OPTIMAL FAULT-TOLERANT LEADER ELECTION IN CHORDAL RINGS

Bernard Mans and Nicola Santoro

TR-245 MAY 1994

School of Computer Science, Carleton University
Ottawa, Canada, KIS 5B6

Optimal Fault-Tolerant Leader Election in Chordal Rings*

Bernard Mans‡ (mans@scs.carleton.ca)

Nicola Santoro§ (santoro@scs.carleton.ca)

Abstract

Chordal rings (or 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 chordal rings where an arbitrary number of links has failed and a processor can only detect the status of its incident links.

We shows that a Leader Election protocol in a faulty chordal ring 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. 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 occured; depending on the application, these distintives elements can thus take the appropriate actions.

1 Introduction

1.1 Chordal Rings

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

*This paper also in IEEE-proceedings of the FTCS'94, 24th Annual International Symposium on Fault-Tolerant Computing, June 15-17 1994, Austin, Texas, USA

†This research supported in part by N.S.E.R.C, grant#A2415. This work has been done while the first author was visiting School of Computer Science, Carleton University, supported by an INRIA postdoctoral fellowship, 78153 Le Chesnay Cedex, France.

‡support by Université du Québec à Hull, Case postale 1250, succ "B", Hull, Québec J8X 3X7 Canada

§School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6 Canada, phone: (613) 788-4333, Fax: (613) 788-4334

network. With alternate paths between nodes, the network can sustain several nodes and links failures. Several ring networks, suggested in [2, 7, 11, 21, 28, 33] 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 **Circulant Graph** or **Chordal Ring**.

A chordal ring $\mathcal{C}_n\langle d_1, d_2, ..., d_k\rangle$ of size n and k-chord structure $\langle d_1, d_2, ..., d_k\rangle$ is a ring R_n of n processors $\{p_0, p_2, ..., p_{n-1}\}$, where each processor is also connected to the processors at distance $1, d_1, d_2, \cdots, d_k, n-d_k, \cdots, n-d_2, n-d_1, n-1$ 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 this labelling is globally consistent; that is, it forms a *Sense of Direction* (as defined in [30]). 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, [5, 6].

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 chordal rings have been widely investigated to analyze their fault-tolerant properties [2, 6, 7, 8, 11, 21, 27, 29] and solutions have been proposed for reconfiguration after links and/or node failures [23, 24, 32]. 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.

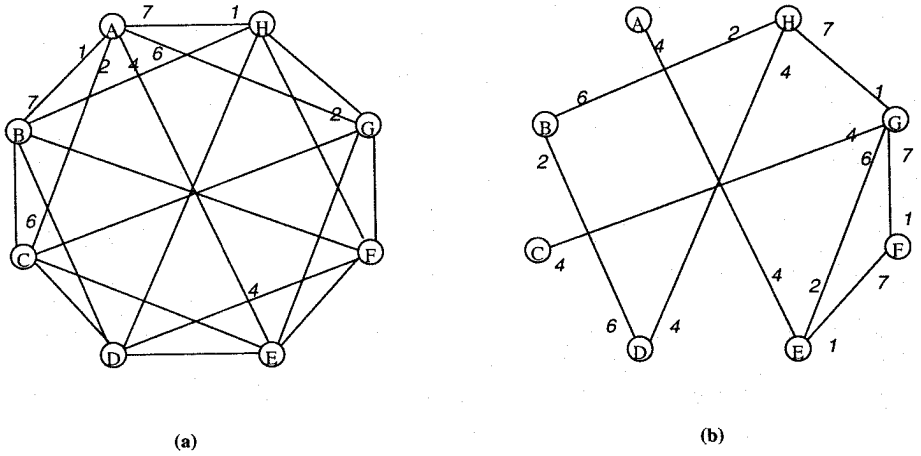

Figure 1: $\langle 2, 4\rangle$ Chordal Ring (a) with Faulty Links (b)

1.2 Election

In distributed systems, one of the fundamental control problem is the **Leader Election**; that 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. 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, they do not know the identities of their neighbours).

