On the Impact of Sense of Direction in Arbitrary Networks

Bernard Mans

Département d'Informatique
Université du Québec à Hull
Hull, J8X 3X7 Québec
Canada

Nicola Santoro

School of Computer Science
Carleton University
Ottawa, K1S 5B6 Ontario
Canada

Abstract

In this paper, we study the positive impact that the availability of "Sense of Direction" has on the message complexity of the Election problem in arbitrary networks of processors.

We present a $\Theta(n \log n)$ solution; without sense of direction, this problem requires $\Omega(e + n \log n)$ messages where e is the number of communication links.

This result confirms and extends the evidence on the impact of sense of direction which, up to new, was established only for specific classes of topologies.

1 Introduction

In this paper, we study the positive impact that the availability of "Sense of Direction" has on the message complexity of distributed problems in arbitrary networks of n asynchronous processors connected by e bidirectional communication links. The processors may have distinct identities but do not know the identities of their neighbors.

The **Sense of Direction** refers to the capability of a processor to distinguish between adjacent communication lines, according to some globally consistent scheme [21]. For example, in a *ring* network this property is 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*, labelings "up" and "down" are added. Sense of direction in a *complete network* is the knowledge of some predefined Hamiltonian cycle and the existence of a label on each link at each processor, that represents the distance along this cycle to the processor at the other end of the link. A similar definition exists for *chordal rings.* The existence of an intuitive labeling based on the dimension provides a Sense of Direction for *hypercube* (each edge between two nodes is labeled on each node by the dimension of the bit of the identity in which they differ).

For these networks, the availability of sense of direction has been shown to have some impact on the message complexity of the Election problem. For example in a *complete graph*, the Election problem can be solved using $O(n)$ messages if sense of direction is available [12, 14, 22] but requires $\Omega(n \log n)$ messages otherwise [10]; such an improvement can be achieved even if each node has sense of direction on just $O(\log n)$ of its incident arcs [1]. Similar results hold for *circulant graphs* or *chordal rings* [1, 8, 17]. In *oriented tori*, an $O(N)$ algorithm has been given [18]. In [3, 19, 23], it is shown how to elect a leader using at most $O(N)$ messages when a Sense of Direction is available for the *hypercube* (which is not a chordal ring).

In an *arbitrary network*, we define a globally consistent labeling on the links by extending in a natural way the existing definitions for particular toplologies. Fix a cyclic ordering of the processors. The network has a 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 1. Note that such a definition intrinsicly 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 *chordal ring* or *circulant graph.* Furthermore, in *hypercubes*, this sense of direction is derivable in $O(N)$ messages from the traditional one [3].

1063-6927/94 $03.00 © 1994 IEEE

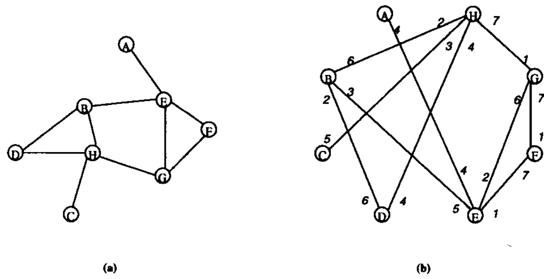

Figure 1: Arbitrary network (a) with sense of direction (b)

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

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 [20], and such a bound is achievable [5]. In contrast, we show 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 [2], it follows that $\Omega(n \log n)$ is also a lower bound on the general. Thus, the bound is tight.

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 [9, 11]. 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).

The algorithm achieving the bound is a novel and non-trivial combination of some existing election techniques for arbitrary graphs without sense of direction.

The Election problem is considered in Section 2, where a $\Theta(n \log n)$ solution is presented. The full description of the algorithm is given in the appendix. The last section contains some concluding remarks.

2 Election with Sense of Direction

We consider an arbitrary network of asynchronous processors. 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. The *election* (or *leader finding*) problem consists in moving from an initial configuration where all the processors are in the same state to a final configuration where exactly one processor is in a *leader* state and all other processors are in state *lost*. The algorithm may be independently started by any subset of the processors.

