A 1.44\ldots N\log N ALGORITHM
FOR DISTRIBUTED LEADER-FINDING
IN BIDIRECTIONAL RINGS OF Processors

by

S. Moran, M. Shalom and S. Zaks

Technical Report #389
November 1985
ABSTRACT

This work presents an algorithm for finding a leader in a bidirectional ring without a common sense of orientation, using at most $1.44n \log n + O(n)$ messages. This improves the previous upper bound of $1.89n \log n + O(n)$ messages.

1. INTRODUCTION

The problem of finding a leader is widely studied in the literature of distributed networks. In this environment, the processors in the network have to identify one of them as a leader, in order to resolve a certain conflict that some (may be all) of them are aware of a certain situation they have detected. This problem is one of the very basic ones that distinguishes distributed networks from the classical situation of one processor controlling a certain function. A better understanding of the problems arising in this situation is crucial in our study of these networks.

Consider $n$ processors $P_0, P_1, \ldots, P_{n-1}$ arranged on a ring such that $P_i$ is adjacent to $P_{i+1} \pmod{n}$ and to $P_{i-1} \pmod{n}$. The processors can communicate only by sending messages along the edges of the ring. Every processor $P_i$ has an identity $id_i$ (chosen from some infinite totally ordered set) which is known only to itself. The aim is to design an algorithm such that when executed at every processor, exactly one processor will be selected as the leader. The algorithm may be started by any subset of the processors.

It is assumed that every message will reach its destination after a finite but unbounded delay, that each communication line preserves the order of the messages sent on it and that all the identities are distinct.

An algorithm is a program that contains three types of executable statements: local computations, message send and message receive statements. An execution is a sequence of events where each event is one of the above three statements.

The complexity measure for such an algorithm is the maximum number of messages sent during any execution of the algorithm. This measure is certainly natural for rings assuming negligible internal processing time.
Three classes of rings are considered in the literature:

1. **Unidirectional rings**: Processor $P_i$ can send messages only to $P_{i+1 \pmod{n}}$.

2. **Bidirectional rings with common sense of orientation**: Each processor $P_i$ can send messages to both sides, and it knows which of its adjacent edges connects it to $P_{i+1 \pmod{n}}$.

3. **Bidirectional rings without common sense of orientation**: Each processor $P_i$ can send messages to both sides, but it does not know which of its adjacent edges connects it to $P_{i+1 \pmod{n}}$.

Obviously, the second class is at least as powerful as the first (or the third) class, in the sense that every algorithm proposed for unidirectional rings (or for bidirectional rings without common sense of orientation) will work also in a bidirectional ring with common sense of orientation. But no other relationship between the classes is known so far. For example, it is not known whether the second class of rings is more powerful than unidirectional rings, in the sense that there are algorithms for this class that solve the leader problem in a number of messages bounded by $f(n)$ while every algorithm which solves the problem in one of the other classes uses at most $g(n)$ messages satisfying $\lim_{n \to \infty} \frac{f(n)}{g(n)} < 1$. In the sequel unless otherwise stated, bidirectional ring will mean bidirectional ring without common sense of orientation.

In [L] an algorithm which solves the problem for unidirectional rings in $O(n^2)$ messages is presented. In [CR] an algorithm that solves the problem in $O(n \log n)$ messages in the average but $O(n^2)$ messages on the worst case is proposed. In [HS] a bidirectional algorithm which uses at most $8n \log n + O(n)$ messages is proposed. This algorithm is later improved in [B] to use at most $6n \log n + O(n)$ messages. But the latter makes use of the sense of orientation. In [GHS] (as a consequence of an algorithm to find the minimum spanning tree in a general graph) an algorithm for bidirectional rings, which requires only $5n \log n + O(n)$ messages in the worst case is obtained. In [P] and [DKR] the same unidirectional algorithm which at the worst case uses $2n \log n + O(n)$ messages is independently obtained. With improvements to
this basic algorithm, in [P] a unidirectional algorithm which has a worst case message complexity $1.44 \ldots n \log n + O(n)$ and in [DKR] a unidirectional algorithm which uses at most $1.356 \ldots n \log n + O(n)$ messages is obtained. The latter is the best algorithm known so far for the unidirectional case. In [KRS] a bidirectional algorithm that uses at most $1.89 \ldots n \log n + O(n)$ messages is proposed. This algorithm is the best algorithm known so far for the bidirectional case.

In this paper we present a bidirectional algorithm (based on Peterson's unidirectional algorithm [P]) that uses at most $1.44 \ldots n \log n + O(n)$ messages. Recently a similar result of $1.44 \ldots n \log n$ was independently found in [LT].

The idea in our bidirectional algorithm is as follows: each processor first chooses a direction, along which it will execute the unidirectional algorithm in [P]. If it so happened that all chose "the same" direction, the execution will be exactly as in [P], otherwise conflicts will occur, and we show that they can be resolved and still maintain the same message complexity.

Section 2 describes Peterson's algorithm, Sections 3 and 4 present our algorithm, Section 5 proves its correctness, and Section 6 analyzes its complexity.
2. Peterson's Algorithm

We present here a unidirectional algorithm that is a slight variation of the algorithm in [P]. This algorithm selects a leader, rather than finds the processor with maximum identity (as the original algorithm does). We made this change in order to ease the modification to the bidirectional case.

