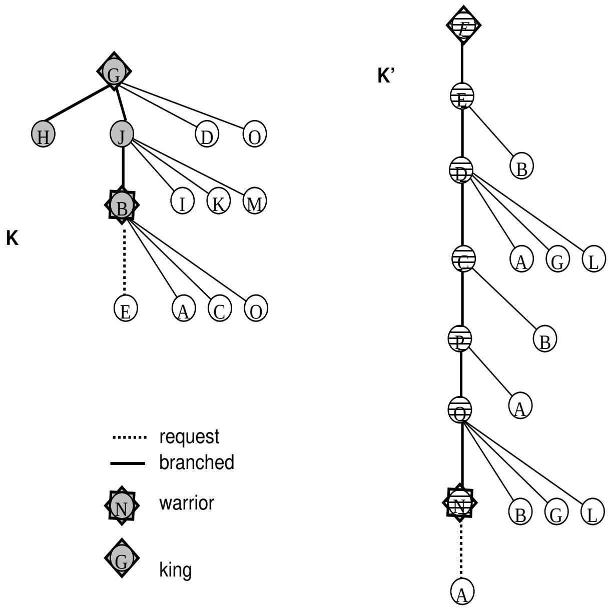


Figure 5: Two Kingdoms in $\mathcal{C}_{16}\langle 3, 8\rangle$

Corollary 2.1 *The status of a chord becomes used if a warrior has previously sent a request through it, or if the chord has been detected as faulty.*

Lemma 2.2 *The local view $List_p$ at a warrior p represents exactly the list of processors which belong to the kingdom of the warrior p.*

Proof By induction. Clearly, this is true at the initialization when the local view is set to 10^*. Assuming the local view $List_p$ at a warrior w is correct and complete before an attack, the warrior modifies its view either after a successful attack (while receiving a surrender message (8): the warrior becomes a citizen, combines the two views, and pass the warrior privilege of the new combined kingdom to the defeated warrior) or after being defeated (while receiving a newstatus message (7): it receives the view of the winning kingdom, and by combination obtains the complete view of the merged kingdom). In both cases, the new local view contains the exact list of processors of the new kingdom, which proves the induction. □

Lemma 2.3 (Safety.) *A warrior never attacks a node of its kingdom.*

Proof As shown in Lemma 2.2, an attack can only be done upon receipt of a new status which creates the new list of all the nodes which belong to the kingdom (7). All the chords linked to these nodes are closed, any remaining unused chord, even randomly chosen, leads to a processor of a different kingdom. Therefore, no cycle can be created in the kingdom. □

Several facts and properties can be observed to clarify the correctness.

Fact 2.1 *From (1) and the asynchronous rule, a waiting citizen, or king, does not process request messages.*

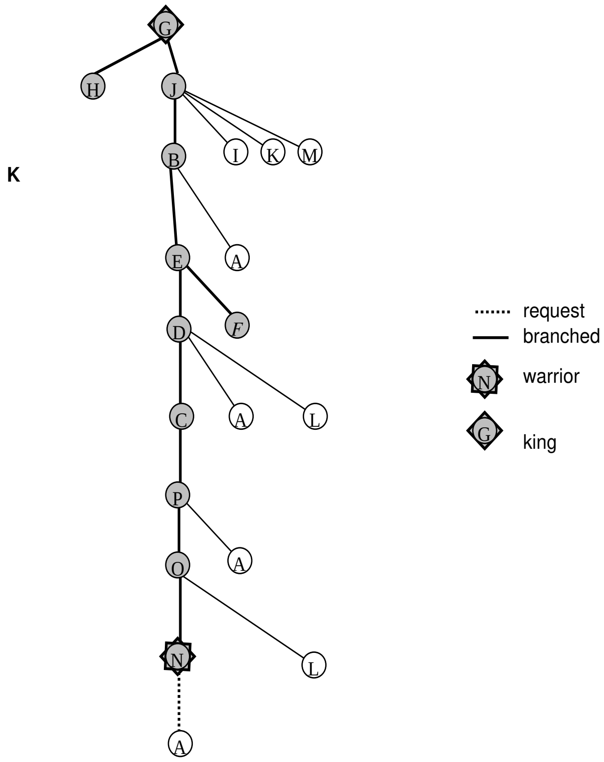


Figure 6: Result of Attack in $\mathcal{C}_{16}\langle 3, 8\rangle$

Fact 2.2 *Eventually each node in a kingdom receives the status of its kingdom. Indeed, at the end of any phase or after being defeated (8), the designated warrior broadcasts the new status along the traversed chords.*