The full algorithm is given in the Appendix; it is based on the algorithm given in [5]. It uses and combines different techniques developed in [7, 10].

2.1 Informal description

The algorithm builds a *Rooted Spanning Tree* by repeatly combining smaller spanning trees. The final root is the leader.

The algorithm proceeds in phases. Initially, each node of the network is a *king*. At the end, all nodes are *citizen* except one which is still a *king*, i.e. the leader. 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 roots : the *king* and the *warrior*. Each king is assigned a *level*, initialized at zero. Each node p stores the identity of its $king_p$, the level $level_p$ of its king, the label of the outgoing arcs to its king and to its warrior. If a node is attacked, it stores the label of the incoming arc from which the attack came.

Our algorithm heavily uses the sense of direction when passing the knowledge of the processors (represented by their distances) that have been already reached. The fundamental property we use is the fact that a processor p_1, at distance d_1 from processor p_2 which is at distance d_2 from processor p_3, is at distance $(d_1 + d_2) \bmod n$ from p_3. Thus, when node p_1 receives a message from node p_2 by the incident arc labeled d_1, node p_1 can unambiguously "decode" the information about other nodes contained in the message. This fact will be used to determine whether an unused arc (i.e., on which no messages have been sent) is outgoing (i.e., connects to a different kingdom) or not. In particular, in the algorithm, each warrior p maintains a local view $List_p$ of all the others processors with the indication of which of them belongs to the kingdom. An attack message contains such a local view.

The algorithm uses two types of control mechanisms: cooperation and concurrency control.

Cooperation control

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 of the new king $king_p$ 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 new warrior is the warrior j. The rule on the level insures that a kingdom consists of at least 2^{level} nodes. The number of combinations (i.e. phases) for each kingdom is therefore at most $\log n$.

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:

- $status_p$ < $status_j$: the warrior is not strong enough to attack this kingdom and, thus, its attack fails (message is killed).
- $status_p$ > $status_j$: 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.

- $status_p$ = $status_j$: 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.

Concurrency control

Several types of local information are used to control the execution and the message complexity:

- *Arc Substate.* The attacks by a kingdom follow a **Depth First Search** strategy along the network. A substate for each branched arc is introduced to specify if the arc is *closed* (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). If a warrior j cannot reach any node outside the kingdom (this is locally determined by the local view $List_j$), then the state of *warrior*, together with $List_j$, is backtracked to its parent and the arc between them becames *closed* (initially, all branched arcs are *opened*). This strategy has the main advantage to limit the number of backtracking after a combination compared to a *breadth first search* strategy.
- *Node Substate.* 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 for each node 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).

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.

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

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 (see theorem 2.2).

Termination

The algorithm terminates when the warrior has no more processors to reach, that is when the kingdom includes all nodes of the graph. Then, the warrior just broadcasts along the tree the termination message which notifies the king which becomes the leader. Each node knows already the *id* of the *king* and knows its outgoing arc which leads to it.

An example of a successful attack is shown in figure 2, where the kingdom K' has a greater status than the kingdom K.

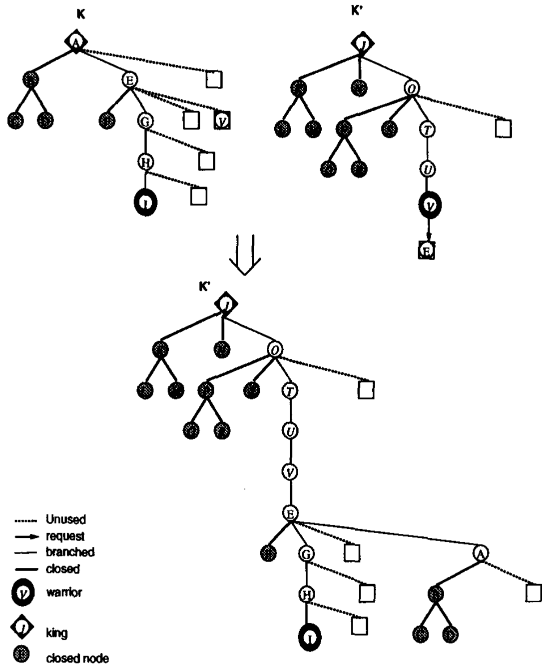

Figure 2: Kingdom K' absorbes the Kingdom K.

