ELECTING A LEADER IN THE PRESENCE OF FAULTS:
A RING AS A SPECIAL CASE

by

L. Shrir and O. Goldreich

Technical Report #354

February 1985
Electing a Leader in the Presence of Faults: a Ring as a Special Case

Liuba Shiri

Oded Goldreich

Department of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel.

February 1985

ABSTRACT

We consider the problem of electing a leader in a network consisting of links which may fail. In particular, we study the special case in which the network is a ring of \( n \) processors (and one of its links may fail). We present an \( O(n \log n) \) algorithm for electing a leader in such a "faulty" ring.
1. Introduction

The problem of electing a leader in an asynchronous communication network has attracted the attention of many researchers. An efficient algorithm for the general bi-directed case has been suggested by Gallager, Humblet and Spira [GHS]. The special case in which the network constitutes a ring has gained much attention, both as a test case ([IL], [B], [IR1], [V]) and per-ce. In particular, lower bounds on the communication complexity of the problem were proved ([B], [FL]) and optimal algorithms were suggested ([HS], [F], [P], [DKR]).

In all the above works, it is postulated that every message sent on a link will eventually arrive (see also [LF]). Following [SR] and [IR2], we consider an asynchronous environment in which this postulate does not always hold. Instead, it is postulated that links may be faulty and lose messages. It is also postulated that the fact that a link is faulty can not be detected, by a processor, during the execution of an algorithm. We consider the problem of electing a leader in the presence of such faults, and concentrate on the special case in which the network constitutes a ring.

The main contribution of this paper is the presentation of an algorithm that, on input \( n \), implements electing a leader in a "faulty" \( n \)-processor ring (i.e. a ring which may contain a faulty link). The algorithm requires \( O(n \cdot \log n) \) messages and is optimal up to a multiplicative constant if the processors IDs are taken from an arbitrary set (and \( n \) is known) [FL]. Note that this extends the scope of the tightness of Frederickson and Lynch lower bound [FL] to the case of "faulty" rings.

In case the number of processors in the ring is not a-priori known, the simple \( O(n^2) \) algorithm is the best possible one (up to a multiplicative constant) [SG].

2. The Model

In this section we present the model of distributed computation to which we relate throughout the paper.

2.1 Communication Network and Distributed Algorithms

We assume a model of distributed computation in which processors send and receive messages via links, and links either transfer messages from one processor to the other or lose messages. Note that the asynchronous nature of communication is captured by the spontaneous nature of the message-transfer and message-
loss actions.

A link is said to be faulty during an execution if it has committed some message-lose actions during the execution. Note that the fact that a link is faulty can not be established by a processor during the execution of the algorithm. Intuitively, one cannot tell whether a message-transmission (that has not occurred yet) will not occur in the future.

2.2 The Complexity Measure

The complexity measure we consider throughout this paper is the worst-case number of message-transfer actions which occur during an execution of an algorithm.

2.3 Electing a Leader

Electing a leader in a distributed environment is a procedure through which one processor is being distinguished from all other processors. Formally, we postulate that leader is one of the possible local states of each processor and that a processor entering this state remains in it throughout the rest of the execution. Electing a leader is the task of reaching a situation in which there is a unique processor in the leader state.

Frederickson and Lynch have considered the problem of electing a leader in a synchronous and reliable n-processor ring, when the processor ID’s are taken from an arbitrary set and n is a-priori known. They have demonstrated a $\Omega(n \cdot \log n)$ lower bound for the problem of electing a leader in this case [FL]. The tightness of their bound for reliable, asynchronous networks is established by any of the known $O(n \cdot \log n)$ algorithms ([HS], [F], [DKR], [P]).

2.4 A Faulty Ring

A ring is a communication network, the underlying graph of which is a simple circuit. Without loss of generality, assume that processor $P_i$ is adjacent to processors $P_{i-1}$ and $P_{i+1}$, where here (and throughout the paper) indices are taken modulo n.

A ring is said to be faulty if some of its links may be faulty during an execution of some algorithm. A ring is said to be safe if it is guaranteed that during any execution of any algorithm, at most one of its links is faulty.

Consider the execution of some algorithm on the ring (PROCESSORS, LINKS), and let NFL denote the set of non-faulty links in this execution. Clearly, if a ring
is safe then the graph \((\text{PROCESSORS}, \text{NFL})\) is connected. Note that the connectivity of \((\text{PROCESSORS}, \text{NFL})\) is a necessary condition for the guaranteed accomplishment of tasks as electing a leader. Thus, we consider only safe rings.

