CONSULTATION IN THE PRESENCE OF FAULTS:
TWO LOWER BOUNDS

by
L. Shira and O. Goldreich

Technical Report #355
February 1985

TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

HAIFA, ISRAEL
Consultation in the Presence of Faults: Two Lower Bounds

Liuba Shiri and Oded Goldreich

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

February 1985

ABSTRACT

Consultation is the task of computing a simple function over local values scattered throughout the network, such that the result is obtained in one (specific) processor.

We consider the problem of implementing consultation 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 show that in such a ring, consultation requires $\Omega(n \log n)$ messages, even when $n$ is a-priori known. When $n$ is unknown, consultation requires at least $\frac{1}{2} (n^2 + n)$ messages. Both lower bounds are shown to be tight.
1. Introduction

Much attention has been recently concentrated in modeling and analyzing the behavior of communication networks. Message transmission delays in these networks are usually unpredictable and hard to quantify. Another disturbing phenomena is that links may fail, resulting in the loss of messages. It is therefore reasonable to model the behavior of communication networks by postulating that a message sent on a (non-faulty) link will eventually arrive; while a message sent on a faulty link may not arrive at all. Since there is no bound on transmission delays, there is no way to distinguish a message which has not yet arrived from a message which has been lost.

Developing and proving correctness of algorithms in the above framework is fairly involved. A helpful methodology consists of using high level abstractions as "announcement" (broadcast) and "consultation" ([SR]). Announcement is the task of sending (identical) information from one processor to all other processors in the network. Consultation is the task of computing a simple function over local values scattered throughout the network, such that the result is obtained in one (specific) processor. Consultation is a very useful abstraction. For example, consultation can be used to efficiently detect the fact that each processor has reached some desired local state.

A natural question regarding announcement and consultation is how efficiently can they be implemented, in the above model. Another interesting question is whether the complexity of these tasks depends on some global knowledge of the structure of the network. We explore these problems in a special communication network - a ring of n processors. A ring of processors has been used several times as a test case in the study of various problems regarding communication networks (e.g. [L], [H], [R1], [PL], [V]). A ring configuration was also considered in [CR], [HS], [F], [GKR] and [P].
The main contribution of this paper is the demonstration of tight lower bounds on the message-complexity of consultation. We show that, in a "faulty n-processors ring" (a ring which may contain a faulty link), consultation requires $\Omega(n \log n)$ messages, even if $n$ is a-priori known to all processors. This lower bound is robust in the sense that it holds even for a non-uniformed complexity measure. Loosely speaking, the consultation algorithm $A_s$ may be specially designed to operate on a $n$-processors ring and still $A_s$ requires $n \log n$ messages. The lower bound is tight up to a multiplicative constant, when the number of processors is a-priori known to each processor. It is interesting to note that a careful examination of our proof technique suggests an improvement of a previous consultation algorithm [3].

In case the number of processors is not a-priori known, the $\Omega(n \log n)$ lower bound is not tight. We show that in this case (i.e. $n$ is not known), consultation in a "faulty" $n$-processor ring requires $\Omega(n^2)$ messages. The $\Omega(n^2)$ lower bound is also tight. Note that announcement can be done using at most $n+1$ message transmissions, even if $n$ is not a-priori known.

**Organization**

In Sec. 2 we formulate describe the model, of a "faulty" ring, which we use throughout the paper. In Sec. 3 we present the main contribution of this paper: a $\Omega(n \log n)$ lower bound on the number of messages required by any algorithm which implements consultation in $n$-processor rings and demonstrate its tightness for the case when $n$ - the number of processors - is a-priori known. In Sec. 4 we present an $\Omega(n^2)$ lower bound on the number of messages required by any algorithm which implements consultation in $n$-processor rings when $n$ is not a-priori known and discuss its tightness. We conclude by extending the lower bound for other networks and by suggesting some open problems.
2. The Model

In this section we formally present the framework for this paper. This includes: the general model of communication to which we refer and its comparison to other models; the message-complexity measure; a formal definition of consultation and the special case of a "faulty ring"; and preliminary remarks concerning the implementation of consultation in it.

2.1 Communication Network and Distributed Algorithms

A communication network $N$ consists of a set of $n$ processors $P = \{P_1, P_2, \ldots, P_n\}$ and a set of links $L$ such that each link connects two processors. The processors have distinct IDs (names) and each processor knows the ID of the processor connected to it by each of its incident links. Each processor executes its local program and may send and receive messages over its incident links. The link incident at processors $P_i$ and $P_j$ (i.e., connecting $P_i$ and $P_j$) consists of two anti-parallel channels denoted $i\leftrightarrow j$ and $j\leftrightarrow i$. The channel $i\leftrightarrow j$ consists of two buffers denoted $\text{send-buffer}(i\leftrightarrow j)$ (SB$(i\leftrightarrow j)$ for short) and $\text{receive-buffer}(i\leftrightarrow j)$ (RB$(i\leftrightarrow j)$ for short). The $i\leftrightarrow j$ channel may transfer messages from SB$(i\leftrightarrow j)$ to RB$(i\leftrightarrow j)$.

A distributed algorithm is a set of deterministic local programs assigned to the processors of a communication network. The local program assigned to process $P_i$ consists of computational statements and communication statements. The computational statements refer only to the local storage of $P_i$. The communication statements are either of the form "send message $m$ to processor $P_j$" or of the form "receive a message from processor $P_j$", where $P_j$ is some processor connected (by a link) to $P_i$.

An execution of a distributed algorithm results in the following communication actions:
1) **Message-Send**: As a result of the execution of the local statement "send \( m \) to \( P_j \)" processor \( P_i \) places the message \( m \) on \( SB(i \rightarrow j) \).

2) **Message-Receive**: As a result of the execution of the local statement "receive a message from \( P_j \)" processor \( P_i \) removes a message \( m \) from \( SB(j \rightarrow i) \) and stores \( m \) in its \((P_i)\)'s local storage.

3) **Message-Transfer**: A message \( m \) is removed from \( SB(i \rightarrow j) \) and placed on \( RB(i \rightarrow j) \).

4) **Message-Loss**: A message \( m \) is removed from \( SB(i \rightarrow j) \).

We stress that the contents of the buffers (SB and RB) can be changed only by the above actions. Note that the asynchronous nature of communication is captured by the spontaneous nature of the message-transfer and message-loss actions (3 and 4 above). Also note that actions 3 and 4 are transparent to the processors.

An execution of a distributed algorithm is said to have terminated if each process has reached its local termination state and all send-buffers are empty.

A link connecting \( P_i \) and \( P_j \) is said to be faulty during an execution (of a distributed algorithm) if during the execution a message is lost either from \( SB(i \rightarrow j) \) or from \( SB(j \rightarrow i) \). The fact that a link is faulty can not be established by any processor, during the execution. Intuitively, one cannot tell whether a message-transmission (that has not occurred yet) will not occur in the future.