2.2 Messages and information used

Information used :

• $Neighbour_p$: list of labels of the arcs at node p.

• $State_p$: $\{King, Warrior, Citizen, Leader\}$ initially $King$.

• $Substate_p$: $\{WaitingForSurrender, WaitingForStatus, Regular\}$; initially $Regular$. Used to control the number of attacks received during each phase by inhibiting the message.

• S_r : $\{unused, branched\}$; initially *unused*. State of an arc r at node p ($Unused_p$ and $Branched_p$ represent the set of the arcs in the corresponding states).

• $SubS_r$: $\{opened, closed\}$; initially *opened*. Substate of a $Branched_p$ arc r at node p ($Opened_p$ and $Closed_p$ represent the set of the arcs in such a state). Used to control the backtracking by closing a subtree which has been completed (from which no new node can be reached).

• $List[0..(n-1)]$: list of bits, initialized to 10^* (i.e. all bits are set to 0 except $List[0]$ which is set to 1; it represents the local view of the kingdom with respect to the labeling.

• $\{W_{out}, W_{in}, K_{out}\}$: labels at a node p of the incident arc for $warrior_p$, the *warrior* attacking p, and $king_p$ respectively.

• $Status_p = (king_p, level_p, List_p)$: status of node p.

• $ReqStatus = (reqking, reqlevel, reqList)$: status of an attack.

Messages Used :

• *(REQUEST,Status)* : it is an attack by a warrior, and is forwarded to its adversary,

• *(SURRENDER,Status)* : it is sent by a defeated warrior to inform the winner of its success,

• *(NEWSTATUS,Status)* : it is broadcasted by the winner on the appropriate tree (depending on Rule A),

• *(BRANCH)* : it is sent by a successful warrior on the edge connecting the two trees,

• *(BACKTRACK,Status)* : it is sent by the warrior to its parent when all its arcs have been closed, that is when all the nodes reachable through this arc are part of the kingdom,

• *(MOVEWARRIOR,Status)* : it is sent by the warrior to one of its opened arcs after a backtracking,

• *(LEADER)* : it is broadcasted by the sole remaining warrior to terminate the algorithm.

Any number of processors can spontaneously start the execution of the algorithm; this is modeled by the reception of a WAKEUP message.

2.3 Correctness

The proof of correctness is similar to [10]. Several properties are introduced for the correctness and the analysis of the message complexity. In the following, numbers between parentheses refer to corresponding sections of the algorithm in the Appendix. All the proofs are sketched.

Lemma 2.1 *A request message traverses first an unused arc. All other arcs traversed by the message are branched.*

Proof A citizen (or king) can send a request only upon receipt of a request (1). The warrior sends the request through an opened arc (7). □

Lemma 2.2 *An arc becomes branched if a warrior has previously sent a request through it.*

Proof A Branch message is sent by a warrior only upon receipt of a surrender message (8), which must have been preceded by a request (6). □

Theorem 2.1 *A kingdom is a directed tree with the king as root.*

Proof By induction. Initially, each kingdom is one node tree (0). The kingdom is defined by the subgraph composed by the branched arcs and their incident nodes.

An attack can only be done upon receipt of a new status which contains the list of all the nodes belonging to the kingdom (7). Therefore, no cycle can be created in the kingdom.

Following a successful attack, the arc connecting the two trees (the absorbing and the absorbed ones) becomes part of the kingdom upon receipt of a BRANCH message (7) sent by the winner warrior. This message is followed by a new status message which will be broadcasted through the absorbed kingdom.