Algorithm P:

```
begin
status := active;
phase := 0;
send (phase, id); /* id is iid for P_i as mentioned in the first section */
repeat
    receive (ph, nid);
    case status of
        active : active (ph, nid);
        relay : send (ph, nid)
    end
until status = leader
end.
```

```
procedure active (ph, nid);
begin
    if nid = id then
        status := leader
    else
        case ph mod 2 of
            0 : if id > nid then
                begin
                    phase := ph + 1;
                    send (phase, id);
                end
            else
                status := relay;
            1 : if id < nid then
                begin
                    phase := ph + 1;
                    send (phase, id);
                end
            else
                status := relay
        end;
end;
```

Note that the algorithm above is message driven, i.e. each processor sends a message only at the beginning of the algorithm or after receiving a message from another processor. Assuming that all the processors are initially awakened, it can be shown by induction on i, for each processor P_i that the i-th message sent by it during two different executions of the algorithm are the same. This assumption can be made true by letting each processor that receives a message of the algorithm and did not started
the algorithm yet, start the algorithm.

By the end of the execution of the algorithm exactly one processor will have status = leader and all the others will have status = relay and be waiting in their receive statements. In other words the execution will result in a deadlock. This can be avoided by letting the leader initiate a special LEADER message, which will forwarded by all the processors on the ring and causes them to terminate the algorithm, increasing the total number of messages sent by n. This procedure is not added to the above code in order not to complicate it.

At any time during the execution of the algorithm there are two types of processors according to their status, namely active and relay. The relay processors merely forward the messages they receive. Thus we can view the algorithm as being executed only at the active processors. At the beginning of the execution of the algorithm each processor is active. The algorithm proceeds in phases starting with phase = 0 at each processor. An active processor receiving a message reacts according to the parity of its phase: at even (odd) phases a processor which receives a message from another processor of bigger (smaller )identity becomes relay , otherwise it proceeds to the next phase and sends an appropriate message which in turn will be received by an active processor from this phase. The difference between the algorithm in [P] and the above algorithm is that in the latter an active processor of odd phase does not change its identity prior to proceeding the next phase. The only processor which will survive as active until the last phase will become the leader. This processor will detect this fact receiving its own message.

Note that the first part of a message (which indicates the phase of the active processor which initiated it) is redundant. We include it in the message in order to make the analogy to the bidirectional algorithm easier. This redundancy follows from the fact that the phase carried on a message of the algorithm is equal to the number of messages sent on this communication line, before this message. This can be proved by induction on the phase carried on the message.
The following short discussion is an outline of the correctness proof of algorithm P: $A_p$ is the set of all the processors which are active in phase $p$. A processor which is active may become relay but the converse may never occur, moreover if there are at least two processors in $A_i$, the processor with the maximum (minimum) identity in $A_i$ will be in $A_{i+1}$ for $i$ even (odd). On the other hand the processor with the minimum (maximum) identity in $A_i$ will not be in $A_{i+1}$ for $i$ even (odd). Thus

$$A_0 \supset A_1 \supset A_2 \supset \cdots$$

and

$$|A_i| > 1 \Rightarrow A_{i+1} \neq \emptyset.$$ 

Obviously, if there is only one processor in $A_i$ it will receive its own message and will be the leader terminating the algorithm.

As for the complexity analysis, it is proved in [P] that the sequence:

$$n = |A_0| = |A_1| = |A_2| = \cdots = |A_p| = 1$$

has the following property: $|A_{i+2}| \leq |A_i| - |A_{i+1}|$ in other words $|A_i| \geq |A_{i+1}| + |A_{i+2}|$

proving that the reverse sequence grows at least as fast as the Fibonacci sequence, namely $|A_{p-i}| \geq F_{i+1}$ where $F_i$ is the $i$-th Fibonacci number ($F_0 = F_1 = 0$, $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$). Because $|A_0| = n$ this proves:

$$p \leq \frac{1}{\log_2 \frac{1 + \sqrt{5}}{2}} \log n \approx 1.44 \ldots \log n$$

As every processor sends one message of each phase the total number $M$ of messages sent will satisfy

$$M \leq 1.44 \ldots n \log n.$$
3. AN INFORMAL DISCUSSION OF THE BIDIRECTIONAL ALGORITHM

We now present an informal discussion of the algorithm that elects a leader in a bidirectional ring. The main idea is to modify algorithm P for this bidirectional environment.

At the beginning each processor $p_i$ has two indistinguishable edges adjacent to itself which it labels arbitrarily by $\alpha_i$ and $\beta_i$. It randomly specifies $\alpha_i$ as the one in the forward direction. Two forward directions are said to be consistent if they are both clockwise or both counter clockwise.

The messages of the algorithm are of the form $(\text{phase}, id, status)$ when each represent the phase, identity and status of the processor that originated the message.

A processor tends to run algorithm $P$ in its forward direction. It sends the first message exactly as in algorithm $P$ and will send further messages as a response to messages it receives. If in the beginning of the algorithm all the forward directions are consistent then algorithm $P$ will be executed. Whereas in the unidirectional case a processor knows the phase of the next message it will receive (and obviously its direction), in the bidirectional case it does not know where the next message will come from and what its phase will be.

1. The message is received from the backward edge:
   1.1. The message carries a phase smaller than the processor's phase: In this case the message is ignored.
   1.2. The message carries a phase bigger than the processor's phase: In this case the processor becomes a relay and forwards the message (in its forward direction).
   1.3. The message carries a phase equal to the processor's phase: In this case the processor reacts according to algorithm $P$.

2. The message is received from the forward edge:
   2.1. The message carries a phase smaller than the processor's phase: In this case the message is ignored.
(2.2) The message carries a phase bigger than the processor's phase: In this case the processor becomes a relay, changes its forward direction and forwards the message (in its new forward direction).

(2.3) The message carries a phase equal to the processor's phase: This situation (which in the sequel will be called collision) is clearly detected by two adjacent processors, after receiving messages of each other. The processor which received the smaller identity becomes a relay, the other one changes its identity to be this bigger identity, changes its status to a special status (termed copy), increment its phase, changes its forward direction and and sends a message in this (forward) direction.