2.1.1 **Comparison to other models: no faults**

Classical models of asynchronous communication (as [L´] and [B]) assume that although the links can be arbitrary slow, they never lose messages. In these (classical) models implementing consultation is easy once a spanning tree is constructed. In the general case, such a tree can be constructed at the cost of \( O(L) + |P| \cdot \log |P| \) messages [GFS]. In the special case of a ring configuration, constructing such a tree is trivial.
Following [SR] and [IR2], the asynchronous model we use allows links to lose messages (i.e. to be faulty). In this model, as we will show in this paper, implementing consultation is far from being trivial. In the special case of a ring it requires $\Theta(P \log |P|)$ messages, when $|P|$ is a-priori known; and $\Theta(P^2)$ messages, when $|P|$ is not a-priori known. The general case calls for further research.

2.1.2 Comparison to other models: detection of faults

We have postulated that there is no time bound on the message-transfer delays, and that there is no way to detect a faulty link. It seems to us that the two assertions depend on each other, and best reflect the unpredictable nature of asynchronous communication. The results presented in this paper heavily rely on the above assertions.

A different approach was taken by Awerbuch and Even [AE]. In this pioneering work it was postulated that links' failures are detected and that there exist a bound on round-trip delay of a link. In particular, it was also shown how to efficiently implement broadcast (announcement) in the AE-model. It seems that consultation can be implemented in the AE-model within the same complexity as announcement.

2.1.3 Comparison to other models: "fault tolerance"

In the model we use, as well as in the AE-model, only faults on links are being considered. The processors are assumed to be reliable. Under this assumption, announcement can be always be achieved at the cost of $O(L)$ messages, and consultation can be achieved by initiating announcement at every processor. In this paper we study the possibility of introducing more efficient consultation algorithms and present lower bound on their efficiency in the special case of a ring.

There is a vast amount of literature concerning fault tolerance. This line of research, initiated to deal with faulty processors, also implies results concerning
the special case in which only the communication actions are unreliable. However, these implications do not seem to illuminate the problem we confront in this paper.

2.2 The Complexity Measure

The complexity measure we consider throughout this paper is the worst-case number of message-transfer actions which occurred during an execution of an algorithm. We consider two versions of this measure: a non-uniformed version in the lower bound on consultation when the number of processors is known; and a uniformed version in the lower bound on consultation when the number of processors is unknown. These versions are formally defined as follows:

Non-Uniformed Complexity Measure: Let \( A_N \) be a distributed algorithm implementing a task \( T_N \) on a specified communication network denoted \( N \). We stress that algorithm \( A_N \) is considered to accomplish its task (\( T_N \)) in each execution. Let \( e_x \) be a possible execution of algorithm \( A_N \) and let \( \text{comp}(e_x) \) denote the number of messages placed on the receive-buffers during the execution \( e_x \). The message-complexity (complexity) of the algorithm \( A_N \) is the maximum value of \( \text{comp}(\cdot) \) when taken over all possible executions of \( A_N \).

(Class)-Uniformed Complexity Measure: Let \( A \) be a distributed algorithm implementing a task \( T \) on a communication network which belongs to a class \( K \). Let \( e_x \) be a possible execution of algorithm \( A \) on a network \( N \in K \) and let \( \text{comp}(e_x) \) denote the number of messages placed on the receive-buffers during the execution \( e_x \). The message-complexity of algorithm \( A \) on the network \( N \) is the maximum value of \( \text{comp}(\cdot) \) when taken over all possible executions of \( A \) on \( N \).

Discussion: Note that in the non-uniformed measure it is not required that the algorithm works on all communication networks neither is its performance
evaluated over all networks. The algorithm $A_N$ may be specially tailored for the
network $N$, and its performance is evaluated only with respect to this network.

2.3 Consultation

A basic function is a binary function $f : \text{Dom} \times \text{Dom} \rightarrow \text{Ring}$ such that:

1) The function $f$ is associative and commutative.

2) The value of the function $f$ relies on both inputs. Namely: there exist $a_1, a_2, b \in \text{Dom}$ such that $f(a_1, a_2) \neq f(b, a_2)$.

[In fact, it is sufficient to postulate that $f$ is non-trivial.]

Let $f$ be a basic function. Define $f_1$ to be a binary function such that for every $a, b \in \text{Dom}$, $f_1(a, b) = f(a, b)$. For $n \geq 2$, define $f_n$ to be a function which maps the $n$-fold cartesian product of Dom into Ring such that for every $a_0, a_1, \ldots, a_{n-1} \in \text{Dom}$:

$$f_n(a_0, a_1, \ldots, a_{n-1}) = f(a_0, f_{n-1}(a_1, \ldots, a_{n-1})).$$

We call $f_n$ an $n$-consult function. The $n$-fold MAXIMUM, OR, AND, SUM are examples of useful $n$-consult functions. The following claims concerning the $n$-consult function $f_n$ can be easily verified using induction on $n$:

1) The value of $f_n$ depends on all $(n)$ inputs. That is, for every $2 \leq i \leq n-1$ there exist $a_0, a_1, \ldots, a_{n-1}, b \in \text{Dom}$ such that

$$f_n(a_0, a_1, \ldots, a_{n-1}) \neq f_n(a_0, a_1, \ldots, a_{n-1}, b, a_{n-1})$$

2) For every permutation $\pi$ (over $\{1, 2, \ldots, n\}$), every $2 \leq i \leq n-2$ and every $a_0, a_1, \ldots, a_n \in \text{Dom}$ we have

$$f_n(a_0, a_1, \ldots, a_{n-1}) = f(f_n(a_0, a_1, \ldots, a_{n-1}))$$

Furthermore, $f_n(a_0, a_1, \ldots, a_{n-1}) = f(a_0, f_n(a_1, a_2, \ldots, a_{n-1})) = f(f_{n-1}(a_0, a_1, \ldots, a_{n-2}), a_{n-1})$. 

Technion - Computer Science Department - Technical Report CS0355 - 1985
Consider a network of \( n \) processors and \( n \) values partitioned among the processors such that processor \( P_i \) contains the value \( v_i \in \text{Dom} \). Let \( f_n \) be a \( n \)-consultation function (based on the basic function \( f \)). Consultation is the task of computing \( f_n(v_0, v_1, \ldots, v_{n-1}) \) such that the result is obtained in \( P_0 \). Without loss of generality, consultation is initiated by processor \( P_0 \).

Consultation is a very useful abstraction. For example, consultation can be used to efficiently detect the fact that each processor has reached some desired local state. This can be done as follows:

One processor, \( P_0 \), initiates the execution of a consultation algorithm. Each processor \( P_i \) delays its execution of the local consultation program until it reaches the desired state.

Consultation is also related to collecting acknowledgments: the task of letting \( P_0 \) know that all processors have received its message. Any consultation algorithm can be modified to collect acknowledgments, and vice versa, preserving complexity.