3. Electing a Leader when the Number of Processors is Unknown

In this section we consider safe rings with unknown number of processors. We discuss two cases: the case in which a processor does not know the identity of any other processor in the ring, and the case in which each processor knows the identities of its neighbours.

In the first case, leader election can not be achieved. The reason being that there is no way to detect whether or not the active processors constitute one simple path. [Assume, on the contrary, that there exist an algorithm such that it elects a leader in each execution on a safe ring. Suppose that the faulty link transfers no messages during its execution. Thus, the links which transfer messages during this execution constitute a simple path containing a unique leader. These processors only know the identity of processors on the path. In particular, this holds for two identity-disjoint rings. Embedding the processors of the above two rings on one larger ring, we derive a contradiction.]

In case each processor a-priori knows the identities of its neighbours, a leader can be elected as follows. Each processor \(P_i\) broadcasts the triple \((ID_{i-1}, ID_i, ID_{i+1})\), immediately upon awaking (either spontaneously or by a message-receive event). Eventually, each processor can reconstruct the structure of the entire ring and elect the processor with maximum ID. The message-complexity of the above algorithm is \(O(n^2)\), when \(n\) is the number of processors. This simple solution is optimal, up to a multiplicative constant \([\text{SG}]\). Note that the above algorithm can be improved in terms of local storage, provided that the link behaviour is subjected to the FIFO rule. It is sufficient for each processor to keep the maximum ID, and the endpoints of the current path it has constructed.

4. Electing a Leader when the Number of Processors is Known

In this section we consider safe rings with a known number of processors. Obviously, the Frederickson and Lynch \(O(n \cdot \log n)\) lower bound holds also for the problem of electing a leader in a faulty \(n\)-processor ring. The algorithm presented in the next section demonstrates the tightness of their bound for the case of safe rings.
4.1 The Leader Electing Algorithm

In this section, we present an $O(n \log n)$ algorithm for electing a leader in a safe $n$-processor ring. The algorithm merges the exponential expending technique of Hirschberg and Sinclair [HS] with a phase-dependent "killing"-rule. This reduces the number of candidates to $O(1)$. Finally, instances of a "collecting acknowledgements" algorithm (see below) are initiated by the remaining candidates.

During the execution of the algorithm, a processor may be in one of the following states: idle, active, leader, passive. A processor is said to be active if it is either idle or active or in the leader state. Initially, all processors are idle. A processor can spontaneously "awake", passing from the idle state to the active state. While being in the active state, a processor $P_i$ operates in increasing local phases $\pi_i = 0, 1, 2, \ldots, L = \lfloor \log_2 n \rfloor$, initiating new messages each time it increases its phase. While being passive, a processor only passes messages.

Conventions:

Let $ID_i$ denote the identity of processor $P_i$. The local programs refer to the lexicographic order of pairs $(\text{phase}, ID)$ in order to determine the next action. Occasionally, processor $P_i$ compares $(\pi_i, ID_i)$ with a pair $(\tau, ID_j)$ contained in an incoming message.

In the description of the algorithm we use a primitive called forward a message hereby defined as follows: Forwarding $(m, d)$ means that $P_i$, upon receiving $(m, d)$ from $P_{i-1}$, sends $(m, d-1)$ to $P_{i+1}$ if $d \geq 1$ and sends $(m, n)$ to $P_{i-1}$ otherwise. [Upon receiving $(m, d)$ from $P_{i+1}$ forwarding means sending $(m, d-1)$ to $P_{i-1}$ if $d \geq 1$ and sending $(m, n)$ to $P_{i+1}$ otherwise.]

The algorithm uses a collecting acknowledgement algorithm as a subroutine. Collecting Acknowledgements (to a processor) is the task of letting a processor, $P_i$, know that all processors have received its message ($0 \leq i < n$). We denote by $CAA_i$ an algorithm implementing Collecting Acknowledgements to processor $P_i$. Efficient implementations of $CAA_i$ are presented in [SR] and [SG]. Further details can be found in Appendix.

All logarithms are taken to base 2. Recall that $L = \lfloor \log n \rfloor$.

The Leader Electing Algorithm (LEA) proceeds as follows:

0) INITIALIZATION: All processors are idle.