In this copy status a processor reacts to the messages according this algorithm in cases (1.1), (1.2), (2.1), (2.2) and (2.3). In case (1.3):

If the message carries its own identity and is originated by an active processor, it becomes relay and forwards the message, otherwise it becomes active and continues the algorithm.

As will be shown in Section 5, during the algorithm there will be at most two processors with the same phase and identity. In case of two, one will be in an active status and one in a copy status. The one in the copy status will become a relay in the next phase. Intuitively a copy processor comes to ensure deadlock freeness in the case that both active processors which the messages originated by them collided (met between two adjacent processors) became relay. It behaves as a copy of one of those processors. If, in the next phase it does not receive a message from the processor which it copies, it changes its status to active replacing it completely. Otherwise it changes its status to relay because there is no place to the copy when the original processor is still active.
4. THE BIDIRECTIONAL ALGORITHM

Following is the algorithm to be performed by a processor in the bidirectional ring. Each processor uses the following local variables:

- **id**: which will be initialized to the processor's own identity but may change during the algorithm.
- **status**: This variable may have one of the values `relay`, `active`, `copy` and `leader`. As in algorithm P, the aim of the algorithm will be to make this variable equal to `leader` in exactly one of the processors.
- **phase**: The phase of the processor which is a non-negative integer, initially zero.
- **forward**: The direction in which the processor tends to execute algorithm P. May have one of the values `α` and `β`.
- **last id**: The id which is carried on the last message sent by this processor. This variable is needed in case of collision.
- **dir**: The direction from which the last message is received.
- **nid**: The id carried on the last message received.
- **st**: The status carried on the last message received.
- **ph**: The ph carried on the last message received.

`α` and `β` are two constants representing the adjacent edges of the processor.

We assume the existence of three primitives:

1. **send** `(direction, arg_1, arg_2, arg_3, ....)` that sends a message `(arg_1, arg_2, arg_3, ....)` along the edge in the given direction.
2. **receive** `(direction, arg_1, arg_2, arg_3, ....)` that seeks for the first message arrived from any direction, and puts that direction `(α or β)` into the variable `direction`, and the contents of the message into the variables `arg_1, arg_2, arg_3, ....`
3. **oppos**(dir) which given `α` as argument returns `β` and vice versa.
begin
id := initiaUd.
status := active.
repeat
forward := transmit (forward, id, status).
receive (dir, ph, nid, status).
if ph > phase then
if ph > phase then
if ph = phase then
\{ Case 2.3 \}
end, idph < phase then
\{ Case 1.2 and 2.2 \}
copy := status := relay.
od collison (nid, status) ;
case status := active (nid, status) ;
else
status := active (nid, status) ;
end.
end.
end.
end.
begin
forward := transmit (forward, id, status).
receive (dir, ph, nid, status).
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
end.
procedure transmit (dir, ph, iden, st) ;
begin
  forward := dir,
  last_id := iden,
  phase := ph,
  line f
  send (forward, phase, iden, st)
end;

Note that procedure active above is practically same as in algorithm P

This reflects the fact that in this part of the ring the algorithm
behaves as algorithm P

Example:

Consider a ring of 10 processors, with initial identities and initial forward directions as depicted in Figure (4.1a). Figures (4.1b),(4.1c) and (4.1d) represent the situation of the processors after a certain time period has elapsed. The numbers above the circles denote the corresponding processor's phase, the numbers below the circles denote their id, an arrow represents the processor's forward direction.

The situation in Figure (4.1b) arises from the situation in Figure (4.1a) according to the following scenario: Processor $p_3$ (in a similar manner as $p_7$) receives the message $(0, 10, active)$ which causes it to detect a collision. It becomes copy because $10 > 8$. $p_2$ receives the message $(0, 1, active)$. It proceeds to the next phase because $1 < 10$ and $0 \mod 2 = 0$. $p_1$ (in a similar manner as $p_6$) receives the message $(0, 3, active)$. It becomes relay because $3 > 1$ and $0 \mod 2 = 0$. Processor $p_2$ (in a similar manner as $p_7$ and $p_9$) now receives the message $(0, 8, active)$ and ignores it. Processor $p_6$ receives the message $(0, 4, active)$, detects a collision but ignores the message because $4 < 6$. The message sent by $p_9$ does not reach its destination yet.

The scenario from Figure (4.1b) to Figure (4.1c) is as follows: Processor $p_8$ receives the message $(1, 6, copy)$ becomes relay and forwards it because $1 > 0$. It then receives the message $(0, 5, active)$ and ignores it because $0 < 1$. $p_3$ receives the message $(1, 10, active)$ and becomes relay because it received a message from the processor it copies. The message $(1; 10, copy)$ (send by processor $p_3$) is received by $p_4$, $p_5$ and $p_6$ which forward it to $p_7$. The latter processor proceeds to phase 2 because $10 > 6$
and $1 \mod 2 = 1$. In the same way the message (1, 6, copy) is forwarded to $p_2$ which becomes relay because $6 < 10$ and $1 \mod 2 = 1$.

The scenario from Figure (4.1c) to Figure (4.1d) is simple: The message (2, 6, active) is sent by $p_7$ is forwarded by all the other processors to $p_7$ itself which becomes the leader. Figure 4.2 describes another scenario which begins from the same initial identities and forward directions and results in the election of $p_3$ as the leader.