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 \((P, L)\), and let \( NFL \) denote the set of non-faulty links in this execution. Clearly, if a ring is safe then the graph \((P, NFL)\) is connected. Note that the connectivity of \((P, NFL)\) is a
necessary condition for the guaranteed accomplishment of tasks as consultation. Thus, we consider only safe rings.

Clearly, announcement can be accomplished using $n$ messages on a safe $n$-processor ring. In the next section we show that this is not the case for consultation, even if the number of processors is known.

3. A Lower Bound on the Complexity of Consultation in a Faulty Ring

In this section, we present a $\Omega(n \log n)$ lower bound on the complexity of consultation on a faulty ring, where $n$ is the number of processors. The bound is robust, in the sense that it holds for the non-uniformed message complexity measure. That is, for every $n$ it is impossible to achieve consultation on a $n$-processors faulty ring within smaller complexity.

Intuitively, the difficulty in achieving consultation lies in the asynchronous nature of communication and the impossibility of distinguishing a faulty link from a slow one. The reader may think of the ring as being cut by the faulty link, resulting in a path. Loosely speaking, reaching consultation requires that processor $P_0$ collects information from all nodes on the path. To this end, processor $P_0$ should "cover" the path. Our proof "forces" the execution of a consultation algorithm into expanding the "covered area" in phases. It is shown that (in worst case) the "uncovered area" can not lose in one phase more than half its size. It is also shown that the "expanding phases" must be interleaved by phases of "sharing information", the complexity of which is proportional to the size of the "currently covered area". Combining the above observations yields the lower bound.
Organization of the Proof of the Lower Bound

We start by introducing the conventions used throughout the proof. These conventions consist of the notions of an "event", "global order of events", "rescheduling" and "a fault which persists". We conclude the first subsection by proving a basic and useful fact. Subsection 3.2 introduces the notion of "affecting" and some basic facts concerning it. This notion is used to impose the "first phase" which is then extensively studied. In subsections 3.3, the notion of "enabling" is introduced and employed in imposing the later phases. In subsection 3.4, we present a worst case execution of a consultation algorithm and show that its message complexity is \( O(n \log n) \).

3.1 Conventions and a Useful Fact

Recall that an execution of a distributed algorithm results in four types of communication actions: message-send, message-receive, message-transfer and message-loss (see section 2.1). Of these actions, only the first two directly involve the processors and only these two directly affect the next step a processor takes. Thus, it is sufficient to consider only the message-send and message-receive actions. These actions are hereafter called events.

Events

We denote by \( \text{SEND}(i,j,m) \) the event of placing message \( m \) (by \( P_i \)) on \( \text{SH}(i \rightarrow j) \); and by \( \text{RECEIVE}(i,j,m) \) the event of removing message \( m \) from \( \text{RS}(j \leftarrow i) \) and storing it in processor \( P_i \). The event \( \text{SEND}(i,j,m) [\text{RECEIVE}(i,j,m)] \) is said to occur in the processor \( P_i \). Without loss of generality, during an execution (of an algorithm) an event occurs at most once. [Otherwise append a running serial number to the identical messages sent via the same channel.]
We remark that the events are governed by causality, imposed from two sources: Link Behaviour (event RECEIVE(j,i,m) is caused by event SEND(i,j,m)) and Local Program (causality can be inferred from the local program, or from its execution in a processor). Since we do not use this term - we do not give a formal definition of it.

An Execution as a Partial Order of Events

An execution is a partial order of events, such that the partial order complies with the causality imposed by the link behaviour and the local programs. Without loss of generality, we may assume that all the events that occur in a processor are in total order (i.e. each processor has a local clock and no two events occur at the processor at the same time). The partial order which constitutes the execution is fully specified by the local orders (of events in each processor) and by the axiom SEND(i,j,m) precedes RECEIVE(j,i,m).

Initialization of the Consultation

Recall that the execution of a consultation is initiated by processor $P_0$ (see section 2.3). Thus, the first event in an execution is a send-message event which occurs (spontaneously) in processor $P_0$. The first event which occurs in any other processor is a receive-message event.

Global Order of Events and Execution Sequences

The partial order defined by an execution can be consistently extended to a total order. Note that this extension does not assume the existence of a global clock in the network; it rather demonstrates one possible global ordering of all the events has there been a global clock. This suggests the following definition:

An execution sequence (of a distributed algorithm) is a sequence of all events which occur during an execution (of the algorithm), such that the sequence
complies with the partial order of the execution. Namely, the execution sequence satisfies the following three properties:

1) Let $e_1$ and $e_2$ be two events occurring in $P_i$, such that $e_1$ occurs before $e_2$. Then $e_1$ precedes $e_2$ in the execution sequence.

2) The event $\text{SEND}(i,j,m)$ precedes the event $\text{RECEIVE}(j,i,m)$ in the execution sequence.

3) The first event in the execution sequence occurs in $P_0$ and the first event in any other processor is a receive-message event.

Evidently, there may be many different ways to extend a partial order to a total order. Thus, there may be several different execution sequences for a given execution of an algorithm. We extensively use this observation for transforming one execution sequence into another (both being extensions of the same execution). This transformation is referred to as "postponing" events, and is applied to events which do not affect each other.

Rescheduling

A more powerful tool than going from one execution sequence to another execution sequence (both extensions of the same execution), is going from one execution of an algorithm to another (execution of the same algorithm). This way of reasoning is called "rescheduling", and is applied to events which occur at the same processor. Rescheduling is used to demonstrate the existence of an execution that requires complexity equal to the lower bound we claim. Since our complexity measure is of worst-case type - this will suffice.

A Fault which Persists: Reduction to a Path

An algorithm which achieves consultation on a safe ring within a complexity
bound, must also achieve it in case the faulty link transfers no message throughout the execution of the algorithm. However, it is not at all clear that an algorithm that achieves consultation when guaranteed that a faulty link transfers no messages, will perform as well in case the faulty link alternatingly transfers and loses messages. One may think that an algorithm may take advantage over a persistent faultiness.

Nevertheless, we prove the lower bound for the special case in which the faulty link transfers no messages at all. Before going into the proof let us consider the consequences of dealing with the special case.

In case the link between $P_{k-1}$ and $P_k$ is faulty and transfers no messages at all, it is virtually a "pseudo-link". It looks like a link to the processors incident to it ($P_{k-1}$ and $P_k$), but as a matter of fact it does not exist. The ring is virtually a simple path, but the processors (including the end-points $P_{k-1}$ and $P_k$) do not know who its end-points are. This path will be denoted by $\text{path}_{\text{e}}$. We will assume that $\text{path}_{\text{e}}$ lies horizontally and refer to its left and right directions. We also assume, w.l.o.g., that a left to right scanning of the $\text{path}_{\text{e}}$ passes through processors $P_k, P_{k+1}, ..., P_{n-1}, P_0, P_1, ..., P_{k-1}$ (see Figure 1).