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.

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. An execution is a sequence of events, each being one of the above three statements. We assume that the messages on each arc arrive with no error, in a finite but unbounded 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 Chordal Ring

The Leader Election problem in chordal rings has been extensively studied assuming that there are no failures in the systems [3, 15, 17, 19, 20, 26, 31]. 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 [9].

The research has thus focused on studying the problem in more restricted environments: (**r1**) failures are detectable, (**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 chordal rings have been developed under assumptions (r2), (r3), (r4) and (r5) [1, 13, 22, 25]. Note that, restriction (r5) (there is a link between any two processors) makes these results applicable only to the special case of chordal rings which are complete graphs.

Graph	# Faults (r3) Links	Nodes	(r1) Detectability	(r2) Occurrence	Termination
Arbitrary Chordal Ring	0	1	No	Arbitrary	Impossible [9]
Complete (r4)	0	$\leq t$	No	Prior	Possible [1, 13, 22]
Complete (r4)	$\leq k$	$\leq t$	No	Prior	Possible [25]
Arbitrary Chordal Ring	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).

In this paper, we consider the Election Problem in asynchronous arbitrary chordal rings 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. We prove that, under these assumptions, a Leader Election protocol in a faulty chordal ring 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. 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 occured; depending on the application, these distintives elements can thus take the appropriate actions. We will now describe the algorithm as executed in each active component; if no partition has occurred, a leader will be elected.

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. A processor can only detect the failure of its incident links. We emphasize the fact that both regular and bypass links can fail (as shown in figure 1(b)).

It is worthy of attention that the detectability assumption (r1) is required to cope with an unbounded number of faulty components (see table 1). Knowledge that a link is faulty can be either *a priori* (namely, in a case where the hardware subsystem provides directly such a knowledge to the processors) or can only be acquired upon an attempt to transmit. From a computational point of view, the latter case is more difficult than the former. 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 $\Omega(e)$ messages. 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 is based on the protocol by [18] for election in non-faulty network with sense of direction; it uses and combines different techniques developed in

[10, 14, 16] for election in non-faulty networks. The full algorithm is given in the Appendix.

2 Election Algorithm

We present an Election algorithm in chordal ring, 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. Any node can independantly 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 occured; depending on the application, these distintives elements can thus take the appropriate actions. We will now describe the algorithm as executed in each active component.

2.1 Description

The algorithm builds a *Rooted Spanning Tree* or *Kingdom* in each component by repeatly 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. 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 contains such a local view in the request message containing a request status $ReqStatus = (reqking, reqlevel, reqList)$.

Informally, the attack is carried out only by a warrior; the warrior will select among its links an outgoing link (i.e., a link which leads to another 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 will be taken; otherwise, the REQUEST message will carry the attack to the other kingdom, as shown in figure 2.

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

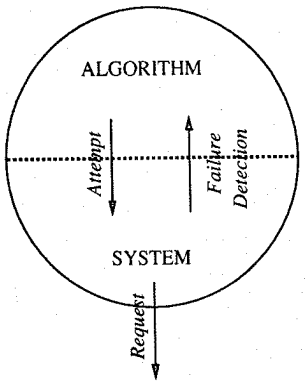

Figure 2: Local Failure Detection

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 completly explored and thus can lead to nodes which have not been reached yet). 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 becames *closed* (initially, all non-faulty chords are *opened*). 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 for $warrior_p$, the *warrior* attacking p, and $king_p$ respectively.

Define the $status_p$ of node p as $(level_p, king_p, List_p)$. We say that $status_p > status_j$ if:
- either (a) $level_p > level_j$
- or (b) $level_p = level_j$ and $king_p > king_j$.

Our algorithm obeys two main rules:

Rule A: a warrior p can only successfully attack a kingdom with *status* less than its own. Let the attack by warrior p be successfull. 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 $level_p$. 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 $warrior_j$.