![Diagram](image-url)
Figure 4.2
5. CORRECTNESS PROOF

We now prove the correctness of our algorithm. \textit{var}_i denotes the variable \textit{var} in the local memory of processor \textit{p}_i, e.g. \textit{status}_i stands for the variable \textit{status} in the code of the algorithm executed at processor \textit{p}_i, etc. \textit{dir} 1 \sim \textit{dir} 2 means that directions \textit{dir} 1 and \textit{dir} 2 are consistent. Assume any possible execution of the algorithm.

\textbf{Lemma 5.1:}

(a) \textit{phase}_i is initially zero, and is incremented at least by one whenever it is assigned a value.

(b) Each processor sends at most one message of a particular phase.

\textbf{Proof:} (a) The only place where \textit{phase}_i is assigned a value is line \textit{f} in procedure \textit{transmit}. This procedure is invoked only in lines \textit{a} through \textit{e}. In line \textit{a} the second parameter is zero, and will be assigned to \textit{phase}_i in line \textit{f}. In lines \textit{c}, \textit{d} and \textit{e}, the second parameter is \textit{phase} + 1 (and again will be assigned to \textit{phase}_i which will therefore be incremented by exactly one in this case), and in line \textit{b} the second parameter is \textit{ph} which is greater than \textit{phase}_i. (b) Follows from (a) and the fact that the primitive \textit{send} is used only in procedure \textit{transmit}.

If at any time during the execution of the algorithm \textit{phase}_i is assigned the value \textit{p} (in line \textit{f}) then \textit{var}_{i,p} denotes the value of \textit{var}_i just after the execution of this assignment statement, otherwise it is defined as:

\[ \textit{var}_{i,p} = \textit{var}_{i,p'} \]

when \textit{p'} is the biggest value assigned to \textit{phase}_i less than \textit{p}, except for \textit{status}_{i,p} which is defined as \textit{relay} in this case.

Let \( A_p \) be the set of processors were in phase \textit{p} in status \textit{active} or \textit{copy}, namely

\[ A = \left\{ p_i \mid \text{status}_{i,p} \in \{\text{active, copy}\} \right\}. \]

\( A_p(x) \) denotes the set of those processors in \( A_p \) with identity \textit{x} in this phase, namely...
\[ A_p(x) = \{ p_i \in A_p \mid id_{i,p} = x \}. \]

\( P_p(i,j) \) is the set of all processors between \( p_i \) and \( p_j \), excluding \( p_i \) and \( p_j \) in the direction forward, from \( p_i \).

**Lemma 5.2** \( p_i \in A_p \) \( (p \neq 0) \) if and only if:

(i) \( p_i \in A_{p-1} \) and it receives a message of an even phase \( p - 1 \), and the identity carried on the message is smaller than \( id_{i,p-1} \), or

(ii) \( p_i \in A_{p-1} \) and it receives a message of an odd phase \( p - 1 \), and the identity carried on the message is greater than \( id_{i,p-1} \), or

(iii) There is a collision detected by \( p_i \), and the identity carried on the message is greater than the identity of the last message sent by \( p_i \).

**Proof**: \( p_i \in A_p \) then \( \text{phase}_i \) is set to \( p \) in line f, which means that procedure \( \text{transmit} \) was invoked (lines a through e). It could not be invoked in line a (since \( p \neq 0 \)), or in line b (since then \( \text{status}_{i,p} = \text{relay} \) and \( p_i \notin A_p \)). In lines c, d and e the situation is exactly as described in (i), (ii) and (iii), respectively.

**Corollary**: A processor is in \( A_p \) \( (p \neq 0) \) only if it receives a message of phase \( p - 1 \).

**Lemma 5.3**: Relay processors do not change neither the identity nor the status of the messages they forward.

**Proof**: Relay processors never invoke procedure \( \text{active} \), thus they may invoke procedure \( \text{transmit} \) only in lines b and e. In line e \( \text{status}_i \) is \( \text{copy} \), thus a relay processor may send a message only by executing line b. In line b the parameters are exactly the same as in the last message received namely \( ph, nid \) and \( st \).

**Lemma 5.4**: A processor \( p_i \) changes the value of \( id_i \) only while executing procedure \( \text{collision} \). \( id_{i,p} = id_{j,p-1} \) where \( ph \) is the value of \( \text{phase} \) after this execution and \( p_j \) is the processor that initiated the message which caused the collision.

**Proof**: Follows immediately from the previous lemma and the code of procedure \( \text{collision} \).
The following theorem describes the set of processors of given phase and identity.

**Theorem 5.1**: For every phase \( p \) and every identity \( x \) exactly one of the following holds:

(i) \( A_p(x) = \emptyset \).

(ii) \( |A_p(x)| = 1 \).

(iii) \( A_p(x) = \{p_i, p_j\} \), with \( status_{i,p} = \text{active} \), \( status_{j,p} = \text{copy} \), \( forward_{i,p} \sim forward_{j,p} \) and \( P_p(i,j) \cap A_p = \emptyset \).

**Proof**: Let \( x \) be any identity. We prove by induction on \( p \).

For \( p = 0 \): At the beginning of the execution there are no two processors with the same \( \text{id} \), so either (i) or (ii) holds.

We assume the theorem holds for \( p \) and prove it for \( p + 1 \). By the inductive hypothesis \( A_p \) satisfies (i), (ii) or (iii).

**Case 1**: \( A_p = \emptyset \).

In this case \( A_{p+1}(x) = \emptyset \). Otherwise, if \( p_i \in A_{p+1}(x) \), then by Lemma 5.2 and Lemma 5.4 follows that \( A_p(x) \neq \emptyset \), a contradiction.