Lemma 2.4 *No waiting cycle of requests may be created.*

Proof Immediate since sending a request does not change the regular state of the warrior (7). Therefore, all the requests which wait on a non regular node do not block the warrior which has initiated them. □

Theorem 2.1 (Progress.) *A deadlock may not be introduced by the waiting which arises when some nodes must wait until some condition holds.*

Proof The message sending is non-blocking. The only case for which a node is blocked waiting for an event is when a warrior waits for a new status message after sending a surrender (1). Similarly, such a surrender message can be deferred at the successful warrior node if it has surrendered to another warrior attack (8). Repeating this setting, a chain of waiting (on surrender) processors can occur. However, this chain cannot become a circular wait: a surrender message is initialized only on a successful attack, that is when the status of an attacking warrior j is strictly lexicographically larger than the status of a defending warrior p. The total ordering on the status defined by the promotion rule forbids such a waiting cycle of processors: $status_j < .. < status_p < .. < status_j$ contradicts the definition. □

Corollary 2.2 *Eventually, no node is in a waiting substate.*

Theorem 2.2 *A kingdom is a rooted directed tree.*

Proof By induction. Initially, each kingdom is a one node tree (0). The kingdom is defined by the subgraph composed by the chords marked K_{out} and their incident nodes, and is rooted by the king. It can also be defined by the subgraph composed by the chords marked W_{out} and their incident nodes: in this case the tree is rooted at the warrior.

Following a successful attack, the chord connecting the two trees (the absorbing and the absorbed ones) becomes part of the kingdom upon receipt of a NEWSTATUS message (7) initiated by the winner warrior and broadcast through the absorbed kingdom.

The outgoing chord to the king is stored in the K_{out} label. The king has a nil value for K_{out} (0). A node (citizen and/or king (3), warrior (7)) changes its label K_{out} only after receiving a new status message announcing the absorption by another kingdom; in this case K_{out} is set to the incoming arc from which such a message is received. This change of orientation guarantees that the tree is rooted at the new king. Note that a similar observation can be repeated for the tree rooted at the warrior. □

Lemma 2.5 (Appropriate Termination.) *The algorithm terminates with a forest of, at most, one rooted spanning tree for each connected components.*

Proof By the safety Lemma 2.3, the progress Theorem 2.1, and Theorem 2.2. In each connected component where at least one processor initiated the Election protocol, the algorithm builds a rooted spanning tree. □

The main Theorem is deduced:

Theorem 2.3 *The algorithm correctly elects a leader.*

Proof By Theorem 2.1, Theorem 2.2, and Lemma 2.5, the theorem holds. The Election protocol is independently started by any subset of processors electing a particular node in each active connected component (the king (10)). Each group of processors in a (partitioned or not) active component forms a consistent view (containing the exact list of reachable processors) with a single elected node: the king. Depending on the application, these distinctive elements can thus take the appropriate actions: e.g., promote themselves leader on a majority basis, wait for the recovery of the faulty components, simulate the non-faulty topology by embedding it into the active connected group, form a restricted (connected) working group,... □

2.3 Analysis

The measure of efficiency analyzed here is the communication complexity (the number and size of messages sent).

Lemma 2.6 *The number of rounds is at most* $\log k$ *for each kingdom, if* k *independent nodes start the algorithm.*

Proof By the promotion rule, based on a tournament, at most $n/2^i$ nodes enter phase i, in fact $k/2^i$ if k independent nodes start the algorithm. The maximum number of rounds is the maximum value of the level of the winning kingdom, i.e., $\log k$. □

Corollary 2.3 *The number of surrender messages sent by a warrior during a particular execution is at most* $\log k$*, if* k *independent nodes start the algorithm.*

Lemma 2.7 *For a given round and a given non-faulty chord* l *in a kingdom, at most two requests will be transmitted through the chord* l*.*