Rule B (controls the number of messages during each phase): three differents 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 (message is killed).

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 path waits for the new status.

3. $status_p = status_j$: as proved later, this case (i.e. an attack within the same kingdom) can not 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 Rule A. The new local view *List* is obtained by merging the two Lists. The initial local view is a list of bits $List[0..(n-1)]$ initialized to 10^* (i.e. all bits are set to 0 except $List[0]$ which is set to 1).

The number of parallel incoming attacks in a kingdom must be limited in order to guarantee a message complexity of $O(n)$ for each phase. 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).

The reason to introduce some substates is 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 (Rule B2).

Secondly, an incoming attack can be received before knowing that the kingdom has already absorbed (or been absorbed by) an other 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 back to *regular* substate. Otherwise, the current attack status is still legal; thus, the inhibition waiting substate must be kept.

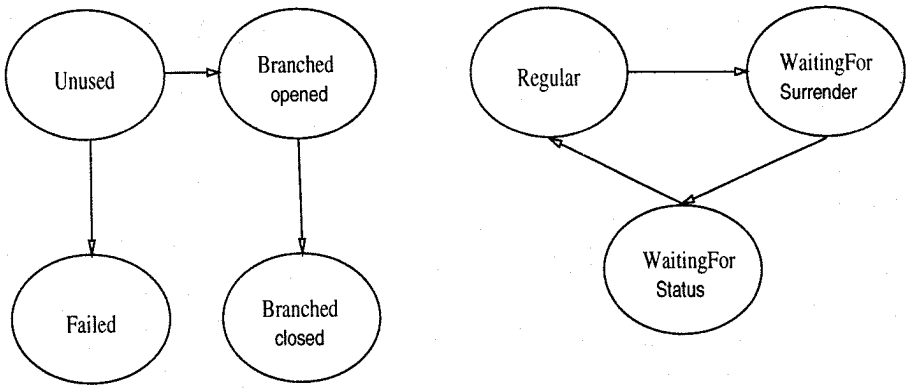

Figure 3: State Transition Diagrams

The extreme case of this 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. Consistently with Rule B, 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 total ordering on the *status* forbids the creation of waiting cycles.

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 depth first search strategy) and all its incident chords will be *closed.* 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. 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 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 Chordal ring $\mathcal{C}_{16}\langle 3, 8\rangle$ is shown in figure 4). The result of the successful attack is shown in figure 6.

Messages Used:

(i) (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,

(ii) (SURRENDER,Status): it is sent by a defeated warrior to inform the winner of its success,
(iii) (NEWSTATUS,Status): it is broadcasted by the winner on the appropriate tree (depending on Rule A),
(iv) (BRANCH): it is sent by a successful warrio on the chord connecting the two trees,
(v) (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,
(vi) (MOVEWARRIOR,Status): it is sent by the warrior to one of its opened chords after a backtracking, backtrack process,
(vii) (TERMINATION): it is broadcasted 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).

The proof of correctness is similar to [18] and is omitted here for brevity.