**Case 2**: \( A_p(x) = \{p_i\} \).

We show that \( A_{p+1}(x) \) contains 0, 1 or 2 elements, and in the last case it satisfies (iii). At most one processor \( p_j (j \neq i) \) may have its \( \text{id} \) set to \( \text{id}_{p}(=x) \) by collision, and therefore \( |A_{p+1}(x)| \leq 2 \). Assume \( A_{p+1}(x) = \{p_i, p_j\}, i, j \) as above. Thus \( p_i \) sent a message \( (p, x, status_{i,p}) \) along its forward direction and \( p_j \) received the same message. The message received is the one which was sent by \( p_i \) and forwarded by the other processors because no other processor could initiate the same message. This implies that all the processors in \( P_p(i,j) \) sent the message \( (p, x, status_{i,p}) \). Therefore no processor in \( P_p(i,j) \) except the one which is adjacent to \( p_j \) could receive another message of phase \( p \) (Lemma 5.1 (b)). The latter becomes relay when it receives the message \( (p, y, st) \) from processor \( p_j \) implying \( P_{p+1}(i,j) \cap A_{p+1} = \emptyset \). From the code of procedure collision follows that
\[
\text{status}_{j,p+1} = \text{copy}
\]
and
\[
\text{forward}_{j,p+1} \sim \text{forward}_{i,p}.
\]
Thus \(p_i \in A_{p+1}\), thus it received a message of phase \(p\) (according to Lemma 5.2) and not from its forward direction, and executed line \(c\) or line \(d\). Therefore
\[
\text{forward}_{i,p+1} \sim \text{forward}_{i,p}
\]
implying
\[
\text{forward}_{j,p+1} \sim \text{forward}_{i,p+1}
\]
and
\[
\text{status}_{i,p+1} = \text{active}.
\]
Case 3: \(A_p(x) = \{p_i, p_j\}\) and all the other conditions of (iii) hold.

As all the processors in \(P_k(i,j)\) are relay it is clear that \(p_j\) will receive the message \((p, x, \text{active})\). From the code it follows that it will become relay, so there may be at most two processors in \(A_{p+1}(x)\): \(p_i\) which may remain active and some other processor \(p_j\) which may detect a collision and sets its id to \(id_{j,p}(=x)\). The arguments are exactly same as for case 2 and (iii) holds for \(p+1\).

A graph \(G = (V_1 \cup V_2, E)\), \(V_1 \cap V_2 = \emptyset\) where
\[
V_1 = \{a_1, a_2, \ldots, a_i\},
\]
\[
V_2 = \{b_1, b_2, \ldots, b_j\},
\]
and
\[
E = \{(a_i, a_{t+1}) | 1 \leq t < i\} \cup \{(b_i, b_{t+1}) | 1 \leq t < j\} \cup \{(a_i, b_j), (b_j, a_i)\}
\]
is called a double chain.

The directions graph \(DG_p\) of phase \(p\) is a directed graph defined as follows:
\(V_{ph} = A_{ph}\), and \((p_i, p_j) \in E_{ph}\) if and only if \(P_{ph}(i,j) \cap A_{ph} = \emptyset\) in other words \(p_j\) is the closest non-relay process to \(p_i\) in the direction \(\text{forward}_{i,ph}\). Intuitively an edge \((p_i, p_j)\) of \(DG_p\) denotes that a message of phase \(p\) is initiated by processor \(p_i\) and is potentially to be received by \(p_j\). It is clear that \(d_{out}(p_i) = 1\) for all \(p_i \in A_p\), so \(DG_{ph}\) is either a
directed cycle consists of double chains. Lemma 5.5: $|A_p| \geq 3 \rightarrow A_{p+1} \neq \emptyset$.

**Proof**: Assume $A_{p+1} = \emptyset$. Thus $A_p = \emptyset$ for $p > p$ (by corollary after Lemma 5.2). We shall consider two cases:

**Case 1**: $DG_{ph}$ is a directed cycle.

We prove for an even $p$ (the case where $p$ is odd is similar): Consider the processor with the maximal idenity in $A_p$. If it is unique then it is easy to see that it will be in $A_{p+1}$. Otherwise by (iii) of theorem 5.1 we have two processors $p_i, p_j$ satisfying

$p_i, p_j \in A_p$, status$_{i,p} = active$, status$_{j,p} = copy$

$P_p(i,j) \cap A_p = \emptyset$.

$p_j \not\in A_{p+1}$ because it will receive $p_i$'s message. Moreover, $P_p(j,i) \cap A_p \neq \emptyset$ if there are at least three processors in $A_p$. Thus $p_i$ will receive a message from a processor with smaller $id$ and $p_i \in A_{p+1}$, a contradiction.

**Case 2**: $DG_{ph}$ is not a directed cycle.

There exist two processors which will send two messages towards each other. Then a collision will occur because all the processors between them are relays or of smaller phases, according to the hypothesis. Thus as a result of this collision there will be a copy processor in $A_{p+1}$, a contradiction.

\[ \square \]

**Lemma 5.6**:

(a) If $|A_p| = 1$ then the algorithm terminates at phase $p$.

(b) If $|A_p| = 2$ then the algorithm terminates at phase $p$ or at phase $p+1$.

**Proof**: If $A_p = \{p_i\}$ then $p_i$ receives its own message and become leader.

If $A_p = \{p_i, p_j\}$ and forward$\_i \sim$ forward$\_j$,

if $id_{i,p} \neq id_{j,p}$ then one of them is in $A_{p+1}$ and the other is not, depending on their id and $pmod2$, the unique processor in $A_{p+1}$ is elected leader,

if $id_{i,p} = id_{j,p}$ then (iii) of Theorem 5.1 holds: from the code follows that the processor with status copy will become relay and the other will be elected leader in
phase $p + 1$.