Proof For a given round and a given non-faulty chord l in a kingdom, a request passing through this chord will face several possible outcomes:

1. The request is successful with an identical level: it will cause the round to increase in both kingdom. Any forthcoming requests with this previous level will be discarded at the incident node.
2. The request is successful with a different (i.e., larger) level: the level value is updated only in the absorbed kingdom. By Lemma 2.3, only requests sent by a different kingdom may occur. Another request with the same level will behave as described in the case 1 limiting the number of such occurrences to two.
3. the request is unsuccessful: that is, the message has been killed further on the path to the warrior. This implies that the level has been increased by another attack, but the nodes incident on this chord does not know it yet. By the concurrency rule enforcing delay, only one other request can wait at the incident node and will be discarded when the newstatus arrives.

A similar argument can be used for a branched chord between two kingdoms. □

Corollary 2.4 *For a given round and a given non-faulty chord* l *in a kingdom, at most two surrender (resp. new status) messages will be transmitted through the chord* l*.*

More precisely,

Theorem 2.4 *The total number of messages used by the algorithm does not exceed*

$$6\,n \log k \;+\; 4(n-1)$$

Proof The number of messages of each kind is the following:

REQUEST : sent, at a given round, through at most $n-1$ non-faulty chords (see Lemma 2.7). Hence, the total number of such request messages sent during the whole execution is bounded by $2\,n \log k$.

SURRENDER : sent through a path in a kingdom only before a modification of its level. Hence, the total number of such messages sent during the whole execution is also bounded by $2\,n \log k$.

NEWSTATUS : broadcast in the kingdom only to increase its level. Hence, the total number of such messages sent during the whole execution is also bounded by $2\,n \log k$.

BRANCH : sent on each *branched* chord of the kingdom, i.e., at most $n-1$ messages.

BACKTRACK : sent on a *branched* chord of the kingdom if the subtree cannot reach further nodes. Hence, the total number of such messages is bounded by the size of the spanning tree, i.e., at most $n-1$.

MOVEWARRIOR : sent on each *opened-branched* chord of the kingdom if the node cannot reach further nodes. Hence, the total number of such messages is also bounded by the size of the spanning tree, i.e., at most $n-1$.

TERMINATION : at most $n-1$ messages.

□

Only seven different types of message exists. The status is composed of: the identity of the king which value is at most m, the *level* which takes at most $\log n$ values, and the *List* which is a n bits array. Therefore, the size of each message is at most $n + \log(7\ m\ \log n)$ bits.

Theorem 2.5 *The algorithm has an optimal worst-case message complexity.*

Proof Given a loop network C, let $F(C)$ denote the set of the possible combination of links failures in C; clearly the cardinality of $F(C)$ is $2^{|E|}$ where E is the set of chords of C. Given $f \in F(C)$ denote by $M(C, f)$ the number of messages required to solve the election problem in C when the failures described by f have occurred. Then, the worst case complexity $WC(C)$ to solve the election problem in C after an arbitrary number of link failures is

$$WC(C) = \max_{f \in F(C)} \{M(C, f)\} \geq M(R_n, \emptyset) = \Omega(n\ \log\ n)$$

where n is the number of processors, and R_n is the ring without bypass; the last equality follows from the lower bound by [6] on rings. □

2.4 Sensitivity to Absence of Failures

The algorithm we have presented uses $O(n \log n)$ messages in the worst case, regardless of the amount of faults in the system.

Consider now the case where no faults have occurred in the system and an Election is required. If all the nodes had *a priori* knowledge of this absence of failures, then they could execute an optimal Election protocol for non-faulty networks. In this case, depending on the chord structure, a lower complexity (in some cases, $O(n)$) can be achieved [4, 18, 23, 24, 32]. However, to achieve this complexity, it is required that the absence of failures is *a priori* known (more specifically, it is *common knowledge* [15]) to all processors.