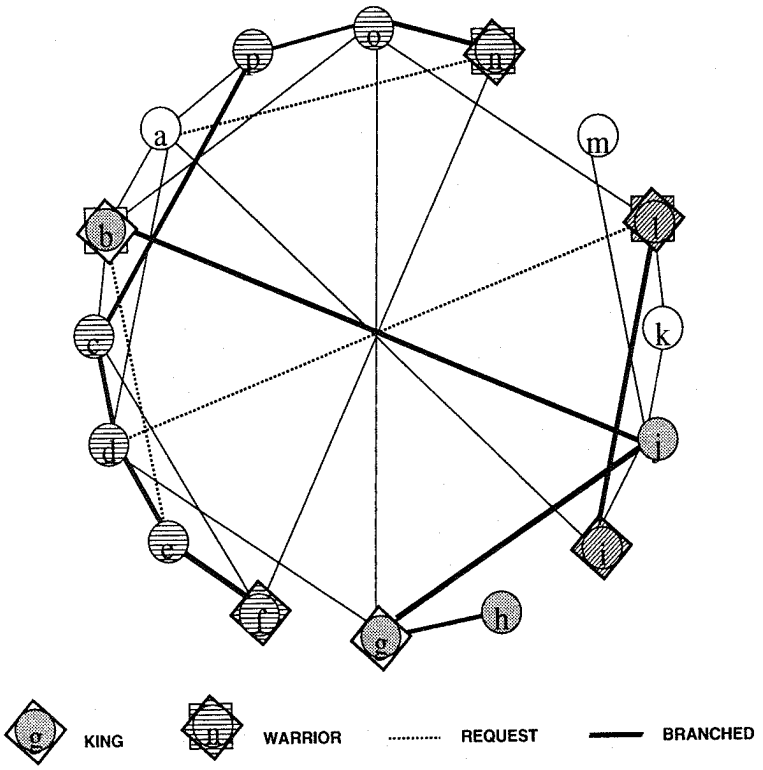

Figure 4: Kingdoms in $\mathcal{C}_{16}\langle 3, 8\rangle$

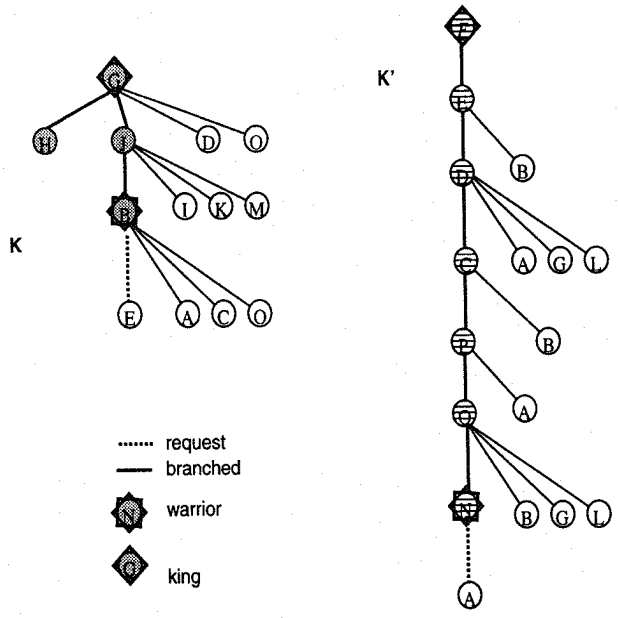

Figure 5: Two Kingdoms in $\mathcal{C}_{16}\langle 3, 8\rangle$

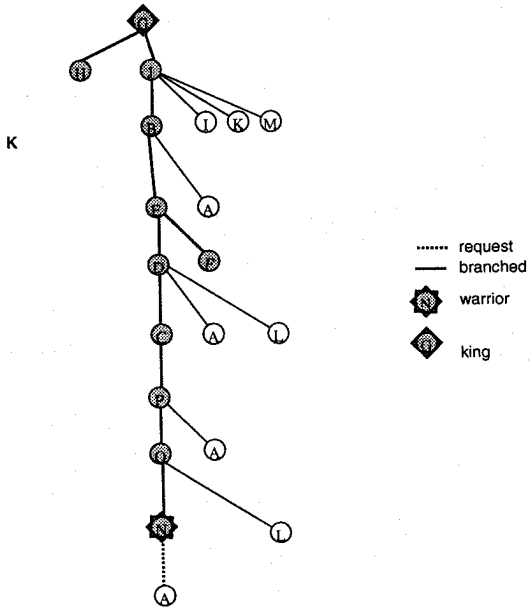

Figure 6: Result of Attack in $\mathcal{C}_{16}\langle 3, 8\rangle$

2.2 Complexity analysis

The measure of efficiency analysed here is the communication complexity. Note that the attempt to transmit on a failed chord is not a transmission.