![Figure 1: path_e](image)

**A Useful Fact**

We conclude this subsection by stating a basic and very useful fact concerning executions of a consultation algorithm, $A$. 

---

Technion - Computer Science Department - Technical Report CS0355 - 1985
Lemma 1: Consider a prefix \( a \) of an execution sequence (of algorithm \( A \) on \( \text{path}_p \)). Assume that no \( \text{RECEIVE}(i-1,i+) \) event occurs in \( a \), where \( 0 < i \leq x - 1 \). Then the sequence of events which results by omitting from \( a \) all the events that occur in processors lying to the right of \( P_{i+1} \) is a prefix of an execution sequence (of \( A \) on \( \text{path}_p \)).

proof: Consider the sequence \( a' \) which results from \( a \) by postponing to its end all the events which occur in processors to the right of \( P_{i+1} \) during \( a \). Let \( b \) be a sequence of events such that \( a'b \) is an execution sequence (of \( A \) on \( \text{path}_p \)). The sequence \( a'b \) can be shown to be an execution sequence as it satisfies the three requirements in the definition of execution sequences:

1) The order of events occurring in a processor during \( a' \) is the same as it is during \( a \).

2) For every \( j \), \( \text{SEND}(j-1,j,m) \) precedes \( \text{RECEIVE}(j,j-1,m) \) in \( a' \). Also, for every \( j \), \( \text{SEND}(j-1,j,m) \) precedes \( \text{RECEIVE}(j-1,j,m) \). This holds (vacuously) also for \( i = j \), the reason being that no \( \text{RECEIVE}(i-1,i+) \) event occurs in either \( a \) or \( a' \).

3) Recall that \( P_0 \) does not lie to the right of \( P_{i+1} \) and therefore events occurring in it are not postponed. Thus, the first event in \( a' \) occurs in \( P_0 \). Evidently, the first event in any other processor is still a receive-message event (see 1 above).

Furthermore, the reader may easily observe that \( a'b \) and \( a'b \) correspond to the same execution (partial order). •

Above as well as throughout the entire section, symmetric facts and properties hold also for the "other direction". [i.e., when one substitutes \( i-1 \) by \( i+1 \) right by left (left by right) and \( 0 < i \leq x - 1 \) by \( x < i+1 \leq 0 \) etc.]
3.2 The Notions of Affecting and Imposing the First Phase

Let A be any consultation algorithm and consider execution sequences of A which satisfy the conventions of subsection 3.1. In this subsection we introduce the notion of a processor which "affects" an event at $P_0$. This notion is then used to impose the "first phase" (on execution sequences) and to infer facts concerning it.

Informal Discussion of the proposed Definition

We consider an execution sequence and a $\text{RECEIVE}(0,1,m)$ event which appears in it. Loosely speaking, the event $\text{RECEIVE}(0,1,m)$ is said to be affected by the processor $P_j$ if a necessary condition for $\text{RECEIVE}(0,1,m)$ event to occur in $P_0$ is that some event occurs in $P_j$.

Definition: Affecting a Receive-Message Event at $P_0$

Let $\rho = \text{RECEIVE}(0,1,m)$. We say that event $\rho$ is affected by processor $P_j$ (with respect to $\text{path}_{P0}$) if the following two conditions hold:

1) Processor $P_j$ lies right of $P_0$ on $\text{path}_{P0}$. (i.e. $0 < j \leq s - 1$.)

2) For every prefix $\alpha$ of an execution sequence, if $\alpha$ contains $\rho$ then $\alpha$ (also) contains an event which occurs in $P_j$.

Clearly, if $\rho$ appears in some execution sequence (of A on $\text{path}_{P0}$) then $\rho$ is affected by $P_1$ (on $\text{path}_{P0}$). We remind the reader that symmetric definitions can be introduced for the "opposite direction" (i.e. substituting left by right, $i + 1$ by $i - 1$ etc.). This holds also with respect to the following definitions and facts. The notion of affecting is now used to specify the termination of the consultation algorithm.

Termination of a Consultation Algorithm

An execution of a consultation algorithm is terminated only after processor $P_0$ has received messages affected by both end-points (of the path). (i.e., every
Lemma 3. Suppose that $P_f$ first-right-affects $P_0$ with respect to $path_x$.

1) Let $path_y$ be a path in which $P_j$ lies to the right of $P_0$ ($j \neq y - 1$). Then $P_0$ is first-right-affects by $P_j$ with respect to $path_y$.

2) Let $\beta$ be a prefix of an execution sequence on $path_y$. Suppose that $\beta$ contains a RECEIVE($0,m$) event, but does not contain a RECEIVE($0,m - 1$) event. Then $P_j$ lies to the right of $P_0$ on $path_y$ (i.e. $j > y - 1$).

Proof:

1) Assume on the contrary, that there exist a prefix $\beta$ of an execution sequence on $path_y$ such that $\rho$ = RECEIVE($0,1,m$) is the only RECEIVE($0,\ldots$) event in $\beta$ and that no event occurs in $P_j$ during $\beta$. By Lemma 1, without loss of generality, we may assume that no event of $\beta$ occurs in a processor which lies to the right of $P_j$.

By our assumption that $\rho$ is the only receive-message event in $\beta$ and by using Lemma 1 again, we may also assume that no events (of $\beta$) occur to the left of $P_0$ (on $path_y$).

It follows that $\beta$ is also a prefix of an execution sequence of $A$ on $path_y$ - contradicting our assumption that $P_f$ first-right-affects $\rho$ on $path_y$. Part 1 of the Lemma follows.

2) By similar arguments Part 2 also follows.

Imposing the First Phase

We now start partitioning each execution sequence of algorithm $A$, into phases, in particular we define the first phase.

The First Expanding Phase

Let $H$ denote some execution sequence of $A$ on $path_y$. The first expanding phase of $H$ (denoted $ExpH_1(H)$) is defined as the shortest prefix of $H$ which contains a message-receive event of processor $P_0$. By the above discussion
concerning \( A \)'s termination there exist such a prefix without loss of generality, the last event in \( \text{ExPh}_1(H) \) is \( \text{RECEIVE}(0,1) \).

**Notation:** Consider an execution sequence, \( H \), (of \( A \) on \( \text{path}_2 \)) in which the last event in \( \text{ExPh}_1(H) \) is of the form \( \text{RECEIVE}(0,1) \). Let \( R_1 \) be the rightmost processor which first-right-affects \( P_0 \) with respect to \( \text{path}_2 \).

Though, \( R_1 \) was defined with respect to \( \text{path}_2 \) - we will shortly show that it is independent of it. Namely,

**Corollary 4:** Let \( R_1 \) be as above. Let \( H' \) be an execution sequence of \( A \) on \( \text{path}_2 \) such that the last event in \( \text{ExPh}_1(H') \) is of the form \( \text{RECEIVE}(0,1) \).

1. \( \text{ExPh}_1(H') \) contains an event which occurs in \( R_1 \). Furthermore, \( \text{ExPh}_1(H') \) contains events which occur at each processor that lies between \( P_0 \) and \( R_1 \) (on \( \text{path}_2 \)).