Now we show how to achieve the same result without requiring this common-knowledge. First observe that the existing optimal algorithms for election in non-faulty loop networks use only a specific subset of the chords to transmit messages. The basic idea is quite simple. A processor "assumes" that its specific incident arcs are non-faulty. Based on this assumption, it starts the corresponding topology-dependent optimal election algorithm A. If a processor x detects a failure when attempting to transmit a message of protocol A, x will start the execution of the algorithm proposed in section 2. Thus, if there is no failures, algorithm A terminates using M_A messages; if there are failures, the overall cost of this strategy is $M_A + O(n \log n)$ which is $O(n \log n)$ since $M_A \leq O(n \log n)$.

The approach actually leads to a stronger result. To obtain the topology-dependent optimal bound M_A for the non-faulty case is sufficient that the chords used by A are fault-free.

3 Extensions and Applications

We will consider in this section the election problem in a different setting. In fact, we study arbitrary networks with sense of direction in absence of faults. We show how the previous results presented in this paper can be immediately used to prove the positive impact that the availability of "sense of direction" has on the message complexity of distributed problems in arbitrary fault-free networks.

3.1 Sense of Direction

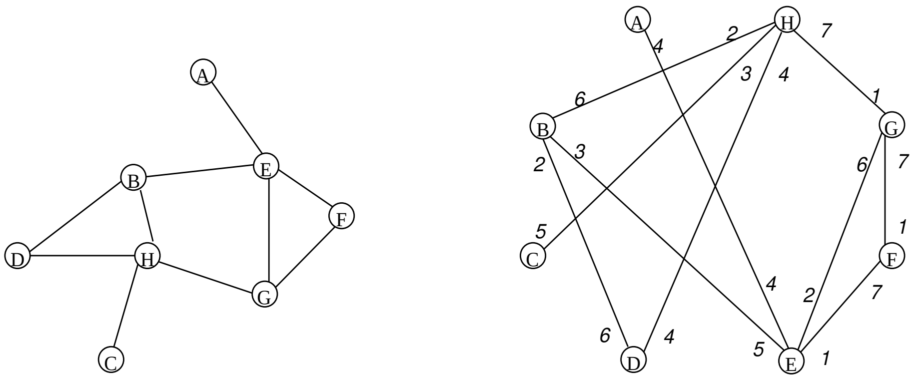
The **sense of direction** refers to the capability of a processor to distinguish between adjacent communication lines, according to some globally consistent scheme [12, 36]. For example, in a *ring* network this property is also usually referred to as orientation, which expresses the processor's ability to distinguish between "left" and "right", where "left" means the same to all processors. In oriented *tori* (i.e., with sense of direction), labelings "up" and "down" are added. The existence of an intuitive labeling based on the dimension provides a sense of direction for *hypercube*, [11]: each edge between two nodes is labeled on each node by the dimension of the bit of the identity in which they differ. Similarly, the natural labeling for loop networks discussed in the previous section is a sense of direction.

For these networks, the availability of sense of direction has been shown to have some impact on the message complexity of the Election problem.

In an *arbitrary network*, we define a globally consistent labeling on the links by extending in a natural way the existing definitions for particular topologies. Fix a cyclic ordering of the processors. The network has a distance sense of direction if at each processor each incident link is labeled according to the distance in the above cycle to the other node reached by this link. In particular, if the link between processors p and q is labeled by distance d at processor p, this link is labeled by $n - d$ at processor q, where n is the number of processors. An example of sense of direction for an arbitrary network is shown in Figure 7. Note that such a definition intrinsically requires the knowledge of the size n of the network, and it includes as special cases the definition of sense of direction for the topologies referred above: the oriented ring ("left" and "right" correspond to 1 and $n - 1$, respectively), the oriented complete networks (n set to the number of links plus one), and the oriented *loop network* or *circulant graph*. Furthermore, in *hypercubes*, this sense of direction is derivable in $O(N)$ messages from the traditional one [11].

3.2 Election in Fault-Free Arbitrary Networks

We now consider the impact of sense of direction on the message complexity of the Election problem.



(a) **(b)**

Figure 7: Arbitrary Network (a) with Sense of Direction (b)

It is obvious that every graph is a subset of the complete graph; that is, any arbitrary network is an "incomplete" complete graph. Less obvious is the fact that:

Every arbitrary network with sense of direction is an "incomplete" loop network.

That is, every arbitrary network is a loop network where some edges have been removed.