The outgoing arc 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 absorbtion 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 in the new king. □

Lemma 2.3 *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 2.2 *A deadlock may not be introduced by the waiting which arises when some nodes must wait until some condition holds.*

Proof In the algorithm, the only situation in which a node is blocked waiting for an event is when a warrior waits for a new status message. This occurs only after sending a surrender (1). Such a surrender message can only be deferred at the successful warrior node if it has surrendered to another warrior attack (8). Because of the total ordering defined by Rule A, a cycle of such events can not be created. □

By Theorem 2.2 and 2.1, the following theorem holds:

Theorem 2.3 *The algorithm correctly elects a leader.*

2.4 Complexity analysis

Rule A insures that the number of phases is at most $\log n$ for each kingdom, in fact at most $\log k$ if k independent nodes start the algorithm.

Lemma 2.4 *The number of surrender messages sent by a warrior during a particular execution is at most* $\log n$.

Proof By Rule A. □

Lemma 2.5 *For a given phase and a given arc l in a kingdom, a unique request could be passed.*

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

- the request is successful : it will cause the phase to increase.
- 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 arc 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 arc between two kingdoms. □

More precisely,

Theorem 2.4 *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] : at a given phase, this message is sent through at most $n-1$ arcs (see Lemma 2.5). Hence, the total number of such request messages sent during the whole execution is bounded by $n \log k$.
- [SURRENDER]: this message is 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]: this message is 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* arc of the kingdom, i.e. exactly $n-1$ times.
- [BACKTRACK] : sent on each *branched* arc of the kingdom if the subtree can not reach further nodes. Hence, the total number of such messages sent during the whole execution is bounded by the size of the spanning tree, i.e. $n-1$.
- [MOVEWARRIOR]: sent on each *opened-branched* arc of the kingdom if the node can not reach further nodes. Hence, the total number of such messages sent during the whole execution is also bounded by the size of the spanning tree, i.e. $n-1$.
- [LEADER]: exactly $n-1$ 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.

3 Concluding Remarks

In this paper, we have studied the positive impact that the availability of "Sense of Direction" has on the messages complexity of distributed problems in arbitrary networks of processors.

For the Election problem, we presented a $\Theta(n\, log\, n)$ solution; without sense of direction, this problem requires $\Omega(e + n\, log\, n)$ messages where e is the number of communication links.

This result confirms and extends the evidence on the impact of sense of direction which, up to new, was established only for specific classes of 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 [9, 11]. 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).

Our result is quite general. In fact, our algorithm can be easily modified to solve the Election problem with the same complexity with any sense of direction [4].

Several issues must still be addressed. First of all, the time complexity, which is out of the scope of this study, appears to be the weak point of this algorithm and should be improved. Because of the waiting process and different phases, the execution can become fully serialized giving an $O(n\ log\ k)$ to ideal time complexity (i.e. assuming that a message transmission takes one time unit and local processing is negligible).

Secondly, the issue of sense of direction in presence of faults must be addressed. Some results have already been established for special cases [6, 13, 15, 16].

Acknowledgements

Research supported in part by N.S.E.R.C. under grant #A2415, and by INRIA France under postdoctoral fellowship.

References

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

[2] 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.

[3] P. Flocchini and B. Mans. Optimal elections in labeled hypercubes. Technical Report TR-93-231, School of Computer Science, Carleton University, Carleton University, Ottawa, Canada, 1993.

[4] P. Flocchini, B. Mans, and N. Santoro. Sense of direction: Formal definitions and properties. In *Proc. Colloquium on Structural Information and Communication Complexity (SICC)*, Ottawa, Canada, 1994. to appear.

[5] 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.

[6] 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, Apr 1990.

[7] K.E. Johansen, U.L. Jorgensen, S.H. Nielsen, S.E. Nielsen, and S. Skyum. A distributed spanning tree algorithm. In *Proc. 2nd Int. Workshop of Distributed Algorithms (WDAG)*, pages 1–12, Amsterdam, 1987.

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

[9] E. Korach, 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.

[10] 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 *Proc. 3rd Symp. on Principles of Distributed Computing (PODC)*, pages 199–207, Vancouver, Canada, Aug 1984.