1) AWAKE: A idle processor can spontaneously become active. This means that the processor enters phase 0.
2) Upon entering a new phase, \( \pi_t \), processor \( P_i \) acts as follows:

2.1) If \( \pi_t < L \) then \( P_i \) sends \( ((\pi_t, ID_k), \sigma^{t-1}) \) to both \( P_{i-1} \) and \( P_{i+1} \).
2.2) If \( \pi_t = L \) then \( P_i \) initiates \( CA_{k} \).

3) Let \( P_i \) be a processor at phase \( \pi_t \) receiving the message \( ((\tau, ID_j), \sigma) \).

3.1) If \( (\tau, ID_j) = (\pi_t, ID_k) \) and \( P_i \) is active then \( P_i \) acts as follows:
- if \( \tau < L \) then \( P_i \) enters phase \( \pi_{t+1} \).
- if \( \tau = L \) then \( P_i \) proceeds according to \( CA_{k} \).

In case \( CA_{k} \) terminates, after having collected acknowledgements from all processors, then \( P_i \) enters the leader state and stops.

3.2) If \( (\tau, ID_j) > (\pi_t, ID_k) \) then \( P_i \) acts as follows:
- \( P_i \) sets its state to passive
- if \( \tau < L \) then \( P_i \) forwards the message \( (\tau, ID_j) \).
- if \( \tau = L \) then \( P_i \) proceeds according to \( CA_{k} \).

[In all other cases \( P_i \) ignores the message. This includes the case in which \( (\tau, ID_j) < (\pi_t, ID_k) \) and the case in which \( i = j \) and \( P_i \) is passive.]

4.2 Validity

Let us consider execution sequences of LEA. An execution sequence is a consistent extension of the partial order defined by an execution. The execution sequence consists of three types of events, and each event occurs in a single processor. The types of events are: message-send, message-receive, and processor-awake. The first event in an execution sequence is a processor-awake event. Also, an-processor-awake event can only be the first event occurring in a processor. Let \( H \) be such a sequence, we denote \( P_i \) 's phase after the first \( t \) events of \( H \) by \( \pi_i(t, H) \). Processor \( P_i \) is said to be alive after \( t \) events of \( H \) if \( P_i \) is not in the passive state after \( t \) events of \( H \). We define a processor \( P_i \) to be \( t \)-leading with respect to \( H \) if \( P_i \) is alive after \( t \) events of \( H \) and has maximum \( (\pi_i(t, H), ID_i) \) among all processors which are alive at this stage. We denote the index of the \( t \)-leader by \( lead(t) \). The \( t \)-leading pair is \( (\pi_{lead(t)}(t, H), ID_{lead(t)}) \).

Let \( H \) be a non-empty execution sequence of LEA. In case \( H \) is finite let \( T \) denote the length of \( H \); otherwise, \( T \) denotes infinity. (In case \( T \) denotes infinity, \( t < T \) holds for every integer \( t \).)

**Lemma 1:** For every \( 1 \leq t \leq T \), there exist a \( t \)-leader with respect to \( H \). Furthermore, \( (\pi_{lead(t-1)}(t-1, H), ID_{lead(t-1)}) \leq (\pi_{lead(t)}(t, H), ID_{lead(t)}) \), for \( 1 \leq t \leq T \).

**Proof:** By induction on \( t \). Trivial if \( t = 1 \). Assume the claim holds for all integers smaller than \( t + 1 \). We consider the following three cases:
CASE I: The \((t+1)\)-st event is an awakening event occurring in some processor \(P_i\). Thus, processor \(P_i\) is alive after \(t+1\) events of \(H\), and the existence of a \((t+1)\)-leader follows. Furthermore, the \((t+1)\)-leading pair either equals the \(t\)-leading pair or is greater than it.

CASE II: The \((t+1)\)-st event is a send-event occurring in \(P_i\). No processor, alive after \(t\) events of \(H\), ceases to be alive after the \((t+1)\)-st event. The existence of a \((t+1)\)-leader follows. Furthermore, \(\text{lead}(t+1) = \text{lead}(t)\).

CASE III: The \((t+1)\)-st event is a receive-message event occurring in some processor \(P_i\). The only processor which may change its state "during" the \((t+1)\)-st event is \(P_i\). We consider two subcases:

subCASE A: The state of \(P_i\) did not change during the \((t+1)\)-st event. Then the existence of a \((t+1)\)-leader follows (as in Cases I and II). Furthermore, the \((t+1)\)-leading pair either equals the \(t\)-leading pair or is greater than it.

subCASE B: The state of \(P_i\) changes during the \((t+1)\)-st event. Note that it changed from the alive state to the passive state, by the execution of step (3.2). We shortly prove that \(P_i\) is not the \(t\)-leader, and therefore the \(t\)-leader is also the \((t+1)\)-leader.

