Time and Message bounds for Election in Synchronous and Asynchronous Complete Networks *

Yehuda Afek[†] Eli Gafni[‡]

Abstract

This paper addresses the problem of distributively electing a leader in both synchronous and asynchronous complete networks. We present $O(n \log n)$ messages synchronous and asynchronous algorithms. The time complexity of the synchronous algorithm is $O(\log n)$, while that of the asynchronous algorithm is $O(n)$. In the synchronous case, we prove a lower bound of $\Omega(n \log n)$ on the message complexity. We also prove that any message-optimal synchronous algorithm requires $\Omega(\log n)$ time. In proving these bounds we do not restrict the type of operations performed by nodes. The bounds thus apply to general algorithms and not just to comparison based algorithms.

1 Introduction

In the *election* problem, a single node, called the leader, is to be selected from a set of nodes which initially differ only in their identifiers (*ids*), with no node being aware of any other id. An arbitrary subset of nodes wakes up spontaneously at arbitrary times and starts the

*A preliminary version of Section 3 was presented in the *Twenty-Second Annual Allerton Conference on Communication, Control, and Computing, Allerton, IL* October 3-5, 1984. A preliminary version of Section 4 was presented in the *Fourth Annual ACM Symposium on Principles of Distributed Computing,* Minaki, Ontario, Canada, August 1985

†Computer Science Dep. Tel-Aviv university, and AT&T Bell Laboratories Murray Hill, NJ 07974. This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-82-C-0064.

‡Computer Science Dep. Tel-Aviv university, and University of California, Los Angeles, Los Angeles, CA 07974. Supported by NSF Presidential Young Investigator Award under grant DCR84-51396 & matching funds from XEROX Co. under grant W881111.

algorithm by sending messages over the network. When the message exchange terminates, a leader is distinguished from all other nodes.

This paper specializes to the problem of electing a leader in a complete network. In such a network, each pair of nodes is connected by a bidirectional communication link. Before the algorithm starts, no node has any information on any of the other nodes. Hence, the incident links of a node, on which no message was sent or received, are indistinguishable. We consider both synchronous and asynchronous modes of communication.

When designing an algorithm, one has to take into consideration the class of inputs for the algorithm. The smaller the class, the more efficient an algorithm might be found since special features of the class may be exploited. Thus, for electing a leader in an arbitrary topology asynchronous networks, a $\Theta(m + n \log n)$ bound on the message complexity was proved,[Bur80, FL87, GHS83, Gal77, PKR82] where n and m are the total number of nodes and links in the network. $\Omega(m)$ is clearly a lower bound for an arbitrary network, since no algorithm may terminate before sending at least one message over each link, as an untraversed link could be the only link connecting two parts of the network, each holding a separate election. While the $\Omega(n \log n)$ part of the lower bound holds for networks as sparse as a ring [Bur80, FL87, FL85], Korach et. al. [KMZ84] noted that the $\Omega(m)$ lower bound *does not* hold for the smaller class of complete networks, where an election algorithm may terminate after one node has communicated with all its neighbors. Subsequently, they proved a lower bound of $\Omega(n \log n)$ messages for asynchronous complete networks, and presented an algorithm that requires $5n \log n + O(n)$ messages and $O(n \log n)$ time.

For synchronous networks, it was shown by Frederickson and Lynch [FL87, FL85] that one should distinguish between two types of algorithms: *general*, in which nodes may perform any computation on the values of their ids, and *comparison*, in which the values of ids can only be used for comparison with each other. They addressed the problem of election in a synchronous ring, for which they presented a general algorithm with $O(n)$ messages, and proved a lower bound of $\Omega(n \log n)$ messages for comparison algorithms, thus showing that general algorithms are strictly more powerful than comparison algorithms in a ring. This difference stems from the capability of general algorithms in synchronous networks to delay messages and processors as a function of the value of their id's.

In this paper we prove that the message complexity of any election algorithm, comparison or general, in a complete synchronous or asynchronous network is $\Theta(n \log n)$, thus proving that general algorithms are not more powerful than comparison algorithms for the problem of election in complete networks. The difference between synchronous rings and synchronous complete networks stems from the fact that in a ring all nodes can be distributively awakened with n messages, whereas in the complete network the awakening problem is as hard as the election problem, hence requiring $\Omega(n \log n)$ messages. Moreover, we prove that $\Omega(n \log n)$ is a lower bound on the message complexity of any algorithm whose execution requires the sending of messages, at least over the links of a connected spanning subnetwork. If all the nodes of a complete network could be awakened with n messages, then a general algorithm could take advantage of the synchronous mode of communication to elect a leader in a linear number of messages by using the principles suggested in [FL87, Gaf85, Vit84].

Consider the following, straightforward synchronous election algorithm in complete networks. Every initiator starts the algorithm by sending messages, containing its id, to all its neighbors. All the nodes then elect the highest id initiator as the leader. The time complexity of this algorithm is two time units, and its worst case message complexity is $\Theta(n^2)$. In Section 3.1 we slow down this simple algorithm and design an $O(\log n)$ time, message-optimal synchronous algorithm. This is done by carefully selecting a dynamic rate at which initiators send messages to their neighbors. Furthermore, by varying the dynamic rate at which initiators send messages to their neighbors, we are able to trade message complexity for a better time complexity. Hence, we present a continuum of $2\log_c n$ time, $2c \cdot n \cdot \log_c n$ messages, $2 \leq c \leq n$, synchronous algorithms, closing the gap between the trivial $O(1)$ time, $O(n^2)$ messages algorithm and the $O(\log n)$ time, $O(n \log n)$ messages algorithm.

When applying the dynamic rate technique in the asynchronous model, we obtain a $5 \cdot n \log n$ messages but still linear time algorithm. The asynchronous algorithm is an improvement over the algorithm of [KMZ84] which takes $O(n \log n)$ time and $5n \log n$ messages.

In an effort to reduce the message complexity of the asynchronous algorithm to $2n \log n$, we present a sequence of three asynchronous algorithms (A, B, and C). The first two algorithms present a tradeoff between time and message complexities. Algorithm A has $O(n \cdot \log n)$ time complexity and $2 \cdot n \cdot \log n$ message complexity. Algorithm B (which was also derived independently in [Hum84]) has $O(n)$ time complexity but $2.773 \cdot n \cdot \log n$ message complexity. Analyzing the communication and time complexities of the two algorithms, we derive a third algorithm, algorithm C, whose time complexity is $O(n)$ and communication complexity is $2n \log n$, an improvement on the $O(n \log n)$ time and $2n \log n$ message algorithm of Peterson [Pet84].

Following the $\Omega(n \log n)$ lower bound on the message complexity and the continuum of synchronous algorithms, we prove an $\Omega(\log n)$ lower bound on the time complexity of any message-optimal election algorithm in synchronous complete networks.