This simple observation have immediate important consequences. It implies that an arbitrary graph with sense of direction is just a faulty loop network (compare Figure 1 and Figure 7): the missing links correspond to the faulty ones. Moreover, in this setting, every processor already know which links are faulty (i.e., missing).

As a consequence, the algorithm described in Section 2 is also a solution to the election problem in fault-free arbitrary graphs with sense of direction [25].

By theorem 2.4, it follows that if there is sense of direction, a solution with $O(n \log n)$ messages exists for the Election problem. Since $\Omega(n \ \log n)$ is a lower bound on the message complexity for the election problem in bidirectional ring with sense of direction [6], it follows that $\Omega(n \ \log n)$ is also a lower bound on the general case. Thus, the bound is tight. In contrast, in arbitrary networks of n processors where the links have no globally consistent labeling (no sense of direction), $\Omega(e + n \log n)$ messages are required to elect a leader [35], and such a bound is achievable [13].

The importance of the result is that it shows the positive impact of sense of direction on the communication complexity of the Election problem in arbitrary network, confirming the existing results for specific topologies. An interesting consequence of our result follows when comparing it to those obtained assuming that each processor knows all the identities of its neighbours [20, 22]. Namely, it shows that it is possible to obtain the same reduction in message complexity requiring much less information (port labels instead of neighbour's name).

4 Concluding Remarks

In this paper, we have presented a $\Theta(n \log n)$ solution for the Election problem in loop networks where an arbitrary number of links have failed and a processor can only detect the status of its incident links. If the network is not partitioned, the algorithm will detect it and will elect a leader. In case the failures have partitioned the network, a distinctive element will be determined in each active component and will detect that a partition has occurred; depending on the application, these distinctive elements can thus take the appropriate actions. Moreover, the algorithm is worst-case optimal.

All previous results have been established only for complete graphs and have assumed an *a priori* bound on the number of failures. No efficient solution has been yet developed for arbitrary circulant graphs when failures are bounded but undetectable.

Our result is quite general. In fact, our algorithm can be easily modified to solve the Election problem with the same complexity for fault-free arbitrary networks with sense of direction.

References

[1] H.H. Abu-Amara. Fault-tolerant distributed algorithm for election in complete networks. *IEEE Transactions on Computers*, 37(4):449–453, April 1988.

[2] H.H. Abu-Amara and J. Lokre. Election in asynchronous complete networks with intermittent link failures. *IEEE Transactions on Computers*, 43(7):778–788, July 1994.

[3] B.W. Arden and H. Lee. Analysis of chordal ring. *IEEE Transactions on Computers*, C-30(4):291–295, April 1981.

[4] H. Attiya, J. van Leeuwen, N. Santoro, and S. Zaks. Efficient elections in chordal ring networks. *Algorithmica*, 4:437–446, 1989.

[5] J.-C. Bermond, F. Comellas, and D.F. Hsu. Distributed loop computer networks: a survey. *Journal of Parallel and Distributed Computing*, 24:2–10, 1995.

[6] H.L. Bodlaender. New lower bound techniques for distributed leader finding and other problems on rings of processors. *Theoretical Computer Science*, 81:237–256, 1991.

[7] J. Bruck, R. Cypher, and C.-T. Ho. Fault-tolerant meshes and hypercubes with minimal numbers of spares. *IEEE Transactions on Computers*, 42(9):1089–1104, September 1993.

[8] D.Z. Du, D.F. Hsu, and F.K. Hwang. Doubly link ring networks. *IEEE Transactions on Computers*, C-34(9):853–855, September 1985.

[9] S. Dutt and J.P. Hayes. Designing fault-tolerant systems using automorphisms. *Journal of Parallel and Distributed Computing*, 12:249–268, 1991.

[10] M.J. Fisher, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the A.C.M.*, 32(2):374–382, April 1985.

[11] P. Flocchini and B. Mans. Optimal elections in labeled hypercubes. *Journal of Parallel and Distributed Computing*, 33(1):76–83, 1996.

[12] P. Flocchini, B. Mans, and N. Santoro. Sense of direction: formal definition and properties. In *Proc. of 1st Colloquium on Structural Information and Communication Complexity, Sirocco'94*, Ottawa, Canada, 1994. 9–34.

[13] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum spanning tree. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983.