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

[12] 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 IPL, vol.28, p.327, 1988.

[13] B. Mans and N. Santoro. Optimal fault-tolerant leader election in chordal rings. In *Proc. 24th Annual Int. Symp. on Fault-Tolerant Computing (FTCS'94)*, Austin, Texas, USA, June 15-17 1994. to appear.

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

[15] 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 *Proc. 3rd Int. Workshop on Distributed Algorithms (WDAG)*, pages 171–182, Nice, France, 1989.

[16] 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, Jan 1991.

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

[18] G.L. Peterson. Efficient algorithms for elections in meshes and complete networks. Technical Report TR-140, Dept. of Computer Science, Univ. of Rochester, Rochester, NY-14627, 1985.

[19] S. Robbins and K.A. Robbins. Choosing a leader on a hypercube. In N. Rishe, S. Najathe, and D. Tal, editors, *Proc. Int. Conf. on Databases, Parallel Architectures and their Applications (PARBASE)*, pages 469–471, Miami Beach, Florida, USA, 1990.

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

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

[22] Gurdip Singh. Leader election in complete networks. In *Proc. 11th Symp. on Principles of Distributed Computing (PODC)*, pages 179–190, Aug 1992.

[23] G. Tel. Linear election for oriented hypercube. Technical Report TR-RUU-CS-93-39, Utrecht University, Department of Computer Science, Utrecht, The Netherlands, 1993.

Appendix: Election algorithm

```
procedure Election(p)
begin
/* initially - processor is asleep */
(0) Upon RECEIPT of (WAKEUP) or other on arc d
      State_p := Warrior; Substate_p := Regular ;
      K_out := nil
      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
         SEND (REQUEST,Status) on arc r ∈ Unused_p
         Unused_p := Unused_p - {r}
      else process as a warrior who lost
end WAKEUP
```

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

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

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

(3) Upon RECEIPT of (NEWSTATUS,Status) on arc r
      TRANSPOSE(List, r)
      if king_p ≠ king then /* new kingdom */
         K_out := r
         if king_p = id_p then State_p := Citizen /* lost */
      fi
      Status_p := Status
      if (Substate_p ≠ Regular) and (ReqStatus < Status_p)
         then Substate_p := Regular
      for each traversed arc d except r
         SEND (NEWSTATUS,Status) on arc d
end NEWSTATUS
```

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

(5) Upon RECEIPT of (MOVEWARRIOR,Status) on arc r
      /* must be K_out */
      TRANSPOSE(List, r)
      State_p := Warrior
      forall arc d with List[d] = 1 do
        Unused_p := Unused_p − {d}
      if Unused_p ≠ ∅ then
        SEND (REQUEST,Status) on r ∈ Unused_p
        Unused_p := Unused_p − {r}
      else /* backtrack again */
        BTRACK
      fi
end NEWSTATUS
```

• **If** $State_p = Warrior$:

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

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

(8) Upon RECEIPT of (SURRENDER,Status) on arc r
      TRANSPOSE(List, r)
      List_p := List_p ∪ List
      if Opened_p = ∅ then State_p = King
        else State_p = Citizen
      fi
      SEND (TRAVERSE) on arc labeled 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 traversed arcs
        else SEND (NEWSTATUS,Status_p) on arc r
        fi
      /* else: do not know yet the new status,
        will forward it as a Citizen */
      fi
end SURRENDER
```

• **forall** $State_p$:

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

(10) Upon RECEIPT of (LEADER,id) on arc labeled r
      terminate execution /* king_p and K_out known */
end LEADER

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 ≠ {K_out} then /* give power to a son */
        arc r ∈ Opened_p − {K_out}
        SEND (MOVEWARRIOR,Status) on arc r
        W_out := r
      else /* backtrack to its parent */
        SEND (BACKTRACK,Status) on arc labeled K_out
        SubS_Kout := closed
        W_out := K_out
      fi
end BTRACK


end Election(p)
```