Finally, we prove a tight lower bound on the continuum of tradeoff between the message and time complexities of the synchronous algorithm. Specifically, we show that if an algorithm, whether comparison or general, elects a leader in at most $\frac{1}{2} \cdot \log_c n$ rounds, then its message complexity is at least $\frac{c-1}{2} n \cdot \log_c n$. Thus showing that on the one hand, any message-optimal algorithm has a $\Omega(\log n)$ time lower bound, and on the other hand, any $O(1)$ time algorithm has a $\Omega(n^2)$ message lower bound.

The models used in this paper are described in Section 2. In Section 3 we present the various algorithms. In Section 4 we present the lower bounds for the synchronous case thus proving the optimality of our algorithms. Throughout the paper, unless otherwise specified, we use log to denote log_2.

2 Models and assumptions

In this section we present the synchronous and asynchronous complete network models which follow the traditional message passing model [Bur80, LF81].

In a *complete network* of n nodes (processors), every node is connected by $n-1$ bidirectional communication links to all the other nodes. Nodes have unique *ids*, but no node knows the *id* of any other node. Each link incident to a node has a unique representation in that node. All messages received at a node are stamped with the identification of the link through which they arrived. By the number of its outgoing links every node knows n before the algorithm starts. However, *all the links incident to a given node on which no message was sent or received are indistinguishable to this node.*

The nodes of the network are initially asleep, but an arbitrary subset wake up at arbitrary times and start the distributed election algorithm. Each node in this subset is called *initiator*. Sleeping nodes may be awakened by receiving a message of the algorithm. When the message exchange terminates exactly one node emerges as the network leader.

Considered here are two modes of communication, *synchronous* and *asynchronous*. In the *synchronous* mode of communication, a global clock is connected to all the nodes in the network. The time interval between two consecutive pulses of the clock is a *round*. At the beginning of each round, each node decides, according to its state, what messages to send and on which links to send them. Each node then receives any messages sent to it in this round and uses all the received messages and its state to decide on its next state. Initiators start the distributed algorithm by entering an initial state and then waiting for the beginning of the next round.

In the *asynchronous* mode of communication, there is no global clock and messages incur arbitrary but finite delay. Messages from all input links are transferred into a central queue. The processor receiving the messages processes them one at a time in the order that they arrive at the central queue. The processing time of a message is negligible in comparison to its communication delay.

Throughout the paper, the *message complexity* (*communication cost*) of an algorithm is the worst case total number of messages sent during the algorithm, where a message contains at most $O(\log n)$ bits. The time complexity is the worst case total number of time units from the first to the last message transmission due to the algorithm, where a time unit in the synchronous case is one round. In the asynchronous case, and only for the purpose of time complexity analysis, one time unit is an assumed upper bound on message transmission delay. In arguing the time complexity, we shall allow a message to traverse a link in any fraction of the time unit.

3 Complete networks election algorithms

In this section we derive a distributed algorithm for election in synchronous complete networks and three algorithms for election in asynchronous complete networks. The message complexity of all the algorithms is $O(n \log n)$.

The underlying mechanism for all the algorithms is similar. Each algorithm is initiated by any subset of nodes, each of which is a *candidate* for leadership. Each candidate tries to capture all other nodes by sending messages on all the links incident to it. The algorithm terminates when a candidate has succeeded in capturing all its neighbors. This candidate becomes the leader. To guarantee that only one node is elected, all candidates but one are *killed*.

All candidates use a variable called *level* to estimate the number of nodes they have already captured. The level variable is used by candidates to contest each other. Captured nodes also have a level variable, which tracks the highest level candidate they have observed. That candidate is their *owner*. All level variables are initialized to 0.

A candidate that sees a node with a larger level than its own is eliminated from candidacy. However, if the candidate's level is larger or equal, the node's level is replaced by the candidate's level. The candidate may then claim the node and try to eliminate the previous owner of the node.

The algorithms differ mainly in three parts: (1) The way candidates determine their level, (2) The way candidates capture their neighbors, and (3) The rule candidates use to eliminate each other. In the synchronous algorithm, the level of a candidate is the number of rounds since it started the algorithm, and a candidate at level $2i$ tries to capture 2^i new nodes simultaneously (in one round). Unlike the synchronous algorithm, in the asynchronous algorithms candidates capture one neighbor at a time. In the first asynchronous algorithm, algorithm A, the level of a candidate is the number of other candidates it has killed. In algorithm B, the level of a candidate is the number of nodes it has already captured (following [Gal77].) Algorithm A achieves a better message complexity while algorithm B achieves a better time complexity. In algorithm C, candidates use a combination of the above two level functions to attain the time complexity of B and the message complexity of A.

To simplify the algorithms, every initiator node spawns two processes, the *candidate* process and the *ordinary* process. The two processes are connected to each other by a bidirectional logical link which behaves like a physical link. A node awakened by receiving a message of the algorithm spawns only an ordinary process. Candidate processes communicate only with ordinary processes and vice versa. Thus, the communication topology can be viewed as a complete bipartite graph, on one side the candidate processes and on the other side n ordinary processes. Henceforth, the term *candidate* will be applied interchangeably to both the process and its initiating node. All messages received by a node are tagged according to the type of their sending process. Messages received from candidate processes are forwarded to the ordinary process. Messages received from ordinary processes are forwarded to the candidate process.

3.1 The Synchronous Algorithm

In this section we present a $2 \cdot \log n$ rounds, $3n \log n$ messages synchronous algorithm. In the next section we will show that this algorithm is message optimal and is as fast as a message-optimal algorithm can be.

The algorithm is given in Figure 1. At every candidate the algorithm proceeds in levels. Every live candidate at level $2i$, $i \geq 0$, tries to capture 2^i new ordinary processes by sending them messages containing its level and id. If in level $2i+1$ the candidate receives acknowledgments from all the ordinary processes it tries to capture, then it proceeds as a candidate to the next level. Otherwise, the process (and hence the node owning it) is eliminated from candidacy.

The level of a candidate is incremented by one every round. Every ordinary process has a *level* and an *owner_id* variables which contain the level and id of the highest-level candidate the process has received a message from (level ties are resolved by selecting the highest id). In every round, every ordinary process first increases its level by one, to reflect the owner's actual level, and then inspects the newly received messages to update its level and owner_id if necessary. If an update occurred, the ordinary process acknowledges its new owner.

In the algorithm description, E is the set of edges incident to a candidate process. Every candidate maintains a list of edges, called *untraversed*, which it has not yet traversed in any direction.