2. Without loss of generality, \( \text{ExPh}_1(H') \) does not contain an event which occurs in a processor which lies to the right of \( R_1 \) (on \( \text{path}_2 \)).

3. \( R_1 \) lies to the right of \( P_0 \) on \( \text{path}_2 \).

**Proof:** In case \( x = y \) the Corollary follows from Lemma 2. In case \( x \neq y \) using Lemma 3, the Corollary follows.

Let \( H' \) be an execution sequence \( H' \) of \( A \) on \( \text{path}_2 \) such that \( R_1 \) lies to the left of \( P_0 \) on \( \text{path}_2 \). By Corollary 4, we have

- The last event in \( \text{ExPh}_1(H') \) is not of the form \( \text{RECEIVE}(0,1) \).

By the guaranteed termination of \( A \) the last event in \( \text{ExPh}_1(H') \) is of the form \( \text{RECEIVE}(0,n-1) \). Let \( L_1 \) denote the leftmost processor in which events occur during every such \( H' \) (for which \( \text{ExPh}_1(H') \) contains a \( \text{RECEIVE}(0,n-1) \) event).

By symmetric arguments (to the ones used in the above proofs) we conclude that:

Let \( \text{Corollary 4} \) Consider an execution sequence \( H' \) of \( A \) on \( \text{path}_2 \) such that \( L_1 \) of \( A \) lies to the right of \( P_0 \) on \( \text{path}_2 \). Then the last event in \( \text{ExPh}_1(H') \) is not of
(the form \( \text{RECEIVE}(0,n-1) \)).

Let us denote by \( r_1 \) the number of links between \( P_0 \) and \( R_1 \) and by \( l_1 \) the number of links between \( L_1 \) and \( P_0 \). We get

**Corollary 5.** \( l_1 + r_1 = n \).

**proof:** Immediate by Corollaries 4 and 4\(^2\).

**Definition:** The **first interval** of the execution sequence \( H \) (denoted \( I_1(H) \)) consists of all the processors in which events occur during \( \text{ExPh}_1(H) \) and all the links between these processors. Let \( \text{left}I_1(H) \) denote the leftmost processor in \( I_1(H) \), and \( \text{right}I_1(H) \) denote the rightmost processor in \( I_1(H) \). Let us denote the number of links in the first interval by \( n_1(H) \).

**Lemma 6:** There exist two execution sequences \( H_B \) (on \( \text{path}_B \)) and \( H_B' \) (on \( \text{path}_B \)) such that:

1) \( \text{left}I_1(H_B) = P_0 \) and \( \text{right}I_1(H_B) = R_1 \).
2) \( \text{left}I_1(H_B') = L_1 \) and \( \text{right}I_1(H_B') = P_0 \).

**proof:** Let us consider an execution sequence \( H_B \) such that \( \text{ExPh}_1(H_B) \) ends with a \( \text{RECEIVE}(0,1) \) event. By Corollary 4, \( I_1(H_B) \) contains all the processors which lie between \( P_0 \) and \( R_1 \). By Corollary 4\(^2\) we may assume, without loss of generality, that no events occur in processors which lie to the right of \( R_1 \). We postpone all events in \( \text{ExPh}(H_B) \) which occur to the left of \( P_0 \). [This can be done still maintaining the partial order of the execution which corresponds to \( H_B \).] The case in which \( \text{ExPh}_1(H_B) \) ends with a \( \text{RECEIVE}(0,n-1) \) event is symmetric. **Lemma 6** follows.

**The First Stabilizing Phase**

Let \( H \) denote some execution sequence of \( A \) on \( \text{path}_A \). The **first stabilizing phase** of \( H \) (denoted \( \text{StPh}_1(H) \)) is empty and takes place at the end of the first
expanding phase.

\textbf{Lemma 7:}

1) The joint message complexity of the first expanding phase of $H$ and the first stabilizing phase (of $H$) is at least $2 n_s(H)$.

2) There exist a $y$ and an execution sequence $H'$ on path$_y$ such that $n_s(H') = \frac{n}{2}$.

\textbf{proof:}

1) Without loss of generality, assume that a $\rho = \text{RECEIVE}(0,1,m)$ is the last event in $\text{ExPh}_y(H)$. Note that a necessary condition for the occurrence of $\rho$ is that for every processor $P_k$ ($1 \leq k \leq r_0$) an event of the form $\text{RECEIVE}(i-1,1,)$ appears in $\text{ExPh}_y(H)$. [Otherwise (by Lemma 1) all the events, which occur in processors which lie to the right of $P_{r_0+1}$, can be postponed - contradictory to our assumption that $P_i$ first-right-affects $P_0$.]

On the other hand, a necessary condition for an event to occur in processor $P_k$ ($1 \leq k \leq r_0-1$) during $\text{ExPh}_y(H)$ is that an event of the form $\text{RECEIVE}(i,1-1,)$ occurs in it. [By recalling the initialization convention in subsection 3.1.]

Part 1 follows.

2) Without loss of generality, $i > r_1$. Let $H'$ be an execution sequence on path$_y$, where at least $r_1$ processors lie to the right of $P_0$ in path$_y$ (i.e., $y = 1 > r_1$). We can reschedule so that $\text{RECEIVE}(0,1,)$ is the last event of $\text{ExPh}_y(H')$.

(Rescaling is never needed in case $i + r_1 = n$.) By Lemma 6, $n_s(H') = r_1$.

Using Corollary 5, Part 2 follows.

3.3 The Notions of Traversable and Enabling and Imposing the Later Phases

The notions defined in this subsection will be used to impose the second and later phases on executions of the algorithm $A$. 

Technion - Computer Science Department - Technical Report CS0355 - 1985
Informal Discussion of the Proposed Notions

We consider a prefix $a$ of an execution sequence and an interval $I$. We say that $I$ is traversable after $a$ if $a$ can be extended by events which occur in $I$ and through which a message passes from one endpoint of $I$ to processor $P_0$. A processor $P_j$ is said to enable $I$ after $a$ if every extension of $a$, at the end of which $I$ is traversable, contains events which occur in $P_j$.

The notion of traversability captures the behavior of consultation algorithms. In such algorithms the intervals, of processors which have already participated in the consultation, are being expanded through "coordination" with $P_0$.

Definition: Interval Traversable after a Prefix

Let $I$ be an interval on path$_a$ containing $P_0$, and $P'_1 \neq P_0$ denote the right end-point of $I$ ($P_1$ lies to the right of $P_0$ on path$_a$). Let $a$ be a prefix of an execution sequence of $A$ on path$_a$. We say that $I$ is right-to-left traversable (Rtol-traversable) after $a$ (on path$_a$) if there is a sequence of events $\beta$ such that:

1) The sequence $a\beta$ is a prefix of an execution sequence (of $A$ on path$_a$). Here and throughout this section, $a\beta$ denotes the sequence formed by the events of the sequence $a$ followed by the events of the sequence $\beta$.