Lemma 2.1 *The number of phases is at most $\log k$ for each kingdom, if k independent nodes start the algorithm.*

Proof By Rule A. □

Corollary 2.1 *The number of surrender messages sent by a warrior during a particular execution is at most $\log n$.*

Lemma 2.2 *For a given phase and a given non-faulty chord l in a kingdom, at most one request will be transmitted through the chord l.*

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

1. the request is successful: it will cause the phase to increase.
2. 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 an other attack, but the nodes incident on this chord does not know it yet.

Therefore, following a request message, only a surrender and/or a new status messages will travel through l during this phase. A similar argument can be used for a branched chord between two kingdoms. □

More precisely,

Theorem 2.1 *The total number of messages used by the algorithm does not exceed $3\,n \log k + 4(n-1)$*

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

REQUEST : sent, at a given phase, through at most $n-1$ non-faulty chords (see Lemma 2.2). Hence, the total number of such request messages sent during the whole execution is bounded by $n \log k$.

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 $n \log k$.

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

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 can not 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 can not 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 differents 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.

2.3 Worst-Case Optimality

Theorem 2.2 *The algorithm has an optimal worst-case message complexity.*

Proof Given a chordal ring 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 [4] on oriented rings. □

3 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 [3, 15, 17, 26, 31]. However, to achieve this complexity, it is required that the absence of failures is *a priori* known (more specifically, it is *common knowledge* [12]) 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 chordal rings 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.

4 Concluding Remarks

In this paper, we have presented a $\Theta(n \log n)$ solution for the Election problem in chordal rings 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 occured; depending on the application, these distintives 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.

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

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

[4] 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.

[5] J. Bruck, R. Cypher, and C.-T. Ho. Efficient fault-tolerant meshes and hypercubes architectures. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS'92)*, pages 162–169, Boston, MA, July 8-10 1992. IEEE.

[6] 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.

[7] 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.

[8] S. Dutt and J.P. Hayes. An automorphic approach to the design of fault-tolerant multiprocessors. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing, (FTCS'89)*, pages 496–503, Chicago, 21-23 June 1989.

[9] 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.

[10] 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.

[11] A. Grnarov, L. Kleinrock, and M. Gerla. A highly reliable distributed double loop network architecture. In *Proceedings of the 10th International Symposium of Fault-Tolerant Computing (FTCS'80)*, pages 319–324, Kyoto, Japan, October 1980.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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 Symposium on Principles of Distributed Computing (PODC'84)*, pages 199–207, Vancouver, Canada, August 1984.

[17] 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.

[18] 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)*, Poznan, Poland, June 21-24 1994. to appear.

[19] S.L. Mantzaris. Almost optimal election in chordal rings. In S. Tzafestas, P. Borne, and L. Grandinetti, editors, *Proceedings of Parallel and Distributed Computing in Engineering Systems, (PDCOM'91)*, pages 459–464, 23-28 June 1991.

[20] G.H. Masapati and H. Ural. Effect of preprocessing on election in a complete network with a sense of direction. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, volume 3, pages 1627–1632, 1991.

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

[22] 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.

[23] A. Nayak and N. Santoro. Limits to reconfigurability of loop topologies. In S. Tzafestas, P. Borne, and L. Grandinetti, editors, *Proceedings of Parallel and Distributed Computing in Engineering Systems, (PDCOM'91)*, pages 453–458, 23-28 June 1991.

[24] A. Nayak and N. Santoro. Performance analysis of redundant ring networks. In D.J. Evans, G.R. Joubert, and H. Liddell, editors, *Proceedings of the International Conference on Parallel Computing, (ICPC'91)*, number 4 in Advances in Parallel Computing, pages 311–322, London, U.K., September 3-6 1991. North-Holland.

[25] 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.

[26] Yi Pan. An improved election algorithm in chordal ring networks. *International Journal of Computer Mathematics*, 40(3-4):191–200, 1991.

[27] 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.

[28] 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.