3.1.1 Time and Message Complexities

Let p be the largest id of a candidate from the set of oldest candidates (i.e., whose level is the largest). We observe the following three facts:

Fact 1: *The level of every ordinary node strictly increases from one round to the next.*

Fact 2 : *At most $\frac{n}{2^{i-1}}$ candidates reach level $2i$, $1 \leq i \leq \log n$.*

Fact 3: $2 \log n$ *rounds after it has started the algorithm, candidate p has captured all the*

```
Candidate program:
    untraversed ← E
    level ← -1 ;
    Each round do:
        level ← level + 1 ;
        if level is even
        then
            if untraversed is empty
            then
                ELECTED, STOP
            else
                K ← Minimum ( 2^{level/2}, |untraversed|) ;
                Send (level, id) over K links from untraversed, and
                remove these links from untraversed ;
        else                                                  /* level is odd */
            Receive all acknowledgment type messages
            if received less than K acknowledgments
            then
                Stop                                /* Not a candidate any more */
    End each round.


Ordinary program:
    l* ← nil ;
    level ← -1 ;
    owner_id ← id ;
    Each round do:
        Send an acknowledgment over l* ;
        level ← level + 1 ;
        Receive all candidate messages (level,id) over link l;
        Let (level*, id*) be the lexicographically largest
        ( level, id) candidate message, and
        l* the link over which it arrived ;
        if (level*, id*) > (level, owner_id)
        then
            (level, owner_id ) ← (level*, id*) ;
        else
            l* ← nil ;
    End each round.
```

Figure 1: The Synchronous Algorithm.

nodes and is elected as the network leader.

Fact 1 follows immediately from the algorithm for ordinary node processes. Fact 3 holds because all the messages of p get acknowledged, and once a node has acknowledged p, it will not acknowledge any other message. Fact 2 follows from fact 1 and the observation that every ordinary node acknowledges at most one message in which the level is i, $0 \leq i \leq \log n$, i.e., the sets of 2^{i-1} nodes that are captured by each candidate that has reached level $2i$ are disjoint.

Following fact 3, the time complexity of the algorithm is $2 \log n$. Since every node sends at most one acknowledgment to a candidate in level $2i$, the total number of acknowledgments is $n \log n$, each of length $O(1)$ bits. Due to fact 2, the total number of candidate messages is $\sum_{i=1}^{\log n} \frac{n}{2^{i-1}} 2^i = 2n \log n$, each message containing $\log n + \log \log n$ bits. The total communication complexity is thus $3 \cdot n \log n$ messages.

A continuum of algorithms can be devised to close the gap between the trivial $O(1)$ time, $O(n^2)$ messages algorithm and the $O(\log n)$ time, $3n \log n$ messages algorithm. Each algorithm in the continuum is the same as the above, except that a candidate in level i tries to capture c^i neighbors, $2 \leq c \leq n$. The time complexity of the algorithm is $2 \log_c n$, and its message complexity is $2c \cdot n \cdot \log_c n$, thus proving that the lower bounds that will be presented in Section 4.3 (Theorem 2) are tight.

3.1.2 Asynchronizing the synchronous algorithm

To maintain the $O(n \log n)$ message complexity in the asynchronous communication mode, we are forced to increase the time complexity to $O(n)$. The increase in the time complexity seems unavoidable and it remains an open question whether a sublinear time, message-optimal asynchronous algorithm exists.

There are two basic differences between the asynchronous and synchronous modes of communication. First, in the asynchronous mode there is no global clock, and second, messages incur an arbitrary but finite delay. The arbitrary delay of messages (and not the absence of the clock) is the source of the increase in the time complexity of the algorithm. Essentially the synchronous algorithm would perform as well without a global clock, if all messages sent by a single node incur exactly the same delay.

To see that a straightforward application of the synchronous algorithm in the asynchronous model will not work consider the following situation: There are two competing candidates, C_1 and C_2, each had already successfully captured one node, and both proceed to capture the nodes v and u at the same time. A message of C_1 was the first to arrive at v which is then captured by C_1, while a message of C_2 was first to arrive at u. Following the rules of the synchronous algorithm, v positively acknowledges only C_1 and u positively acknowledges only C_2. Thus, both candidates are killed since none had all of its messages positively acknowledged, and the algorithm deadlocks. In the following sections we present algorithms which overcome this and similar problems in the asynchronous case.

3.2 Three Asynchronous Algorithms

Our aim in this subsection is to derive a $2 \cdot n \log n + O(n)$ messages, linear-time asynchronous algorithm. To this end we present a sequence of three asynchronous algorithms (A, B, and C), each devised to circumvent the problems of the previous, so that algorithm C achieves the desired complexity.

Unlike the synchronous algorithm, a candidate process in the asynchronous algorithms tries to capture all the other nodes by successfully traversing its incident links one at a time.

3.2.1 Algorithm A

Level: In this algorithm, the level of a candidate is the *total number of other candidates that it has killed.*

Capturing and Elimination Rule: To capture node v the level of a candidate must be strictly larger than that of v, in which case the candidate captures v without killing the previous owner of v. When two live candidates in the same level encounter each other, the higher id one kills the other and increases its level by one.

When candidate P arrives at node v which is currently owned by candidate Q, the following rule is used:

If $Level(P) < Level(v)$, P is killed.
If $Level(P) > Level(v)$, v is captured by P, and v gets P's level.
If $Level(P) = Level(v)$, P is sent to Q.

Upon arriving to Q:
If $(Level(P), id(P)) < (Level(Q), id(Q))$, P is killed.
If Q has already been killed, P is killed too.
If $(Level(P), id(P)) > (Level(Q), id(Q))$, **then** (1) Q is killed, (2) P increases its level by one, and (3) P captures v.

Details: To keep track of its owning candidate, every captured node has two link pointers, $father$ and $potential\_father$. The father pointer points to the link through which the node was most recently captured, and the potential_father pointer points to the link through which a candidate which tries to claim the node from its father, has arrived.

As in the synchronous algorithm every initiator spawns two independent processes, *candidate* and *ordinary*. The two processes are connected by a bidirectional logical link which behaves like a physical link.

A formal description of algorithm A is given in Figure 2. When a candidate arrives at node v whose level is the same as its own, and the id of v's father, Q, is smaller, it becomes v's potential_father. The potential_father is then sent to Q in an attempt to kill it. If another candidate at the same level with even higher id arrives at v before the potential_father returns from Q, then this other candidate is killed. If the potential_father survives at Q it first increments its level by one, then returns to v and captures it, and only then, returns to its initiating node. However, if the potential_father finds that Q is already killed, it eliminates itself as well (since Q is killed, there must exist a higher level candidate in the network).

Analysis:

Since at most half of the candidates at level k go up to level $k+1$, the maximum level achievable during the algorithm is $\log n$. Clearly, every time a node is recaptured its level is increased by at least one. Hence, the total number of capture messages possible is at most $n \cdot \log n$. Each capture uses 2 messages, which sums up to a total of $2 \cdot n \cdot \log n$ messages. The extra messages spent by candidates which go over father links to other candidates is at most $2 \cdot n$, since each such traversal results in the elimination of one live candidate. Thus, the message complexity of the algorithm is $2 \cdot n \cdot \log n \ + \ 2 \cdot n$ messages, each of length $\log n \ + \ \log\log n$ bits.