[14] O. Goldreich and L.Shrira. Electing a leader in a ring with link failures. *Acta Informatica*, 24:79–91, 1987.

[15] J.Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the A.C.M.*, 37(3):549–587, July 1990.

[16] A. Itai, S. Kutten, Y. Wolfstahl, and S. Zaks. Optimal distributed t-resilient election in complete networks. *IEEE Transactions on Software Engineering*, 16(1):415–420, April 1990.

[17] K.E. Johansen, U.L. Jorgensen, S.H. Nielsen, S.E. Nielsen, and S. Skyum. A distributed spanning tree algorithm. In *Proceedings of 2nd International Workshop of Distributed Algorithms (WDAG'87)*, volume 312 of *Lectures Notes in Computer Sciences*, pages 1–12, Amsterdam, 1987. Springer-Verlag.

[18] T.Z. Kalamboukis and S.L. Mantzaris. Towards optimal distributed election on chordal rings. *Information Processing Letters*, 38:265–270, 1991.

[19] J.L. Kim and G.G. Belford. A distributed election protocol for unreliable networks. *Journal of Parallel and Distributed Computing*, 35:35–42, 1996.

[20] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. *A.C.M. Transactions on Programming Languages and Systems*, 12(1):84–101, Jan 1990.

[21] E. Korach, S. Moran, and S. Zacks. Tight lower and upper bounds for a class of distributed algorithms for a complete network of processors. In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing (PODC'84)*, pages 199–207, Vancouver, Canada, August 1984.

[22] I. Lavallée and G. Roucairol. A fully distributed (minimal) spanning tree algorithm. *Information Processing Letters*, 23:55–62, Aug 1986.

[23] M.C. Loui, T.A. Matsushita, and D.B. West. Election in complete networks with a sense of direction. *Information Processing Letters*, 22:185–187, 1986. see also *Information Processing Letters*, vol.28, p.327, 1988.

[24] B. Mans. Optimal distributed algorithms in unlabeled tori and chordal rings. In *Proc. of the 3rd International Colloquium on Structural Information and Communication Complexity, Sirocco'96*, page in print, Siena, Italy, June 1996. Carleton Press.

[25] B. Mans and N. Santoro. On the impact of sense of direction in arbitrary networks. In *Proceedings of the 14th International Conference on Distributed Computing Systems, (ICDCS'94)*, pages 258–265, Poznan, Poland, June 21-24 1994.

[26] B. Mans and N. Santoro. Optimal fault-tolerant leader election in chordal rings. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing, (FTCS'94)*, pages 392–401, Austin, Texas, USA, June 15-17 1994.

[27] H. Masuyama and T. Ichimori. Tolerance of double-loop computer networks to multinode failures. *IEEE Transactions on Computers*, 38(5):738–741, May 1989.

[28] T. Masuzawa, N. Nishikawa, K. Hagihara, and N. Tokura. Optimal fault-tolerant distributed algorithms for election in complete networks with a global sense of direction. In *Proceedings of the 3rd International Workshop on Distributed Algorithms (WDAG'89)*, pages 171–182, Nice, France, 1989. Springer-Verlag.

[29] Sape Mullender. *Distributed Systems.* ACM Press, Addisson-Wesley, 1993.

[30] A. Nayak and N. Santoro. On reliability analysis of chordal rings. *Journal of Circuits, Systems and Computers*, 5(2):199–213, 1995.

[31] N. Nishikawa, T. Masuzawa, and N. Tokura. Fault-tolerant distributed algorithm in complete networks with link and processor failures. *IEICE Transactions on Information and Systems*, J74D-I(1):12–22, January 1991.

[32] Yi Pan. A near-optimal multi-stage distributed algorithm for finding leaders in clustered chordal rings. *Information Sciences*, 76(1-2):131–140, 1994.

[33] J.M. Peha and F.A. Tobagi. Comments on tolerance of double-loop computer networks to multinode failures. *IEEE Transactions on Computers*, 41(11):1488–1490, November 1992.

[34] C.S. Raghavendra, M. Gerla, and A. Avizienis. Reliable loop topologies for large local computer networks. *IEEE Transactions on Computers*, C-34(1):46–55, January 1985.