2) The sequence $\beta$ contains only events which occur in processors that belong to the interval $I$.

3) For every $k$, $1 \leq k \leq |I|$, the sequence $\beta$ contains an event of the form RECEIVE($k-1,k$).

Similarly, one can define the notion of a left-to-right traversability (Ltol-traversability). We say that $I$ is traversable after $a$ (on path$_a$) if $I$ is either Rtol-traversable or Ltol-traversable after $a$ (on path$_a$).

Definition: Nothing can occur in an Interval after a Prefix

Let $I$ be an interval on path$_a$ containing $P_0$, and $P'_1$ denote the right end-point of
Let $a$ be a prefix of an execution sequence of $A$ on $\text{path}_{\alpha}$, containing no events which do not occur in $I$. We say that no event can occur in $I$ after $a$ (on $\text{path}_{\alpha}$) if there exist no non-empty sequence $a'$ such that $aa'$ is a prefix of an execution sequence (on $\text{path}_{\alpha}$) containing only events which occur in $I$.

**Definition: A Processor Enables an Interval-Traversal**

Let $0 \leq i < x - 1$ and $P_i$ be the right-end point of an interval $I$ which lies in $\text{path}_{\alpha}$. Let $a$ be a prefix of an execution sequence (of $A$ on $\text{path}_{\alpha}$). Suppose that $a$ contains only events which occur in $I$, and that no event can occur in $I$ after $a$.

Processor $P_j$ is said to right-enable $I$ after $a$ on $\text{path}_{\alpha}$ if the following holds. For every sequence of events $\beta$ which satisfies following four conditions (1, 2, 3 and 4 below), $\beta$ contains an event which occurs in $P_j$. The four conditions are:

1) The sequence $a\beta$ is a prefix of an execution sequence (of $A$ on $\text{path}_{\alpha}$).

2) The sequence $\beta$ does not contain events which occur in processors that lie left of $I$.

3) $\beta$ contains a RECEIVE($i, i+1, m$) event.

4) $I$ is traversable after $a\beta$ on $\text{path}_{\alpha}$.

Note that the third condition is implied by the others in case $i > 0$, and that the fourth condition is satisfied vacuously in case $i = 0$.

**Some Facts Concerning the Notion of Enabling**

**Lemma 8** Let $I$ be an interval containing $P_0$, in $\text{path}_{\alpha}$. Let $P_i$ be the right-end point of $I$ and $0 \leq i < x - 1$. Let $a$ be a prefix of an execution sequence on $\text{path}_{\alpha}$ such that no event can occur in $I$ after $a$ on $\text{path}_{\alpha}$. Let $\beta$ be a sequence of events, which do not occur to the left of $I$, such that $I$ is traversable after $a\beta$ on $\text{path}_{\alpha}$.

1) If $P_j$ right-enables $I$ after $a$ on $\text{path}_{\alpha}$ then $\beta$ contains an event which occurs in $P_k$, for every $1 \leq k \leq j$. 


2) If \( j < k \leq z - 1 \) and \( P_j \) does not right-enable \( I \) after \( \alpha \) on \( \text{path}_y \), then there exist a sequence of events \( \beta \) such that \( I \) is traversable after \( \alpha \) (on \( \text{path}_y \)) and, for every \( k \) (\( j < k \leq z - 1 \)), \( \beta \) does not contain an event which occurs in \( P_k \). Furthermore, \( \beta \) does not contain an event which occurs to the left of the left endpoint of \( I \).

proof: Immediate by Lemma 1 and the definition of right-enabling.

The notions of traversability and right-enabling defined above are path dependent. We show now that the notion of right-enabling is in fact path independent, as long as the path contains the interval \( I \) in one piece. [Trivially, the notion of traversability is also path independent for such paths.]

Lemma 9: Let \( I, P_i, P_j, \alpha \) and \( \beta \) be as Lemma 8. Suppose that \( P_j \) right-enables \( I \) after \( \alpha \) on \( \text{path}_y \). Assume that \( I \) is also an interval in \( \text{path}_y \) (i.e., that the left endpoint of \( I \) lies to the right of \( P_y \) and the right endpoint of \( I \) lies to the left of \( P_{y-1} \)).

1) Let \( \text{path}_y \) be a path in which \( P_i \) lies to the left of \( P_j \) (\( j > y - 1 \)). Then \( P_j \) right-enables \( I \) after \( \alpha \) on \( \text{path}_y \).

2) If \( I \) is traversable after \( \alpha \) on \( \text{path}_y \) and no event occurs during \( \beta \) to the left of \( I \) then \( P_i \) lies to the left of \( P_j \) on \( \text{path}_y \) (i.e., \( j > y - 1 \)).

Discussion: Lemma 9 states that shifting the location of the faulty link does not cancel the right-enabling power of \( P_j \) on \( I \) after \( \alpha \), as long as the faulty link is not shifted between \( P_i \) and \( P_j \). On the other hand, shifting the faulty link in between \( P_i \) and \( P_j \) does not allow to traverse \( I \) (after \( \alpha \)) unless processors on \( P_i \)'s left-hand side are active during \( \beta \).

proof (of Lemma 9):

1) Assume that there exist a \( \beta \) such that \( I \) is traversable after \( \alpha \) on \( \text{path}_y \) but no event occurs in \( P_j \) during \( \beta \). By Lemma 1, without loss of generality, we may assume that no event occurs in a processor which lies to the right of \( P_j \).

By Lemma 9, we may assume (w.l.o.g.), that \( \beta \) does not contain events
Note: which occur left of I's left end-point.

All \( A \)'s also are a prefix on an execution sequence on \( path_\alpha \). In contradic-
tion to our assumption that \( M_I \) right-enables \( I \) after \( \alpha \) on \( path_\beta \). Part 1 of the
Lemma follows.

2) By similar arguments Part 2 also follows.

2.3) Implying the Latter Phases

We now proceed in partitioning execution sequences of \( A \) into phases. All

consistions, facts and proofs should be thought as being repeatedly read, each time

for an incremented value of \( t \); the first "reading pass" refers to the value \( t=1 \), the

next for \( t=2 \) etc.

Definition: A Dummy Phase

Consider an execution sequence \( H \) (of \( A \) on \( path_\beta \)). Let \( \alpha \) denote the prefix of

which consists of all events until the end of the \( t \)-th stabilizing phase. Note that

\( \alpha \) does not have a \( path_\alpha \) (i.e. \( path_\alpha \) equals \( B_\alpha \) --- \(- left_1(H) \) --- \(- right_1(H) \) ---

\( \alpha \)). Either events can occur in \( L(H) \) after \( \alpha \) on \( path_\alpha \) or not. In case events can

not occur in \( L(H) \) after \( \alpha \) we may assume (without loss of generality) that there

exists a sequence of events \( \beta \) such that:

Case 1: The sequence \( \beta \) is a prefix of \( H \).