The time complexity of the algorithm is $O(n \cdot \log n)$ by the following scenario, in which $\frac{n}{2}$ of the nodes are captured serially $\log\left(\frac{n}{2}\right)$ times. The algorithm is started by node v_0 which captures $\frac{n}{2}$ nodes in level 0. Then, a new node, v_1, spontaneously starts the algorithm, kills v_0, increases its level to 1 and recaptures the same $\frac{n}{2}$ nodes. After v_1 has captured the $\frac{n}{2}$ nodes, two new nodes spontaneously start the algorithm, try to kill each other, and the one which survives, v_2, reaches level 1. Node v_2 then kills v_1 and recaptures the $\frac{n}{2}$ nodes at level 2. The scenario continues until the entire network has been captured by $v_{\log \frac{n}{2}}$ which is elected as a leader.

The basic reason for the high time complexity of the algorithm is that the level of a candidate is not a function of the number of nodes it has already captured. A candidate which spent a lot of work (and time) accumulating nodes might be killed by a candidate which did not spend nearly as much. In the next algorithm we will employ a linear level function, suggested by Gallager in a similar context [Gal77], thus reducing the time complexity to $O(n)$.

Initially:

$level \leftarrow size \leftarrow 0$; $owner\_id \leftarrow potential\_id \leftarrow 0$; /* $size$ used in algorithm C only*/
untraversed $\leftarrow E$; $father \leftarrow potential\_father \leftarrow nil$; killed $\leftarrow$ true ;

Candidate (id) :

while (untraversed $\neq \phi$) do;
  $l \leftarrow$ any(untraversed) ; send($level, id$) on l ;
R: receive($level', id'$) over l' ;
  if ($id' = id$)AND not killed **then** /*successful capturing.*/
    $level \leftarrow level'$; untraversed $\leftarrow$ untraversed - l ;
  else-if ($level', id' < level, id$) OR killed /* lexicographically*/
  then Discard the message, goto R ;
  else send($level', id'$) over l'; killed $\leftarrow$ true ; goto R ;
end while;
if not killed **then** announce(ELECTED);

Ordinary:

$level \leftarrow$ -1 ;
while (not terminated) do;
  receive($level', id'$) over l' ;
  case $level'$ of :
  (1) $level' < level$: Discard message ;
  (2) $level' > level$: /* Replace the father */
    $father \leftarrow l'$; $level \leftarrow level'$; $owner\_id \leftarrow id'$;
    $potential\_id \leftarrow 0$; $potential\_father \leftarrow$ nil ;
    send($level', id'$) over the $father$ link ;
  (3) $level' = level$:
    if ($id' < owner\_id$) **then** Discard message;
    else-if ($id' = potential\_id$) **then**
      $father \leftarrow potential\_father$;
      $level' \leftarrow level' + 1$;
      owner_id $\leftarrow id'$; $potential\_id \leftarrow 0$;
      $potential\_father \leftarrow$ nil ;
      send($level'$,id') over the $father$ link ;
    else-if there is already a $potential\_father$ **then** Discard message ;
    else /* there is no $potential\_father$ */
      $potential\_id \leftarrow id'$; $potential\_father \leftarrow e'$;
      send($level', id'$) over the $father$ link ;
  end case ;
end while ;

Figure 2: Algorithm A.

3.2.2 Algorithm B

Level: In this algorithm, the level of a candidate is the *number of nodes it has already captured.*

Capturing and Elimination Rule: To capture node v; (1) the (level, id) of a candidate must be lexicographically larger than the (level, id) of the previous owner of v, and (2) the previous owner must be killed.

When candidate P arrives at node v which is currently owned by candidate Q, the following rule is used:

if $(Level(P), id(P)) < (Level(v), id(Q))$, P is killed.
if $(Level(P), id(P)) > (Level(v), id(Q))$, (1) v gets P's level, and (2) P is sent to Q.

When P arrives at Q:

if $(Level(P), id(P)) < (Level(Q), id(Q))$, P is killed.
if Q has been killed already **then** P returns to v and retries to capture it.
if $(Level(P), id(P)) > (Level(Q), id(Q))$, **then** (1) Q is killed, and (2) P returns to v and retries to capture it.

Upon returning to its initiating node from a successful capturing, P increases its level by one.

Details: A candidate, C, that arrives at an already captured node v whose level is smaller than its own, replaces v's level with its own and becomes v's $potential\_father$. C is then sent to the father candidate of v. If C survives at v's father, and meanwhile no other candidate replaced C as the potential_father of v, then C becomes $v's$ father. If v has not yet been captured, the potential_father automatically becomes the father of v. A formal description of algorithm B is given in Figure 3.

Analysis: The algorithm is deadlock-free since candidates never wait for each other, and the (*level*, *id*) pair is lexicographically increasing along any chain of candidates which kill each other.

The time complexity of the algorithm is $O(n)$ since candidates never wait for each other and a candidate which has done more work is never killed by a candidate which has done less work. Thus, each killed candidate spent, in the worst case, less time than the one killing it.

```
Initially
level ← owner_id ← 0 ;
untraversed ← E ; father ← nil ; killed ← false ;

Candidate ( id ) :
while ( untraversed ≠ φ ) do;
      l ← any( untraversed ) ;
      send(level, id) on l ;
R:    receive(level', id') over l' ;
      if (id' = id)AND not killed then                      /* successful capturing.*/
                  level ← level + 1 ;
                  untraversed ← untraversed - l ;
      else                      /*another candidate tries to eliminate candidate id */
                  if (level', id' < level, id)                  /* lexicographically*/
                  then Discard the message, goto R ;
                  else                                  /* Candidate id is eliminated */
                      send(level', id') over l';
                      killed ← true ;
                      goto R ;
end while;
if not killed announce(ELECTED);

Ordinary:
for_ever do;
      receive(level', id') over l' ;
      case level', id' of :
      (1) level' , id' < level, owner_id:
                Discard message ;
      (2) level', id' > level, owner_id :
                potential_father ← l' ;
                level ← level' ;
                owner_id ← id' ;
                if father = nil then father ← potential_father ;
                send(level', id') over the father link ;
      (3) level', id' = level, owner_id:
                father ← potential_father ;
                send(level', id') over the father link ;
      end case ;
end for_ever ;
```

Figure 3: Algorithm B.

To prove that the communication complexity of the algorithm is $O(n \log n)$ we use a Lemma which was introduced in [Gal77].

Lemma 1 *For any given k, the number of candidates that own $\frac{n}{k}$ or more nodes is at most k.*

Proof: Let C_1 and C_2 be any two candidates which owned $\frac{n}{k}$ nodes at some point of time. We shall show that each of C_1 and C_2 must have owned disjoint sets of at least $\frac{n}{k}$ nodes each. If they never tried to claim a node from each other, we are done. The first time that C_1 (w.l.o.g.) tries to claim a node, say v, from C_2, either one of them dies, or C_2 has already been killed. If C_1, w.l.o.g., caused the death of C_2 then clearly it must have owned at least $\frac{n}{k}$ nodes disjoint from C_2, at the time of killing. If C_2 is already dead, C_1 must still own at least $\frac{n}{k}$ nodes in order to claim v to itself. ∎

Corollary 2 *The largest candidate to be killed by another candidate owns at most $\frac{n}{2}$ nodes, the next largest owns at most $\frac{n}{3}$ nodes, etc.*

Lemma 3 *The message complexity of algorithm B is $4 \cdot n \cdot\ ln\ n\ (=\ 2.773 \cdot n \cdot \log_2 n)$ messages.*

Proof: Since in capturing one node a candidate makes at most 4 hops, a candidate which owned k nodes incurs at most $4 \cdot k$ messages. By Corollary 2, the total cost is then bounded by $4 \cdot n \cdot \sum_{i=1}^{n} \frac{1}{i}$ messages. Note that each message of the algorithm contains at most $2 \cdot \log n$ bits. ∎

The number of candidates at a particular level was constrained by the disjointness property. Hence, a candidate which captures many nodes from another candidate, tries to eliminate that other candidate as many times as the number of nodes it captures from it. This gives rise to the factor 4 in the message complexity.