Claim: \(P_i\) is not the \(t\)-leader.

Proof: Recall that \(P_i\) becomes passive as a result of receiving the message \((\tau, ID_j)\), such that \((\tau, ID_j) > (\pi_i(t,H), ID_i)\). If \(P_j\) is alive after \(t\) events of \(H\), the claim follows immediately. Otherwise, let \(t'\) be the largest integer such that \(P_j\) was alive after \(t'\) events of \(H\). Note that \(\tau = \pi_j(t',H)\) and that the \(t'\)-leading pair is either equal or greater than \((\pi_j(t',H), ID_j)\). By the induction hypothesis, \(t\)-leading pair is either equal or greater than the \(t'\)-leading pair. We conclude that \((\pi_{\text{lead}(t)}(t,H), ID_{\text{lead}(t)}) \geq (\pi_{\text{lead}(t')}(t',H), ID_{\text{lead}(t')}) \geq (\pi_j(t',H), ID_j) > (\pi_i(t,H), ID_i)\) and the Claim follows.

Thus, the induction step is satisfied in all cases and the Lemma follows.

We have shown above that throughout the execution of LEA there is always a live processor: the current-leader. We now show that the messages initiated by the \(t\)-leader are being forwarded at least as long as their initiator is the current-leader. Namely

Lemma 2: Let \(1 \leq t < T\). Let \(P_j\) be the \(t\)-leader and let \(\tau = \pi_j(t,H)\). Suppose that \(i \neq j\) and that the \((t+1)\)-st event of \(H\) consists of processor \(P_i\) receiving the message \((\tau, ID_j, d)\). Then \(P_i\) acts as follows:

1) If \(\tau < L\) then \(P_i\) forwards \((\tau, ID_j)\).
2) If \( \tau = L \) then \( P_i \) acts as directed by \( CA_{A_i} \) upon receiving \( d \).

**proof:** Let us show that \( (\pi_i(t,H),ID_i) < (\tau, ID_j) \). This is trivial if \( P_i \) is alive after \( t \) events of \( H \). In case \( P_i \) is passive at this stage, let \( t' < t \) be the largest integer such that \( P_i \) is alive after \( t' \) events of \( H \). By the Claim presented in the proof of Lemma 1, processor \( P_i \) is not the \( t' \)-leader. Using Lemma 1 we get, \( (\pi_i(t,H),ID_i) = (\pi_i(t',H),ID_i) \). The Lemma follows by observing \( P_i \)'s local program. 

**Lemma 3:** If \( H \) is finite then \( P_{\text{lead}}(\tau) \) is in the leader state after \( H \).

**proof:** By Lemma 1, \( P_{\text{lead}}(\tau) \) is alive after \( H \). Since \( H \) is not empty \( P_{\text{lead}}(\tau) \) is either in the active state or in the leader state, after \( H \). Assume that \( P_{\text{lead}}(\tau) \) is active after \( H \). Let \( j = \text{lead}(T) \) and \( \tau = \pi_j(T,H) \). We consider two cases:

CASE I: \( \tau < L \). Consider the messages initiated when \( P_j \) entered phase \( \tau \). By Lemma 2(1), these messages can not be ignored by any processor. Note that these messages are sent to distance \( 2^n (\leq 2^{L-1} \leq \frac{N}{2}) \) in both possible directions. Thus, at most one of these messages is lost over a faulty link, and the other one must return to \( P_j \) causing it to increase its phase. This contradicts our assumption that \( \text{LEA} \) terminated with \( P_j \) at phase \( \tau \).

CASE II: \( \tau = L \). Consider \( CA_{A_i} \) which was initiated by \( P_j \) when it entered phase \( L \). By Lemma 2(2), the execution of \( CA_{A_i} \) was not interfered, and thus must have terminated after collecting acknowledgements from all processors. This contradicts our assumption that \( \text{LEA} \) terminated with \( P_j \) in the active state.

The Lemma follows.

**Lemma 4:** At most one processor enters the leader state during \( H \).

**proof:** Let \( P_j \) be the first processor entering the leader state during \( H \), and let \( t \) denote the number of events after which this happens. We now show that all other processors are passive after \( t \) events of \( H \).