Case 2: The sequence \( \beta \) contains only events which occur in processors that lie in

\( \alpha \).

Case 3: Events can not occur in \( L(H) \) after \( \alpha \) (on \( path_\alpha \)).

The sequence \( \beta \) is called the \( t \)-th dummy phase of \( H \) (denoted \( D\alpha \beta \alpha \) \( H \)). In case

no events can occur in \( L(H) \) after \( \alpha \) (on \( path_\alpha \)), the \( t \)-th dummy phase of \( H \) is said

to be empty (and takes place after \( \alpha \)).
Notation: Let $\text{AllPh}_i(H)$ denote the empty prefix of $H$, and $\text{AllPh}_i(H)$ denotes $\text{AllPh}_i(H)\text{ExPh}_i(H)\text{Sum}_i(H)\text{DePh}_i(H)$, $i \leq 1$.

Definition: The $(i+1)$-st Expanding Phase

Let $H$ be an execution sequence on $\text{path}_a$ such that $l_i(H) \neq \text{path}_a$. Recall that by the Termination Condition (see subsection 3.2), $l_i(H)$ must be enabled after $\text{AllPh}_i(H)$ by a processor not in $S_i(H)$. This suggests the following definition.

The $(i+1)$-st expanding phase of the execution sequence $H$ on $\text{path}_a$ (denoted $\text{ExPh}_{i+1}(H)$) is the shortest event sequence $a$ such that:

1) The sequence $\text{AllPh}_i(H)a$ is a prefix of $H$.

2) $l_i(H)$ is traversable after $\text{AllPh}_i(H)a$ on $\text{path}_a$.

By the above a exists and is not empty. By Lemma 8, we may assume that $a$ contains events which occur only in processors that lie in $l_i(H)$ or on one of its sides.

Without loss of generality, we assume that this is the right-hand side (i.e., that $a$ does not contain events which occur to the left of $\text{left}l_i(H)$). This suggests the following:

Notation: Let $R_i(H)$ be the rightmost processor right-enabling $l_i(H)$ after $\text{AllPh}_i(H)$ on $\text{path}_a$.

Although, $R_{i+1}(H)$ was defined with respect to $\text{path}_a$ - it is independent of it (as shown below).

Corollary 10: Let $H$ and $R_i(H)$ be as above. Let $H'$ be an execution sequence of $A$ on $\text{path}_a$ such that $\text{AllPh}_i(H')$ equals $\text{AllPh}_i(H)$. Suppose that (both $\text{AllPh}_i(H)\text{ExPh}_{i+1}(H)$ and $\text{AllPh}_i(H')\text{ExPh}_{i+1}(H')$ contain no events which occur on the left-hand side of $l_i(H)$). Then:

1) $\text{ExPh}_{i+1}(H')$ contains an event which occurs in $R_i(H')$. Furthermore, $\text{ExPh}_{i+1}(H')$ contains events which occur at each processor that lies between $l_i(H')$'s right end-point and $R_i(H')$ on $\text{path}_a$.

2) Without loss of generality, $\text{ExPh}_{i+1}(H')$ does not contain an event which occur
in a processor which lies to the right of $R_{x+1}(H')$ (on path$_y$).

3) Either $R_{x+1}(H') = P_{x+1}$ or $R_{x+1}(H')$ lies to the left of $P_{x+1}$ (on path$_y$).

**proof:** In case $x=y$ the Corollary follows from Lemma 6. In case $x \neq y$, using Lemma 9 and noting that $R_{x+1}(H) = R_{x+1}(H')$, the Corollary follows. 

Consider an execution sequence $H'$ of $A$ on path$_y$ such that $AllPh(s)(H) = AllPh(s)(H')$ and $R_{x+1}(H')$ lies to the left of $I_i(H')$ on path$_y$. By Corollary 10, without loss of generality, we have

(2) $ExpPh_{x+1}(H')$ does not contain events which occur on the right-hand side of $I_i(H')$ (on path$_y$).

By the guaranteed termination (of A) (in case $I_i(H') \neq path_y$) $ExpPh_{x+1}(H')$ must contain events which occur to the left of $I_i(H')$. Let $L_{x+1}(H')$ denote the leftmost processor left-enabling $I_i(H')$ after $AllPh(s)(H')$ (on path$_y$). By symmetric arguments (to the ones used in the above proofs) we conclude that:

(2a) Consider an execution sequence $H^f$ of $A$ on path$_y$ such that $AllPh(s)(H^f) = AllPh(s)(H'^f)$ and $L_{x+1}(H^f)$ lies to the right of $I_i(H^f)$ (on path$_y$).

Then $ExpPh_{x+1}(H^f)$ does not contain events which occur on the left-hand side of $I_i(H^f)$ (on path$_y$).

Let us denote by $r_{x+1}(H)$ the number of links between the rightmost processor of $I_i(H)$ and $R_{x+1}(H)$; and by $L_{x+1}(H)$ the number of links between the leftmost processor of $I_i(H)$ and $L_{x+1}(H)$. We get

**Corollary 11:** For every execution sequence $H$,

$$r_{i}(H) + I_i(H) + r_{x+1}(H) \leq n.
$$

**proof:** Immediate by Corollaries 10 and 10a.

**Definition:** The \((i+1)\text{-st interval}\) of the execution sequence $H$ (denoted $I_{i+1}(H)$) consists of all the processors in which events occur during $AllPh(s)(H)ExpPh_{x+1}(H)$.
Lemma 12: For every execution sequence $H$ (of $A$ on $\text{path}_H$), there exist two execution sequences $H_R$ (on $\text{path}_H$) and $H_L$ (on $\text{path}_H$) such that:

1) $\text{left}_H(H) = \text{All}_H(H_R)$, $\text{left}_H(H_L) = \text{left}_H(H)$, and $\text{right}_H(H_L) = \text{right}_H(H)$. 

2) $\text{All}_H(H) = \text{All}_H(H_L)$, $\text{left}_H(H_L) = \text{left}_H(H)$ and $\text{right}_H(H_L) = \text{right}_H(H)$. 

Proof: First note that $\text{All}_H(H) = \text{All}_H(H_L) = \text{All}_H(H_L)$ implies that $l_i(H) = l_i(H_L) = l_i(H_R)$, $l_i(H) = L_i(H)$, $L_i(H) = L_i(H_L)$, and $R_i(H) = R_i(H_R)$ = $R_i(H_L)$. Also note that we deal with the case in which $l_i(H) \neq \text{path}_H$. 

By Corollary 10.1 and 10.2, there exist an execution sequence $H_R$ such that Part 1 holds. Similarly also Part 2 holds and the Lemma follows.