In algorithm A the amount of work a candidate spent was not factorized into the level function and thus the time complexity was $O(n \log n)$. Although in algorithm B the time complexity was reduced to $O(n)$, it has the problem that candidates could be re-eliminated many times thus increasing the message complexity. In the next algorithm we eliminate both problems by employing a level function which is a combination of the two.

3.2.3 Algorithm C

Here we make two modifications to algorithm A in order to achieve a linear-time complexity with no increase in the communication cost. First, we incorporate an estimate of the amount of work spent by each candidate into the level function of algorithm A. Second, we enable candidates with a high level ($> \log n$) to capture many nodes in parallel (in one time unit). We start describing the algorithm with the first modification. The second modification will be introduced during the performance analysis.

Level: In this algorithm the level of a candidate is increased according to two rules. The rule of algorithm A, and in addition after each capture the candidate increases its level to be at least log(*total number of nodes captured*), i.e., after returning from a successful capture the level is set to max(log *(# nodes captured), present level).*

Capturing and Elimination Rule: Same as in algorithm A.

Details:

The formal description of the algorithm is similar to that of algorithm A. A variable, called "*size*", is used to count the number of nodes captured by a candidate. The only change is to replace the first "then" clause, within the "while" loop of a candidate program to:

then

$size \leftarrow size + 1$;
$level \leftarrow max(level', \log size)$;
untraversed $\leftarrow$ untraversed - 1 ;

Analysis: To analyze its performances we will first show that:

Lemma 4 *The maximum level reachable during any execution of algorithm C is* $\log n + \log\log n + 1$.

Proof: Let N_i be the total number of candidates that reach level i during the execution of the algorithm. Consider the maximum number of candidates which could possibly pass from level $i-1$ to level i. There are two ways in which a candidate can go from level $i-1$ to level i. First, by capturing 2^{i-1} nodes at level $i-1$ for $i \leq \log n$, and second, by killing another candidate which is at level $i-1$. We note that N_i is maximized if as many candidates as possible pass from level $i-1$ to level i by capturing other nodes (i.e., $\frac{n}{2^{i-1}}$ candidates) and the rest of the candidates (i.e., $N_{i-1} - \frac{n}{2^{i-1}}$) kill each other in pairs. Hence,

$$N_i \leq \frac{(N_{i-1} - \frac{n}{2^{i-1}})}{2} + \frac{n}{2^{i-1}} \tag{1}$$

Solving (1) for N_i we get:

$$N_i \leq \frac{n}{2^i} \cdot (i+1) \tag{2}$$

Substituting $N_i = 1$ in (2) and solving for i gives us the maximum level, which is $\log n + \log\log n + 1$. ∎

Using the same argument as in algorithm A we find that the message complexity of algorithm C is $2 \cdot n \cdot (\log n + \log\log n + 2)$ messages, each of length $\log n + \log(\log n + \log\log n)$ bits.

With the above modification the time complexity of algorithm A is reduced to $O(n \cdot \log\log n)$. This is due to the fact that if the highest level in the network is i, it must increase within time $min(2^i, n)$. In order to further reduce the time complexity to $O(n)$, processes at levels higher than $\log n$ will try to capture $n/\log n$ nodes in parallel. Thus a candidate which has reached level $\log n$ will send messages over $n/\log n$ untraversed links incident to it. Each of these messages carries the (level, id) of the candidate. When a message arrives at an adjacent node the node compares its level to that of the message. If the message level is higher, the node replaces its (level, id) with the message, thus making the candidate the new father of the node. The node then sends the candidate an acknowledgment of successful capture. If the message level is smaller, it returns no message. Finally, if the message level is the same as that of the node but the message id is higher, a notification to that effect is sent back to the candidate.

The candidate waits for all the $n/\log n$ acknowledgments. If all the acknowledgments indicate a successful capture, the candidate proceeds to the next $n/\log n$ untraversed incident links. If, on the other hand, some of the acknowledgments indicate that they have encountered the same level, one of the links is arbitrarily chosen and a process that behaves as in algorithm A is sent along that link. If the process returns, the candidate increases its level and proceeds to the next $n/\log n$ untraversed links (links on which no successful capture was reported are not considered traversed).

To analyze the algorithm with this modification we make two observations: First, the maximum attainable level in the algorithm is still bounded by $\log n + \log\log n + 1$. Second, by substituting $i = \log n$ in equation (2), we find that the maximum number of candidates which reach level $\log n$ is $\log n$.

The last modification has increased the communication complexity of the algorithm by at most $O(n)$ messages. Each node is still captured at most $\log n + \log\log n + 1$ times, however the death of a candidate at level greater than $\log n$ might be associated with at most $2 \cdot n/\log n$ messages. Since there are at most $\log n$ such candidates the increase due to killings is bounded by $O(n)$.

To show that the time complexity of the algorithm is $O(n)$ we arrange the candidates in a rooted tree. Each level of the tree corresponds to the candidates that have reached that level in the algorithm, i.e., the nodes at level i in the tree correspond to the candidates that have reached level i in the algorithm. The parent of a candidate at level i in the tree is either the candidate that caused the death of the given candidate or, the same candidate at the next level. The time delay of the algorithm is the sum of the delays incurred by candidates along the path from the first initiator (candidate) to wake up, at level 0, to the root.

To evaluate this time delay we note that no candidate that either survives or is killed at level i spends more than 2^{i-1} time units in level i, $i \leq \log n$. In level i, $\log n < i \leq \log n + \log\log n$ no candidate spends more than $\log n$ time units since it captures nodes at a rate of $n/\log n$ per time unit. Hence, the total time delay of the algorithm is bounded by $\sum_{i=1}^{\log n} 2^i \; + \; \log n \cdot \log\log n \; = \; n + \log n \cdot \log\log n$. Note that we scale a time unit to be the maximum delay it takes to capture one node, which is a constant.

In the above calculation we did not include the actual time it takes candidates to kill each other. Since there are at most n candidates and no candidate tries to kill a dead one (unlike algorithm B), this delay is also bounded by $O(n)$.