[35] N. Santoro. On the message complexity of distributed problems. *Journal of Computing Information Science*, 13:131–147, 1984.

[36] N. Santoro. Sense of direction, topological awareness and communication complexity. *SIGACT NEWS*, 2(16):50–56, summer 1984.

[37] H.M. Sayeed, M. Abu-Amara, and H. Abu-Amara. Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links. *Distributed Computing*, 9:147–156, 1995.

[38] G. Singh. Leader election in the presence of link failures. In *Proceedings of 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, page 375, Los Angeles, California, August 14-17 1994.

[39] J. Tyszer. A multiple fault-tolerant processor network architecture for pipeline computing. *IEEE Transactions on Computers*, C-37(11):1414–1418, November 1988.

[40] J. Wolf, M.T. Liu, B. Weide, and D. Tsay. Design of a distributed fault-tolerant loop network. In *Proceedings of the 9th Annual International Symposium on Fault-Tolerant Computing, (FTCS'79)*, pages 17–24, Madison, June 1979. IEEE.

[41] B. Yi. *Faults and Fault-Tolerance in Distributed Systems: the Election problem.* PhD thesis, Georgia Institute of Technology, 1994.

[42] B. Yi and G. Peterson. Election on faulty rings with incomplete size information. In *Proceedings of 6th IEEE Symposium on Parallel and Distributed Processing, (SPDP'94)*, Dallas, Texas, USA, October 26-29 1994.