Note that in order for \( P_j \) to enter the leader state, collecting acknowledgements from all other processors must have been achieved in it. It follows that all processors have participated in \( CA_{A_j} \). Evidently, each processor, (excluding \( P_j \)) participating in \( CA_{A_j} \) during the \( k \)-th event of \( H \), is passive after the \( k \)-th event of \( H \). Also, a processor passive after \( k \) events is also passive after \( k+1 \) events.

Thus, no other processor may enter the leader state.

Using Lemmas 3 and 4, we get...
Corollary 5: If $H$ is finite then there exist a unique processor which is in the leader phase after $H$. Furthermore, no other processor is in the leader state during $H$.

The validity of LEA follows from the finiteness of its executions. Finiteness can either be easily verified by observing LEA or trivially inferred from the complexity discussion that follows.

4.3 Message-Complexity

Lemma 6: Let $0 \leq \pi < L$. Let $I$ be an interval of $2^\pi$ consecutive non-faulty links, and $2^\pi + 1$ processors. There are at most two processors in $I$ that have entered phase $\pi + 1$ during $H$.

proof: Let $\text{LEFT}$, $\text{MID}$ and $\text{RIGHT}$ be any three processors that lie in $I$ such that $\text{MID}$ lies between $\text{LEFT}$ and $\text{RIGHT}$ in $I$. Suppose that $\text{MID}$ has entered phase $\pi + 1$ during $H$. It follows that the message $(\pi, \text{ID}_{\text{MID}})$ has been forwarded by either $\text{LEFT}$ or $\text{RIGHT}$. While doing so this processor is not in a greater phase than $\pi$, and after doing so this processor is in the passive state. The Lemma follows.

Notation: Let $0 \leq \pi \leq L$. We denote by $n_\pi$ the number of processors entering phase $\pi$ during $H$.

Corollary 7: Let $0 \leq \pi \leq L$. Then $n_\pi \leq 2\left\lfloor \frac{n}{2^{\pi-1}+1} \right\rfloor$.

proof: Trivial for $\pi \in \{0,1\}$. Let $\pi > 1$. Consider a partition of the ring into processor-disjoint intervals each consisting of $2^{\pi-1}$ non-faulty links, and a special interval consisting of the faulty link and less than $2^{\pi-1}$ non-faulty links. The total number of such intervals is $\left\lfloor \frac{n}{2^{\pi-1}+1} \right\rfloor$. By Lemma 6, each such interval contains at most 2 processors which entered phase $\pi$ during $H$. The Corollary follows.

Definition: A message $((\pi, \text{ID}_j), d)$ is said to be $(\pi,j)$-outgoing if $d < \frac{n}{2}$; and is said to be $(\pi,j)$-ingoing otherwise.

Lemma 8: Let $0 \leq \pi < L$. If a $(\pi,j)$-ingoing message is sent by processor $P_i$ during $H$ then no processor which lie between $P_j$ and $P_i$ (including $P_i$ but excluding $P_j$) does not enter the $(\pi+1)$-st phase. Furthermore, if $(\pi,j)$-ingoing messages are send from both $P_{j+r}$ and $P_{j-s}$ (where $1 \leq s, r \leq 2^n$) then $P_j$ enters phase $\pi + 1$.

proof: Let $r \leq 2^n$. First note that during $H$ at most one $(\pi,j)$-outgoing message is received by processor $P_{j+r}$. In case processor $P_{j+r}$ has ignored the $(\pi,j)$-outgoing message then no $(\pi,j)$-ingoing message is being forwarded during $H$, by any processor $P_{j+q}$, where $q \leq 2^n$. In case $P_{j+r}$ has forwarded the $(\pi,j)$-outgoing
message then it has set its state to passive, and cannot ignore \((\pi,j)\)-ingoing messages in latter stages. Note that previous messages initiated by \(P_{j+r}\) can not cause \(P_j\) to enter the passive state. The Lemma follows.

**Corollary 9:** Let \(0 \leq n < L\). The number of receive-events concerning messages of the form \(((\pi,\cdot),\cdot)\) is at most \(14n + 2^{n+3} = O(n)\).