4 Lower bounds for election in complete networks

Two algorithms for election in synchronous complete networks were discussed in the previous sections. The first is a $O(n^2)$ messages $O(1)$ time algorithm and the second is a $O(n \log n)$ messages $O(\log n)$ time algorithm. The two algorithms raise two questions: (1) Is $\Omega(n \log n)$ also the lower bound on the message complexity of election in synchronous complete networks, and (2) If $\Omega(n \log n)$ is the message complexity lower bound, then how fast can a message-optimal algorithm be, i.e., is there a $O(n \log n)$ messages $O(1)$ time algorithm, or is $\Omega(\log n)$ the lower bound on the time complexity of any message-optimal synchronous algorithm.

In this section we answer these questions by proving first, that $\Omega(n \log n)$ is a lower bound on the worst case message complexity of a synchronous algorithm, and second, by proving that $\Omega(\log n)$ is a lower bound on the time complexity of any message-optimal synchronous algorithm. In proving these bounds we do not restrict the type of operations performed by the nodes. The bounds thus apply to general algorithms and not just to comparison based algorithms. This proves that the synchronous algorithm of Section 2 is optimal.

Furthermore, in Section 2 we have presented a continuum of, $\frac{2c}{\log c} \cdot n \log n$ messages $2 \cdot \log_c n$ time, synchronous algorithms where, $c = 2, \ldots, n$. In Theorem 2 each algorithm in the continuum is shown to be optimal by proving that if an algorithm (whether comparison or general) elects a leader in at most $\frac{1}{2} \cdot \log_c n$ rounds, then its message complexity is at least $\frac{c-1}{2 \cdot \log c} \cdot n \log n$.

To show the lower bounds, a scenario in which any synchronous (and hence also asynchronous) algorithm transmits at least $\frac{n}{2} \cdot \log n$ messages, is constructed using an adversary argument. A similar argument is then used to show that the delay of any message-optimal algorithm is at least $\Omega(\log n)$ rounds.

4.1 Definitions and Assumptions

Consider an arbitrary election algorithm on the synchronous model defined above. An *event* is the sending of a message over a previously unused link (two messages sent in the same round in opposite directions over a previously unused link are considered two separate events). With each event we associate a link (s, d), where s is the source node and d is the destination node of the corresponding message. With each round i of the algorithm we associate a set of events, R_i. A sequence $E = (R_0, R_1, \ldots)$ is called an *execution*. An $execution-prefix$ E_j is a prefix, $(R_0, R_1, \ldots, R_j)$, of an execution E. With each run of the algorithm we associate an execution, called a $legal - execution$, that includes all events which occurred in the run, arranged in order of the corresponding rounds. Henceforth, any mention of a message refers to an event.

With each execution-prefix E_j we associate a graph G_j, whose nodes are the nodes of the network and whose links are the links associated with the events in E_j. A *cluster* in an execution-prefix E_j is any maximally connected component of G_j. The "*degree*" of a node v in an execution-prefix E_j is the degree of v in G_j. The $potential - degree$ of node v in an execution-prefix E_j is the degree of v in E_j plus the number of times that v is a source node of an event in R_{j+1}. The $potential - degree$ of a set of nodes is the maximum potential-degree among its nodes.

For the purpose of proving the lower bounds we introduce a slightly different model than the standard synchronous model, called the $stopping - model$. The stopping model allows us to withhold the clock pulse, at the beginning of round j from any set of clusters, C, in E_{j-1}, given that no node in C is expected to receive a message in round j from a node not in C. The nodes in C are then said to be $frozen$ in round j. Therefore, a frozen node in a round neither sends nor receives any message in that round; nor does it change its state. The stopping-model will be used to prevent large differences in the clusters' growth rates.

The stopping model essentially allows us to select any cluster in an execution-prefix E_j and send it back in time, i.e., the nodes of the cluster are awakened some number of rounds later than they were in E_j.

A $stopping - execution$ is an execution which corresponds to a run of the algorithm in the stopping-model. A stopping-execution is called a k $stopping - execution$ if the cumulative number of pulses withheld over all frozen sets of clusters throughout the run is k. Obviously, a 0-stopping-execution is a legal-execution.

Lemma 5 *For any k stopping-execution E, there exists a $k-1$ stopping-execution E' which contains exactly the same events as E does.*

Proof: Let l be the minimum index of a round in which any set of clusters is frozen, and let C be a set of clusters which is frozen in l. An execution-prefix E' which satisfies the lemma can be obtained from E by shifting all events which occurred before round l and involve nodes in C, one round forward. This affects neither any event in later rounds nor any event which involves nodes not in C. Because, neither in E nor in E' there is an event connecting a node in C with a node not in C in any round R_j, $j \leq l$, and because R_{l+1} in E is identical to R_{l+1} in E'. Notice that in E' the nodes in C are awakened one round later than in E. ∎

Corollary 6 *For any stopping-execution there exists a legal-execution which contains the same events.*

In the next two sections we will prove the lower bounds on the stopping model. Using Corollary 6, these bounds apply also to the non stopping model. In our proofs we do not restrict the type of operations performed by the nodes, hence proving the bounds for general algorithms.

4.2 A Lower Bound on Message Complexity

At the end of any election algorithm all nodes know who the leader is, hence any such algorithm has to send messages along the links of a spanning subnetwork. In other words, by the end of the algorithm the whole network is contained in one cluster. Thus, no cluster in the algorithm can defer indefinitely the sending of messages to nodes not in the cluster, as the rest of the network might not wake up spontaneously.

In the following proof of the lower bound we will use an adversary argument to construct a stopping-execution (and thus a legal-execution) which contains at least $\frac{1}{2}n\log n$ events. In the beginning of each round, the adversary determines, first which clusters to freeze, and then determines which links to use for the new events generated by the unfrozen nodes. The first feature is used to delay the formation of larger clusters until later rounds in the run, thus avoiding large differences in the clusters' growth rates; the second feature is used to send as many messages as possible within one cluster, and thus again minimizing the cluster's growth rate. The second feature is possible since links incident to a given node on which *no* message was sent or received are indistinguishable to this node.

In the proof of Theorem 1 a $\frac{n}{2} \cdot \log n$ events execution-prefix is constructed formally. We will first demonstrate the construction through an example.

<table>
<tr><td>node id
Phase</td><td>1
•</td><td>2
•</td><td>3
•</td><td>4
•</td><td>5
•</td><td>6
•</td><td>7
•</td><td>8
•</td><td>9
•</td><td>10
•</td><td>11
•</td><td>12
•</td><td>13
•</td><td>14
•</td><td>15
•</td><td>16
•</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>2</td><td>1</td><td>5</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>9</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td colspan="2">2</td><td colspan="2">2</td><td colspan="2">5</td><td colspan="2">2</td><td colspan="2">2</td><td colspan="2">4</td><td colspan="2">9</td><td colspan="2">2</td></tr>
<tr><td>2</td><td colspan="4">4</td><td colspan="4">5</td><td colspan="4">4</td><td colspan="4">9</td></tr>
<tr><td>3</td><td colspan="8">8</td><td colspan="8">9</td></tr>
</table>

Table 1: Potential-degrees of nodes in the example.