If $A_p = \{p_i, p_j\}$ and $\text{forward}_i \neq \text{forward}_j$, then there will be a collision and a processor $p_f$ (possibly $f$ is one of $i, j$) will be unique in $A_{p+1}$ and elected leader in phase $p + 1$.

In the next section we show that

**Lemma 5.7**:  

$$|A_0| \geq |A_2| \geq |A_4| \geq \ldots$$

Lemmas 5.5 and 5.7 and the fact that $|A_0| = n$ yield to:

**Corollary**: For every ring of $n$ processors and any execution of the algorithm there is a number $n$ such that $|A_n| \leq 2$.

**Corollary**: Lemma 5.5 and corollary above ensure termination of any execution of the algorithm with the desired result.
6. COMPLEXITY ANALYSIS

We analyze in this section the complexity of the algorithm. The outline of the analysis is similar to the one for the unidirectional algorithm (in Section 2). Assuming any execution of the algorithm, we prove that for all $p$

$$|A_{p+2}| \leq |A_p| - |A_{p+1}|$$

(*)

This, together with Lemma 5.5 implies Lemma 5.7. Moreover it proves that the number $\pi$ mentioned at the end of the previous section, satisfies:

$$\pi \leq 1.44 \ldots \log n$$

and the number $M$ of total messages sent during the execution of the algorithm satisfies:

$$M \leq 1.44 \ldots n \log n$$

for exactly the same reasons as in Section 2.

Let $DG_p$ be the directions graph of phase $p$ for this execution as defined in the previous section. Consider a double chain $(L \cup R, E)$ of $DG_p$ where $L = \{p_1,p_2,...,p_{|L|}\}$ and $R = \{p_{|L|+1},p_{|L|+2},...,p_{|R|}\}$. Let $S$ be the set of those processors in this double chain or between $p_{|L|}$ and $p_{|R|}$ that will be active or copy in phase $p+1$; namely,

$$S = \{p_1,p_2,...,p_{|R|-|L|}\} \cap A_{p+1}$$. We already know that $S \neq \emptyset$ (Case 2 of Lemma 5.5), so let $S = \{p_{|L|+1},p_{|L|+2},...,p_{|R|}\}$ with the (global) direction $\text{dir}$ if all the processors of $S$ are consistent in this direction in phase $p$; namely,

$$\text{forward}_{p+1} \sim \text{forward}_{p+2} \sim \ldots \text{forward}_{|S|,p+1} \sim \text{dir}$$

The Consistency Graph $CG_p = (V, E)$ of phase $p$ is constructed from the graph $DG_p$ as follows: if $DG_p$ is a directed cycle then $V = E = \emptyset$. Otherwise, each element in $V$ represents a double chain of $DG_p$, and if $u$ and $v$ represent two adjacent double chains $C_u$ and $C_v$ of $DG_p$, then $(u, v) \in E$ if $C_u$ is consistent in the direction $\text{dir}$ and $C_v$ is its adjacent double chain in this direction. A double chain $C_u$ of $DG_p$ represents a continuous section of the ring that does not receive (or send) any messages from (or to) the outside in phase $p$. Moreover, an edge of $CG_p$ outgoing from (incoming to) a vertex $u$ means that in phase $p+1$ $C_u$ will send (receive) a message to (from) outside. Therefore $E$ shows exactly the edges that will carry messages in phase $p+1$ between the double
chains of $\mathcal{D}_{G_p}$.

A graph $G = (V_1 \cup V_2 \cup \{u\}, E)$ where $V_1 \cap V_2 = \emptyset$, $u \not\in V_1 \cup V_2$, $V_1 = \{a_1, a_2, \ldots, a_i\}$, $V_2 = \{b_1, b_2, \ldots, b_j\}$, and $E = \{(a_t, a_{t+1}) \mid 1 \leq t < i\} \cup \{(b_t, b_{t+1}) \mid 1 \leq t < j\} \cup \{(a_i, u), (b_j, u)\}$ is called a weak double chain. Note that a single node and a directed path are special cases of weak double chain. The next lemma describes the structure of the graph $\mathcal{C}_{G_p}$.

From the definition of $\mathcal{C}_{G_p}$ it is clear that it is a directed graph with all out-degrees $\leq 1$, which implies the following:

**Lemma 6.1**: Exactly one of the following holds:

(i) $\mathcal{C}_{G_p}$ is an empty graph.

(ii) $\mathcal{C}_{G_p}$ is a directed cycle.

(iii) $\mathcal{C}_{G_p}$ is a graph consisting of components, each of which is a double chain or a weak double chain.

Each component of $\mathcal{C}_{G_p}$ (a cycle, a double chain or a weak double chain) is called a block. The next lemma follows directly from the definitions of $\mathcal{C}_{G_p}$ and $\mathcal{D}_{G_p}$.

**Lemma 6.2**: If $\mathcal{C}_{G_p}$ is a directed cycle then all the processors in $A_{p+1}$ for every $i \geq 1$ are consistent; namely, $\mathcal{D}_{G_{p+1}}$ is a directed cycle.

If there exists a number $p$ such that $\mathcal{C}_{G_p}$ is a directed cycle, let $\pi'$ be the smallest such number, otherwise let $\pi' = \infty$.

**Lemma 6.3**: The inequality $(\ast)$ holds for all $p > \pi'$.

**Proof**: According to the above lemma, $\mathcal{D}_{G_{p+1}}$ is a directed cycle and the algorithm performs as algorithm $\mathcal{P}$, for which $(\ast)$ holds.