[29] C.S. Raghavendra and J.A. Silvester. A survey of multi-connected loop topologies for local computer networks. *Computer Networks and ISDN Systems*, 11(1):29–42, January 1986.

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

[31] Gurdip Singh. Leader election in complete networks. In *Proceedings of 11th Symposium on Principles of Distributed Computing (PODC'92)*, pages 179–190, August 1992.

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

[33] 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.

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}
$Substate_p := WaitingForSurrender$
$ReqStatus := Status$
$W_{in} := r$
else delay message /* control requests number */
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 $king_p \neq king$ **then** /* new kingdom */
  $K_{out} := r$
  if $king_p = id_p$
    then $State_p := Citizen$ /* no more king */
fi
$Status_p := Status$
if ($Substate_p \neq Regular$) **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 ($Substate_p \neq Regular$) **then**
  /* previous warrior 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 $Unused_p = \emptyset$ **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) Upon RECEIPT 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}
accept delayed messages /* possible waiting requests */
if (State = Warrior) then
  if List[1..n] ≠ 1* then /* start a new attack */
    ATTEMPT /* attempt on request */
    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 the List ≠ 1* then 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
  /* Complete 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)
```

School of Computer Science, Carleton University
Recent Technical Reports

TR-225 **Mixture Decomposition for Distributions from the Exponential Family Using a Generalized Method of Moments**
S.T. Sum and B.J. Oommen, June 1993

TR-226 **Switching Models for Non-Stationary Random Environments**
B. John Oommen and Hassan Masum, July 1993

TR-227 **The Probability of Generating Some Common Families of Finite Groups**
Vincenzo Acciaro, September 1993

TR-228 **Power Roots of Polynomials over Arbitrary Fields**
Vincenzo Acciaro, September 1993

TR-229 **Optimal Parallel Algorithms for Direct Dominance Problems**
Amitava Datta, Anil Maheshwari and Jörg-Rüdiger Sack, October 1993

TR-230 **Uniform Generation of Forests of Restricted Height**
M.D. Atkinson and J.-R. Sack, October 1993

TR-231 **Optimal Elections in Labeled Hypercubes**
Paola Flocchini and Bernard Mans, December 1993

TR-232 **On the Complexity of Computing Gröbner Bases in Characteristic 2**
Vincenzo Acciaro, December 1993

TR-233 **Broadcasting Session Keys**
Mike Just, Evangelos Kranakis, Danny Krizanc, and Paul van Oorschot, February 1994

TR-234 **String Taxonomy Using Learning Automata**
B. John Oommen and Edward V. de St. Croix, March 1994

TR-235 **Distributed Cyclic Reference Counting**
Frank Dehne and Rafael D. Lins, March 1994

TR-236 **Exact and Approximate Computational Geometry Solutions of an Unrestricted Point Set Stereo Matching Problem**
Frank Dehne and Katia Guimaraes, March 1994

TR-237 **Scalable and Architecture Independent Parallel Geometric Algorithms with High Probability Optimal Time**
Frank Dehne, Claire Kenyon and Andreas Fabri, March 1994

TR-238 **Finding the Extrema of a Distributed Multiset**
Paola Alimonti, Paola Flocchini and Nicola Santoro, March 1994

TR-239 **Killing Two Birds with One Stone**
Evangelos Kranakis, Danny Krizanc, Anil Maheshwari, Jörg-Rüdiger Sack, Jorge Urrutia, April 1994

TR-240 **Some Computational Problems on Central Simple Algebras over Q**
Vincenzo Acciaro, April 1994 (Not available)

TR-241 **Extending Cryptographic Logics of Belief to Key Agreement Protocols**
Paul C. van Oorschot, May 1994

TR-242 **Modern Key Agreement Techniques**
Rainer A. Rueppel and Paul C. van Oorschot, May 1994

TR-243 **On Unifying Some Cryptographic Protocol Logics**
Paul F. Syverson and Paul C. van Oorschot, May 1994

TR-244 **Efficient DES Key Search**
Michael J. Wiener, May 1994