Example 1 *Assume n is a power of 2. A stopping execution-prefix with at least $\frac{n}{2} \cdot \log n$ events is constructed in $\log n$ phases, each lasts few rounds. In phase j, $j = 0, \ldots, \log n$, the nodes are partitioned into sets of size 2^j, such that every subset contains two subsets of phase $j-1$. The adversary continues the execution (does not freeze) of any subset in phase j, whose potential-degree is less than $2^j - 1$, all other subsets are frozen. The destination of events generated by nodes whose potential-degree is less than 2^j, is chosen by the adversary to be within the same subset. Thus, no cluster with more than 2^j nodes is created in phase j and every cluster is contained in one subset. Each phase is ended when all the nodes are frozen, i.e., when in every subset there is a nodes which tries to send a message to a node not in the subset.*

Table 1 gives the potential degrees of the subsets by the end of each phase. In phase 0 of the example all nodes are awakened. Nodes 1, 2, 3, 5, 7,..., 13, 15, 16 have potential-degree 1, node 4 2, node 6 5 and node 14 9. If any of these nodes started with potential-degree 0, all the nodes with potential-degree greater than 0 are frozen until eventually all will have potential-degree at least 1. Thus in phase 0 no event was generated. In phase 1, node 3 is frozen because node 4 has potential-degree 2. Similarly node 5 because of node 6, and 13 because of 14. By the end of the first phase the constructed execution-prefix, E_{i_1}, contains the following events: $\{(1,2)(2,1)(7,8)(8,7)(9,10)\ (10,9)(11,12)(12,11)(15,16)(16,15)\}$. By the end of the second phase, E_{i_2} contains, at least, the following additional events: $\{(3,4)\ (4,3)(4,2)\ (2,3)\}$. E_{i_3}, by the end of the third phase, contains, at least, the following additional events: $\{(4,1)(4,5)(6,5)(6,7)(6,8)(6,4)(6,3)(8,5)\}$. In the first round of the fourth phase the following events will be added: $\{(6,2)(6,1)(6,9)(11,10)(11,9)(11,13)(13,14)(14,9)\ (14,10)(14,11)\ (14,12)(14,13)(14,15)\ (14,16)(14,8)(14,7)(15,14)\}$. After these events the whole network is contained in one cluster and a good election algorithm could terminate without generating any more events.

Clearly, by the end of phase j there are at least $\frac{n}{2^j}$ nodes with potential-degree 2^j. Thus every phase eventually contributes at least $\frac{n}{2}$ new events, summing up to $\frac{n}{2} \cdot \log n$ events. If n is not a power of 2, then $\frac{n}{2} < 2^m < n < 2^{m+1}$ and the scenario will be constructed over 2^m nodes, the rest of the nodes will be awakened only by the end of the last phase. In the next theorem we prove the lower bound formally.

Theorem 1 *A stopping-execution of an election algorithm in a synchronous complete network of n nodes contains at least $\frac{n}{2} \cdot \log n$ events, in the worst case.*

Corollary 7 *The message complexity of any election algorithm in a synchronous complete network of n nodes is at least $\frac{n}{2} \cdot \log n$.*

Proof of Theorem 1: Assume w.l.o.g. that $n = 2^q$. We define a sequence of partitions $(P_0, ..., P_q)$ of the nodes such that each subset in partition P_0 contains one node, and each subset in P_j contains two subsets from P_{j-1}, $1 \leq j < q$. Hence, each subset in P_j contains 2^j nodes.

We construct, in q phases, a sequence of stopping-execution-prefixes $(E_{i_0}, ..., E_{i_q})$, $i_0 = 0$, each being a prefix of the next. E_{i_0} is an empty execution-prefix in which all nodes have been awakened and the potential-degree of each node is at least 1. This is done by withholding the clock pulse from any node whose potential-degree is at least 1 until there is no node with potential-degree 0. Inductively we assume that: (1) Any cluster in E_{i_j} is contained within one subset in P_j. and (2) The potential-degree, in E_{i_j} of every subset in P_j is at least 2^j. Obviously, E_{i_0} satisfies these assumptions.

Assuming that $E_{i_{j-1}}$ has been constructed, we describe how the adversary constructs E_{i_j}, $j = 1, ..., q-1$. In each round of phase j, we freeze all the subsets in P_j whose potential-degree $\geq 2^j$. When all subsets are frozen, phase j is completed. The source and destination nodes of any message sent in this phase are both in the same subset in P_j. This is always possible since every node that has a potential-degree $\geq 2^j$ is frozen. Clearly, E_{i_j} satisfies the inductive assumptions. In the $q-th$ phase no freezing takes place. After that phase, the network is contained in one cluster and the algorithm is assumed to produce no more events.

Clearly, there are at least $n/2^j$ nodes whose degree at the end of the algorithm is at least 2^j, for $j = 0, ..., q-1$. Thus, the total number of events is at least $\frac{n}{2} \cdot \log n$. ∎

Given that the message complexity of any election algorithm on a synchronous complete network is $\Omega(n \log n)$, the question arises how fast can a message-optimal algorithm be. In the next section we prove that the time complexity of any message-optimal algorithm is $\Omega(\log n)$.

4.3 A Lower Bound on Time Complexity

In this section we will extend the techniques of the previous section to prove that the shorter the length of the execution the larger the lower bound on the number of events it must contain.

Theorem 2 *Any stopping-execution of an election algorithm in a synchronous complete network of n nodes which terminates in less than $\frac{1}{2} \cdot \log_c n$ rounds, contains at least $\frac{c-1}{2 \cdot \log c} \cdot n \log n$ events.*

Corollary 8 *The time complexity of any message-optimal election algorithm in a synchronous complete network of n nodes is $\Omega(\log n)$ rounds.*

Proof of Theorem 2: Consider an election algorithm whose time complexity is at most $\frac{1}{2} \cdot \log_c n$. Assume w.l.o.g. that $n = c^q$. A construction similar to the proof of Theorem 1 will be used here. We construct, in q phases, a sequence of stopping-execution-prefixes $(E_{i_0}, ..., E_{i_q})$, $i_0 = 0$, each being a prefix of the next, and a sequence of partitions $(P_0, ..., P_q)$, the subset of each partition containing c subsets of the previous. Each subset of P_0 contains one node, thus each subset of P_j contains c^j nodes. E_{i_0} is an empty execution-prefix in which all nodes are awakened spontaneously. Inductively we assume that: (1) Any cluster in E_{i_j} is contained within one subset of P_j, and (2) The potential-degree in E_{i_j} of every subset in P_j is at least c^j. Obviously, E_{i_0} and P_0 satisfy these assumptions.

Assuming that $E_{i_{j-1}}$ has been constructed, the adversary constructs E_{i_j} by first constructing partition P_j, and then E_{i_j}. Let $(S_1, ..., S_k)$, $k = n/c^{j-1}$ be the subsets of P_{j-1} indexed in nondecreasing order of their potential-degrees in $E_{i_{j-1}}$. Then the $i-th$ subset of P_j is defined as the union of $S_{(i-1)c+1}, ..., S_{ic}$ $i = 1, ..., n/c^j$. This implies that with the exception of at most one subset in P_j, called the *boundary* subset, each of the subsets in P_j consists of subsets in P_{j-1} all of which either have potential-degree above or equal to c^j, or below it.