Recall that each node of $\mathcal{C}_{G_p}$ represents a double chain of $\mathcal{D}_{G_p}$ and each node in $\mathcal{D}_{G_p}$ represents a processor in $A_p$. We therefore say that a block represents all the processors represented by its nodes and all the processors not in $A_p$ which are between them.
Lemma 6.4: A processor that is not represented by a block of $CG_p$ is not an element of $A_p, A_{p+1}$ or $A_{p+2}$.

Proof: This follows from Lemma 5.2 and the fact that such a processor will not receive a message of phase $p$ or $p+1$.

For a subset $X$ of processors denote by $X_p$ the set of all those processors in $X$ that are active or copy in phase $p$; namely, $X_p = X \cap A_p$.

In the sequel we prove

Theorem 6.1: For each block $B$ of $CG_p$

$$|B_{p+2}| \leq |B_p| - |B_{p+1}|$$

Lemma 6.5: The inequality (*) holds for all $p \leq \pi'$.

Proof: From the definition of $\pi'$ it follows that $CG_p$ is not empty, thus it can be partitioned into blocks that represent disjoint sets of processors. Summing up the inequalities of Theorem 6.1 (one inequality for each block), and using Lemma 6.4, the desired result is obtained.

Corollary: The inequality (*) holds for all $p$.

Proof of Theorem 6.1: Let $B$ be a block of $CG_p$. The proof continues by considering several cases. Each case will be accompanied by an appropriate diagram. In each of these diagrams there will be three parts, representing the situation of the processors of $B_p$ in phases $p, p+1$ and $p+2$. The notations for the diagrams are the same as in Section 4. In addition, missing arrows are not relevant to the discussion, a white node represents a processor the status of which is unknown and a set of nodes surrounded by a dashed rectangle is such that the inequality

$$|X'_{p+2}| \leq |X'_p| - |X'_{p+1}|$$

will be proved for one particular subset $X'$ of it.

Now we return to the proof.
Case A: The block B is a weak double chain consisting of a single node (which represents a double chain):

Assume that \( p \) is even and \( y > x \). There are three possible cases.

Case A1: \( p_i \in A_{p+1} \) and there are at least two processors in each side of the double chain (see Figure 6.1).

![Diagram showing the block B as a weak double chain with two possible configurations and a processor distribution on both sides of the chain.]

Two messages of phase \( p \) will collide between the rightmost processor of \( L_p \) and the leftmost processor \( (p_1) \) of \( R_p \), thus some processor \( p_i \) in this part of the ring will increment its phase to \( p + 1 \). By the algorithm we have that \( id_{i,p+1} = id_{i,p} = y \) and by Theorem 5.1 we have \( status_{i,p+1} = copy, forword_{i,p+1} = forword_{i,p+1} \) and \( P_{p+1}(t,l) \cap A_{p+1} = \phi \). Therefore the processor \( p_i \) will send a message of phase \( p + 1 \), and that message will collide with another message originated from this block, because otherwise the message will go all the way out of this block which implies the existence of an outgoing edge from the block \( B \) in \( CG_p \), a contradiction. Therefore there will be a processor \( p_m \) in \( B_{p+2} \) between \( p_i \) and the second processor on the left of \( B_p \).

We made the assumption \( p_i \in A_{p+1} \), which implies that \( p_i \) will become relay in phase \( p+2 \), namely \( p_i \not\in A_{p+2} \).
From the definitions of \( DG_p \) and \( CG_p \) it follows that every message received by a processor in \( B_i \) is sent by a processor in \( B_i \) \((i = p, p+1)\). Thus the leftmost and rightmost processors in \( B_p \) are not in \( B_{p+1} \) and the two leftmost and the two rightmost processors of \( B_p \) are not in \( B_{p+2} \) (for there are no processors that will send them a message in the corresponding phases). This is true except for the case when \( p_m \) is the second processor on the left in \( B_p \). As can be verified this case does not affect the results.

Let \( L' = L - \{p_i, p_m\} \) and \( R' = R \). Consider a processor \( p_i \) in \( L'_{p+2} \). It received a message of phase \( p \) originated by some processor \( p_j \) and \( id_{i,p} > id_{j,p} \) (recall that \( p \) is even) and survived to phase \( p+1 \). It then received a message of phase \( p+1 \) originated by some processor \( p_k \) in \( L'_{p+1} \subseteq L'_p \) and \( id_{i,p} = id_{i,p+1} < id_{k,p+1} = id_{k,p} \) (recall that \( p+1 \) is odd), and survived to phase \( p+2 \). Thus the originators of those two messages are different. Therefore \( p_j \not\in L'_{p+1} \) because otherwise it would send a message that would be received by \( p_i \), contradicting the fact that \( id_{j,p} \neq id_{k,p} \). Obviously \( p_j \in L'_{p} \) (note that \( p_i \) could not be the leftmost processor in \( L' \)). Therefore \( p_j \in L'_p - L'_{p+1} \). \( p_j \)'s message of phase \( p \) was clearly received by only one processor (namely \( p_i \)) which was in \( L'_p \), thus proving that the mapping \( f \) defined by \( f(p_i) = p_j \) is a one to one mapping from \( L'_{p+2} \) to \( L'_p - L'_{p+1} \). This implies \( |L'_{p+2}| \leq |L'_p - L'_{p+1}| \). But \( L'_{p+1} \subseteq L'_p \), hence \( |L'_{p+2}| \leq |L'_p| - |L'_{p+1}| \) which proves \((***)\) for \( L' \). Note that the last argument which holds similarly for \( R' \) is exactly the same as in \([P]\). Thus:

\[
\begin{align*}
|B_{p+2}| &\leq |L'_{p+2}| + |R'_{p+2}| + |\{p_m\}| \quad (6.1.1) \\
&\leq (|L'_p| - |L'_{p+1}|) + (|R'_p| - |R'_{p+1}|) + 1 \\
|B_p| &\geq |L'_p| + |R'_p| + 1 \\
&\quad \quad (6.1.2)
\end{align*}
\]

and

\[
|B_{p+1}| \leq |L'_{p+1}| + |R'_{p+1}| + 2 + |\{p_i\}| \quad (6.1.3)
\]

implying:

\[
|B_{p+2}| \leq |B_p| - |B_{p+1}|
\]

for this particular block.