Appendix: Fault-Tolerant Election

```
procedure Election(p)
begin
/* initially - processor is asleep */
(0) Upon RECEIPT of (WAKEUP)
          or other on chord d
    State_p := Warrior;
    Substate_p := Regular;
    K_out := nil
    for each faulty chord d detected
        Unused_p := Unused_p − {d}
        SubS_d := closed
    level_p := 0; king_p := id_p
    List_p := 0*; List_p[0] := 1
    Status_p := {king_p, level_p, List_p}
    if message = WAKEUP then
      ATTEMPT /* attempt on request */
    else process as a warrior who lost
end WAKEUP


• If (State_p = Citizen) or (State_p = King) :

(1) Upon RECEIPT of (REQUEST,Status)
      on chord r
    TRANSPOSE(List, r)
    if Status_p < Status then
      if Substate_p = Regular then
        SEND (REQUEST,Status) on chord W_out
        Substate_p := WaitingForSurrender
        ReqStatus := Status
        W_in := r
      /* control requests number */
      else delay message
    else kill message
end REQUEST

(2) Upon RECEIPT of (SURRENDER,Status)
      on chord r
    /* r must be W_out */
    TRANSPOSE(List, r)
    SEND (SURRENDER,Status) on chord W_in
    Substate_p := WaitingForStatus
end SURRENDER

(3) Upon RECEIPT of (NEWSTATUS,Status)
      on chord r
    TRANSPOSE(List, r)
    if king_p ≠ king then /* new kingdom */
      K_out := r
      if king_p = id_p
        then State_p := Citizen /* not king */
    fi
    Status_p := Status
    if (Substate_p ≠ Regular)
      and (ReqStatus < Status_p)
      then Substate_p := Regular
    for each traversed chord d except r
      SEND (NEWSTATUS,Status) on chord d
end NEWSTATUS

(4)Upon RECEIPT of (BACKTRACK,Status)
         on chord r
   TRANSPOSE(List, r)
   SubS_r := closed
   State_p := Warrior
   forall chord d with List[d] = 1
         do Unused_p := Unused_p − {d}
         ATTEMPT /* attempt on request */
         if (Substate_p ≠ Regular) then
         /* backtrack before receipt of request */
           SEND (SURRENDER,Status_p) on chord W_in
           Substate_p := WaitingForStatus
   else /* backtrack again */
         BTRACK
   fi
end NEWSTATUS

(5) Upon RECEIPT of (MOVEWARRIOR,Status)
         on chord r
   /* must be K_out */
   TRANSPOSE(List, r)
   State_p := Warrior
   forall chord d with List[d] = 1
         do Unused_p := Unused_p − {d}
   ATTEMPT /* attempt on request */
   if Unused_p = ∅ then
      BTRACK
   fi
end NEWSTATUS

• If State_p = Warrior :

(6) Upon RECEIPT of (REQUEST,Status)
      on chord r
   TRANSPOSE(List, r)
   if Status_p < Status then
     if Substate_p = Regular then
       W_in := r
       SEND (SURRENDER,Status_p) on chord W_in
       Substate_p := WaitingForStatus
     else delay message /* control requests number */
   else kill message
end REQUEST
```

```
(7) UponRECEIPT of (NEWSTATUS,Status)
        on chord r
    TRANSPOSE(List, r)
    List := List_p ∪ List
    K_out := r
    Status_p := Status
    Substate_p := Regular
    for each traversed chord d except r do
        SEND(NEWSTATUS,Status)
            on chord d
    forall chord d with List[d] = 1
        do Unused_p := Unused_p − {d}
    /* possible waiting requests */
    accept delayed messages
    if (State = Warrior) then
        if List[1..n] ≠ 1* then
            /* start a new attack */
            ATTEMPT
            if Unused_p = ∅ then BTRACK
        else /* no more node to attack */
            /* The graph is connected */
            for each d ∈ Traversed_p do
                SEND(TERMINATION,Status)
                    on chord d
                State_p = Citizen
            od
        fi
    fi
end NEWSTATUS

(8) Upon RECEIPT of (SURRENDER,Status)
        on chord r
    TRANSPOSE(List, r)
    List_p := List_p ∪ List
    if Opened_p = ∅ then State_p = King
        else State_p = Citizen
    fi
    SEND (TRAVERSE) on chord r
    S_r := traversed
    SubS_r := opened
    W_out := r /* new warrior direction */
    if Substate = Regular then
        if level = level_p then
            level_p := level_p + 1
            SEND (NEWSTATUS,Status_p)
                on all traversed chords
        else
            SEND (NEWSTATUS,Status_p)
                on chord r
        fi
    /* else: new status yet unknown,
        will forward it as a Citizen */
    fi
end SURRENDER

•forall State_p :

(9)Upon RECEIPT of (TRAVERSE)
        on chord r
    S_r := traversed
    SubS_r := opened
end TRAVERSE

(10) Upon RECEIPT of (TERMINATION,Status)
        on chord r
    /* if List ≠ 1* the graph is not connected */
    for each d ∈ Traversed_p − {r} do
        SEND (TERMINATION,Status) on chord d
    od
    terminate execution
    /* king_p and K_out already known */
end TERMINATION

procedure TRANSPOSE(List, r)
    NewList: array of [1..n] bits
    begin
        forall i ∈ [1..n] do
            NewList[(r + i) mod n] := List[i]
        forall i ∈ [1..n] do
            List[i] := NewList[i]
end TRANSPOSE

procedure BTRACK
    if Opened_p = ∅ then
        /* Election in the connected Component */
        for each d ∈ Traversed_p do
            SEND (TERMINATION,Status) on chord d
        od
    else
        if Opened_p ≠ {K_out} then
            /* give power to a son */
            SEND (MOVEWARRIOR,Status)
                on chord r ∈ Opened_p − {K_out}
            W_out := r
        else /* backtrack to its parent */
            SEND (BACKTRACK,Status) on chord K_out
            SubS_Kout := closed
            W_out := K_out
        fi
    fi
end BTRACK

procedure ATTEMPT
    if Unused_p ≠ ∅ then
        repeat Select unused chord r
                SEND (REQUEST,Status) on r
                if chord r is Faulty then
                    Failed_p := Failed_p + {r}
                Unused_p := Unused_p − {r}
        until Unused_p = ∅ or r non-Faulty
    fi end ATTEMPT

end Election(p)
```

List of Figures

Optimal Elections in Faulty Loop Networks and Applications

To appear in IEEE Transactions on Computers, Regular Paper #E-4427

Bernard Mans and Nicola Santoro
corresponding author: bmans@mpce.mq.edu.au

Author to contact:

Bernard Mans,
Dept. of Computing,
School of Mathematics, Physics, Computing and Electronics,
Macquarie University,
Sydney, NSW 2109,
Australia

bmans@mpce.mq.edu.au
phone: +61-2-9850-9574
Fax: +61-2-9850-9551