In each round of phase j, $j = 1, ..., q-1$, we freeze all the subsets in P_j whose potential-degree $\geq c^j$. When all subsets are frozen phase j is complete. The destinations for messages to be sent by node v are selected from the subsets which included v in partitions $P_0, ..., P_j$, in that order of priority. This is always possible since every node that has a potential-degree $\geq c^j$ is frozen. Clearly, E_{i_j} and P_j satisfy the inductive assumptions. After the $q-th$ phase, the network is contained in one cluster and the algorithm is assumed to produce no more events.

We now show that every node is the destination of at least $\frac{1}{2} \cdot (c-1) \log_c n$ events in E_{i_q}, hence Theorem 1.

As the time complexity is at most $\frac{q}{2}$, every node must have been frozen in all the rounds of at least $\frac{q}{2}$ phases. Otherwise, the legal-execution corresponding to E_{i_q} would contain more than $\frac{q}{2}$ rounds, contradicting the assumption on the time complexity. If node v is frozen in all the rounds of phase j, and it is not in a boundary subset, it will later receive one message from every subset of P_{j-1} which does not contain v and is with v in a subset of P_j Thus, for each phase that v is frozen in all its rounds, v is the destination of $c-1$ events. The total number of events in E_{i_q} is thus at least $(c-1) \cdot (\frac{q}{2}) \cdot n \; - \; U$, where U is the number of events that should be discounted due to the boundary subsets. Due to the boundary subset in phase j at most $c^{j-1} \cdot (c-1)$ events should be discounted. Hence, $U =$ at most $n \cdot c$ and the total number of events is thus at least, $\frac{c-1}{2 \cdot \log c} \cdot n \log n - n \cdot c$. ∎

Example 2 *In table 2 we show how E_{i_2} is constructed from E_{i_1} assuming $n = 27$ and $c = 3$. Subsets $\{4, 5, 6\}$, $\{22, 23, 24\}$, and $\{16, 17, 18\}$ are frozen during phase 2, because node 5 has potential degree 9, node 23 10, and node 17 20. These subsets will become active again in phase 3, and then each node in them will be the destination of at least 2 new events, e.g., nodes 4, 5, and 6 each will receive a message from node 23 and 17. Nodes 22, 23, and 24 each will receive a message from node 5 and 17, and nodes 16, 17, and 18 will receive a message from node 5 and 23.*

phase	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15	16 17 18	19 20 21	22 23 24	25 26 27
1	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •
	5	9	5	4	4	20	4	10	4
phase	10 11 12	13 14 15	19 20 21	25 26 27	1 2 3	7 8 9	4 5 6	22 23 24	16 17 18
2	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •
	4	4	4	4	5	5	9	10	20
	ACTIVE			ACTIVE			FROZEN		
	19			18			20		

Table 2: Potential degrees in the time lower bound proof.

5 Conclusions

The effect of synchronous and asynchronous communication on the problem of distributively electing a leader in a complete network was examined. On the one hand, it was proved that the message complexity is not affected by the choice of the communication mode. In both modes of communication, the message complexity was shown to be $\Theta(n \log n)$. With synchronous communication, the time complexity of message-optimal algorithms was proved to be $\Theta(\log n)$, whereas with asynchronous communication, only an $O(n)$ upper bound on the time complexity was obtained. The lower bound on the time complexity in asynchronous communication remains an open question and is the subject of the following conjecture:

Conjecture: The time complexity of any message-optimal asynchronous election algorithm on a complete network is $\Omega(n)$.

The implication of the conjecture is that synchronous communication is faster by a factor of $n/\log n$ than asynchronous communication. An analogous result was obtained in [AFL83], where a particular synchronous system of parallel processors was proved to be faster by a factor of $\log n$ than the corresponding asynchronous system.

Three asynchronous election algorithms (A, B, and C) were presented. The simplicity of the complete network topology, and hence, of termination detection, enabled us to concentrate on the synchronization among contending candidates. With each of the three algorithms we can associate an analogous algorithm for arbitrary topology networks, which uses the corresponding method to synchronize different initiations of the algorithm but a different method to traverse the network (i.e., to detect termination). The analogy to algorithm B was given in [Gal77]. In [GHS83] the same level function as in algorithm A was used, however, there candidates merge their "territories" (rather than kill each other) when they meet. In [Gaf85] the time complexity of [GHS83] is improved by replacing its level function with that of algorithm C. Awerbuch [Awe87] used the ideas of algorithm C and [Gaf85] to further improve the time complexity of [GHS83] to $O(n)$. Each of the methods can be applied to other classes of networks. For example, applications of method A in different classes of topologies are discussed in [KKM85]. As in algorithm A, the time complexity of the algorithms in [KKM85] is $O(n \log n)$, whereas similar applications, but of methods B or C, improve the time complexity of [KKM85] while maintaining the message optimality.

References

[AFL83] E. Arjomandi, M. J. Fischer, and N. A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *JACM*, 30(3):449–456, July 1983.

[Awe87] B. Awerbuch. Linear time distributed algorithms for minimum spanning trees, leader election, counting and related problems. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 230–240, May 1987.

[Bur80] J. E. Burns. A formal model for message passing systems. Technical report, Indiana Univ., May 1980.

[FL85] G. N. Frederickson and N. A. Lynch. A general lower bound for electing a leader in a ring. Technical report, Purdue University, March 1985. CSD-TR-512.

[FL87] G. N. Frederickson and N. A. Lynch. Electing a leader in a synchronous ring. *Journal of The ACM*, 34(1):98–115, January 1987.

[Gaf85] E. Gafni. Improvements in the time complexity of two message-optimal election algorithms. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, August 1985.

[Gal77] R. G. Gallager. Finding a leader in a network with O(e) + O(n log n) messages. Unpublished Note, 1977.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77, January 1983.

[Hum84] P. A. Humblet. Selecting a leader in a clique in O(n log n) messages. In *Proc. of 23rd Conference on Decision and Control*, pages 1139–1140, December 1984.

[KKM85] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, August 1985.

[KMZ84] E. Korach, S. Moran, and S. Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, pages 199–207, August 1984.

[LF81] N. A. Lynch and M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[Pet84] G. L. Peterson. Efficient algorithms for elections in meshes and complete networks. Technical report, Dep. of Computer Science, Univ. of Rochester, August 1984.

[PKR82] J. Pachl, E. Korach, and D. Rotem. A technique for proving lower bounds for distributed maximum-finding algorithms. In *Proc. of the 14th Ann. ACM Symp. on Theory of Computing*, pages 378–382, 1982.

[Vit84] P. Vitanyi. Distributed election in an archimedean ring of processors. In *Proc. of the 16th ACM Symp. on Theory of Computing*, pages 542–547, 1984.