**Case A2**: \( p_i \not\in A_{p+1} \) and there are at least two processors in each side of the double chain (see Figure 6.2).
The only difference from case A1 is that in this case $p_t \not\in A_{p+1}$ and therefore $p_t$ may be in $A_{p+2}$. Thus for $L' = L - \{p_m, p_t\}$ and $R' = R$ the argument is exactly as above and

$$|B_{p+2}| \leq |L'_{p+2}| + |R'_{p+2}| + |\{p_m, p_t\}|$$

(6.2.1)

$$\leq (|L'_p| - |L'_{p+1}|) + (|R'_p| - |R'_{p+1}|) + 2$$

$$|B_p| \geq |L'_p| + |L'_p| + |\{p_t\}| + 4 = |L'_p| + |R'_p| + 5$$

(6.2.2)

and

$$|B_{p+1}| \leq |L'_{p+1}| + |R'_{p+1}| + 2 + |\{p_t\}|$$

(6.2.3)

$$= |L'_{p+1}| + |R'_{p+1}| + 3$$

implying **) for $B$.

**Case A3** : There is only one element in one side of $B$ (see Figure 6.3).
Note that the numbers of the nodes that are not crossed and are outside of the dashed rectangles in Figure 6.1 are 4, 3 and 1, respectively, for each line, which are the underlined constants in the inequalities (6.1.1), (6.1.2) and (6.2.3) above, and satisfy the inequality $1 \leq 4 - 3$. For Figure 6.2 these numbers are 5, 3 and 2, respectively, and again $2 \leq 5 - 3$. For Figure 6.3 the numbers are 3, 2 and the same property holds.

**Case B:** B is a weak double chain which is a path of length $k$ (see Figure 6.4). Note that any node $v$ in $CG_p$ such that $d_{out}(v) = 1$ is a double chain of $DG_p$ such that all the processors of one of its sides became relay, namely $L'_{p+1} = \phi$ or $R'_{p+1} = \phi$ for $L' = L - \{p_1\}$ and $R' = R - \{p_1\}$. The numbers of the nodes that are not crossed and are outside of dashed rectangles in the three lines of the figure are respectively:

$$\begin{align*}
|B_{p+2}| &\leq |R'_{p+2}| + |\{p_m\}| - |R'_{p} - R'_{p+1}| + 1 \\
|B_p| &\geq |R'_p| + 3 \\
|B_{p+1}| &\leq |R'_{p+1}| + 1 + |\{p_1\}| = |R'_{p+1}| + 2
\end{align*}$$

implying (***) for B.
2 + 1 + (k - 1) + 2 = k+4
1 + 1 + (k - 1) + 1 + 1 = k+3

and satisfy the desired inequality.

Case C: $B$ is a weak double chain $(V_1 \cup V_2 \cup \{u\}, E)$ where $|V_1| = k \geq 1$ and $|V_2| = k' \geq 1$.

Figure 6.5 represents the case. The numbers of the nodes that are not crossed and are outside of the dashed rectangles are:

\[
2 + 1 + (k - 1) + (k' - 1) + 1 + 2 = k + k' + 4
\]
\[
1 + 1 + (k - 1) + 1 + (k' - 1) + 1 + 1 = k + k' + 3
\]

Case D: $B$ is a double chain $(V_1 \cup V_2, E)$ where $|V_1| = k \geq 1$ and $|V_2| = k' \geq 1$.

Figure 6.6 deals with this case. The number of the nodes that are not crossed and are outside of dashed rectangles are:

\[
2 + 1 + (k - 1) + (k' - 1) + 1 + 2 = k + k' + 4
\]
\[
1 + 1 + (k - 1) + (k' - 1) + 1 + 1 = k + k' + 2
\]

Case E: $B$ is a directed cycle with $k$ nodes as in Figure 6.7.

The number of the nodes that are not crossed and are outside of dashed rectangles are

\[
k
\]
\[
k
\]
\[
0
\]

This completes the proof of the theorem.
Figure 6.4
Figure 6.6
7. SUMMARY

We presented a leader finding algorithm for bidirectional rings having a message complexity of $1.44 \cdot n \log n$ which is obtained by a transformation on an existing algorithm for unidirectional rings. In this case the transformation preserved the worst case message complexity of the algorithm. It might be the case that this transformation can be used for other algorithms and other networks, help in proving a relationship between these two classes of rings (in terms of worst case message complexity), and help in the development of a methodology for the design of distributed algorithms.

An interesting observation is that the complexity analysis indicates that the worst case for the bidirectional algorithms is the case where all the processors initially choose the same direction; This contradicts the intuition that the algorithm is doing a lot of extra work to cope with the collision situation, which will not arise in such a case.

The following open problems are of interest:

(1) Is it possible to use our transformation on the algorithm in [DKR] to obtain a bidirectional algorithm of worst case message complexity $1.356 \cdot n \log n$?

(2) Is it possible to implement the improvement method in [DKR] on our algorithm to obtain this result?

(3) What can be said about the average message complexities of our algorithm and the unidirectional algorithm in [P]?
REFERENCES


[KRS] E. Korach, D. Rotem and N. Santoro, "Distributed Election in a Circle Without a Global Sense of Orientation"