**proof:** The number of \((\pi,\cdot)\)-outgoing messages received during \(H\) is at most \(n^\pi 2 \cdot 2^n\). By Lemma 8, the number of \((\pi,\cdot)\)-ingoing messages received during \(H\) is at most \(n_{n+1}^\pi 2 \cdot 2^n + (n_n - n_{n+1}) \cdot 2^n\). Let \(m_n\) denote the number of \(((\pi,\cdot),\cdot)\) messages received during \(H\). We have \(m_n \leq (3n_n + n_{n+1}) \cdot 2^n\). Using Corollary 7 we get

\[
m_n \leq 2^n \cdot \left(3 \cdot \left\lfloor \frac{n}{2^n} \right\rfloor + 2 \cdot \left\lfloor \frac{n}{2^n+1} \right\rfloor \right) < 2^{n+1} \cdot \left(3 \cdot \frac{n}{2^n} + \frac{n}{2^n+1} + 4 \right) \leq 14n + 2^{n+3}.
\]

**Corollary 10:** Let \(c(n)\) denote the complexity of the collecting acknowledgement algorithm in use. Then the number of receive-events concerning messages of the form \(((L,\cdot),\cdot)\) is \(O(c(n))\). Furthermore, using the collecting acknowledgements algorithm presented in [SG], the complexity involved in messages of the form \(((L,\cdot),\cdot)\) is at most \(2n \cdot \log n + 10n = O(n \cdot \log n)\).

**proof:** Using Corollary 7, one can easily verify that \(n_L \leq 8\). The complexity of the \(L\)-th phase is thus bound from above by \(8 \cdot c(n)\).

The first phase of the \(CAA_j\) presented in [SG] consists of processor \(P_j\) sending two copies of a message to processor \(P_{j+\left\lfloor \frac{n}{2} \right\rfloor}\) and waiting for \(P_{j+\left\lfloor \frac{n}{2} \right\rfloor}\)'s acknowledgement. Both the message and its acknowledgement are send via both possible directions. Therefore there exist at most two processors which conclude the first phase of their corresponding \(CAAs\). The complexity of the \(L\)-th phase of LEA is bounded by \(2 \cdot c(n) + 6 \cdot \frac{n}{2}\). Since \(c(n) = n \cdot \log n + 3.5n\) for that \(CAA\), the Corollary follows.

Combining Corollaries 5, 9 and 10, we get

**Theorem:** Let \(CAA\) be the collecting acknowledgements algorithm presented in [SG]. After \(O(n \cdot \log n)\) receive-message events, LEA locally terminates electing a unique leader.

**Corollary:** The message-complexity of the leader electing problem in safe \(n\)-processor rings is \(O(n \cdot \log n)\).
4.4 Conclusions Concerning LEA

We have presented an efficient algorithm for electing a leader in a safe ring. The methodology we have used is first to restrict the number of possible candidates and then to initiate a collecting acknowledgements task at each candidate. The success of this methodology, in the case we have considered, rests upon both the ability to efficiently restrict the number of possible candidates and the ability to efficiently collect acknowledgements. These abilities, in turn, rely on some global knowledge of the structure of the network (in our case the number of processors).

An interesting question is whether the above methodology can be applied to solve the leader election problem for a more general class of safe networks. Note that a positive answer is interesting only if it improves the obvious broadcast-based solution (which uses $O(n \cdot m)$ messages, where $n$ is the number of processors and $m$ is the number of links).

Acknowledgements

We would like to thank Sara Porat and Miki Rodeh for very helpful discussions.
References


Appendix: Details Concerning Collecting Acknowledgements Algorithm's

A simple collecting acknowledgements algorithm ($CAA_0$) is presented in [SR]. The $SR$-algorithm proceeds in phases. Each phase reduces by half the size of the "yet uncovered" part of the ring (the "covered part of the ring" is the interval of processors from which $P_0$ has already collected acknowledgements). This is done as follows:

At each phase, processor $P_0$ sends a "covering message" to the midst of the "yet uncovered part of the ring", via both possible directions. Upon receiving the first copy of the "covering message", the midst acknowledges it via both possible directions. Both types of messages are being forwarded by all processors from their initiator to their destination.

After $[\log n]$ phases the entire ring is covered. The message complexity of the $SR$-algorithm is at most $2n \log n$. A more efficient CAA, requiring at most $n \log n + 3.5n$ messages, is presented in [SG].

The reader may be interested to know that the CAA presented in [SG] is almost optimal. It has been proved that $n \log n - n$ is a lower bound on the message complexity of any CAA, even if it was specially designed to operate on a safe $n$-processor ring.