Let $E = \ldots$

Similarly to Lemma 7, we get

**Lemma 13**: 

1) The message complexity of the $(i+1)$-st expanding phase of $H$ is at least

$$2|\text{All}_H(H)| - n(H)$$

2) There exist a $y$ and an execution sequence $H'$ on $\text{path}_y$ such that $n_{y,H}(H) \leq n_{y,H}(H') + \frac{n_y - n_{y,H}(H)}{2}$ and $\text{All}_H(H) = \text{All}_H(H')$. 

Proof:

1) Let $r_{ext}$ be such that $P_{ext} = \text{right}_H(H)$; and $r_{rev}$ be such that $P_{rev} = \text{right}_H(H)$. Without loss of generality, assume that a receive-message event $p =$ RECEIVE$(r_{int}, r_{int} + 1, m)$ occurs in $\text{right}_H(H)$ during $E_{\text{All}_H(H)}$. Note that a necessary condition for the occurrence of $p$ is that for every processor $P_k$ in $\text{All}_H(H)$, an event of the form RECEIVE$(e, 1, r_{int})$ appears in $E_{\text{All}_H(H)}$. Otherwise (by Lemma 1) all the events, which occur in processors which lie to the right of $P_{k-1}$, can be postponed – contradictory to our
assumption that $R_{e_1}(H)$ right-enables $l_1(H)$ after $\text{null}(H)$ (on path$_a$.)

On the other hand, a necessary condition for an event to occur in processor $P_k$ ($\text{int}+1 \leq k \leq z-1$) during $\text{Ex}P_{e_1}(H)$ is that an event of the form $\text{RECEIVE}(x,k-1)$ occurs in it. [By recalling the initialization convention in subsection 3.4.1 Part 1 follows.

2) Without loss of generality, $l_{e_1}(H)+x_{e_1}(H)$. Let $H_R$ be an execution sequence (on path$_B$) as in Lemma 12. It follows that $r_{e_1}(H_R)-r_{e_1}(H_R) = x_{e_1}(H_R)$. Using Corollary 11, Part 2 follows.*

**Notation:** Let $n_{e_1}(H)$ denote the number of links between left$l_{e_1}(H)$ and $P_0$; $m_{e_1}(H)$ denotes the number of links between $P_0$ and right$n_{e_1}(H)$.

**The $(i+1)$st Stabilizing Phase**

Let $H$ denote some execution sequence of $A$ on path$_A$ such that $l_1(H)$ does not equal path$_A$. Then the following holds:

**Lemma 14:** There exists an execution sequence $H'$ (of $A$ on path$_A$) and a sequence of events $\beta$ such that

1) $\text{null}(H')\text{Ex}P_{e_1}(H') = \text{null}(H')\text{Ex}P_{e_1}(H')$.

   [This implies $l_{e_1}(H') = l_{e_1}(H')$.]

2) $\text{null}(H')\text{Ex}P_{e_1}(H')\beta = \text{prefix of } H'$.

3) $\beta$ contains only events which occur in processors that lie in $l_1(H')$ (i.e., $l_1(H')$).

4) $\beta$ contains at least $n_{e_1}(H)$ receive events, in case $\text{Ex}P_{e_1}(H')$ does not contain events which occur to the right of $l_1(H')$; and $\beta$ contains at least $m_{e_1}(H)$ receive events, in case $\text{Ex}P_{e_1}(H')$ does not contain events which occur to the left of $l_1(H')$.

**proof:** First recall that $\text{null}(H')$ equals $\text{null}(H')$ implies $l_1(H')=l_1(H')$ and $n_{e_1}(H')=n_{e_1}(H')$. By the discussion preceding the lemma, $l_1(H)$ is traversable after $\text{null}(H')\text{Ex}P_{e_1}(H')$ (on path$_A$). Assume that $\text{Ex}P_{e_1}(H')$ does not contain events
which occur to the left of $I_i(H)$. Recalling the definition of traversability, we conclude that there exist a sequence $\beta$ satisfying conditions 2 and 3, and containing a RECEIVE($k,k+1$) event, for every deck $\eta_{r_1}(H) = 1$. The Lemma follows.

The above suggests the following

**Definition**: The $(i+1)$-at stabilizing phase of $H$ (denoted $S_{\eta_{r_1}}(H)$) is the longest event sequence $\beta$ such that:

1) The sequence $\exists \eta_{r_1}(H) \exists \eta_{r_1+1}(H) \beta$ is a prefix of $H$.

2) The sequence $\beta$ contains only events which occur in processors which lie in the interval $I_i(H)$.

**Notation**: Let $n_i'(H)$ denote $n_{L_i}(H)$, in case $\exists \eta_{r_1}(H)$ does not contain events to the right of $I_i(H)$; and $n_{r_1}(H)$ otherwise.

**Note**: that the $(i+1)$-st stabilizing phase of $H$ may be empty, but on the other hand, there exists an $H'$ as in Lemma 14. Thus, without loss of generality

**Corollary 15**: The message-complexity of $S_{\eta_{r_1}}(H)$ is greater or equal to $n_i'(H)$.

### 3.4 Complexity Calculations

In this subsection we derive the lower bound using an adaptive adversary sequence and calculations based on Lemma 7, 13 and 14. Let us denote by $l(H)$ the number of triples of phases defined upon the execution sequence $H$. That is, $\exists \eta_1(H)$ is a proper prefix of $H$ and $\exists \eta_1(H)$ equals $H$. In other terms $I_i(H) < \text{path}_b$ and $I_i(H) \geq \text{path}_b$, where $H$ is an execution sequence of $A$ on $\text{path}_b$.

The adaptive adversary sequence consists of subsequences exhibiting different expanding policies. First we introduce an execution sequence $H$ which exhibits a minimal expanding policy (i.e. at each expanding phase, $i$, $n_i(H) = n_{r_1}(H) =$
3. By $\forall H', \forall P_{\text{sys}}(H') = \forall H', \forall P_{\text{sys}}(H)$, Claim $\ref{claim:adaptive}$ holds also with respect to $H'$.

Using Lemma $\ref{lemma:adaptive}$, we get an execution sequence $H''$ such that $\forall H', \forall P_{\text{sys}}(H') = \forall H', \forall P_{\text{sys}}(H)$ and $\forall H', \forall P_{\text{sys}}(H') = \forall H', \forall P_{\text{sys}}(H)$.

The message complexity of $\forall H', \forall P_{\text{sys}}(H')$ is greater or equal to $n_{p}(H')$.

3. By $\forall H', \forall P_{\text{sys}}(H') = \forall H', \forall P_{\text{sys}}(H)$, Claim $\ref{claim:adaptive}$ holds with respect to $H''$ and the message complexity of $\forall H', \forall P_{\text{sys}}(H'') = \mathcal{O}(n_{p}(H''))$.

Hence, the adaptive adversary follows with respect to $H''$. The Lemma follows.

Lemma $\ref{lemma:adaptive}$ has introduced an adversary which tries to minimize the expansion at each expanding phase. The sum of the $n_{i}(H)$'s which result from this adversary is $\mathcal{O}(n \log n)$ (as the objective function $\sum_{i=1}^{n} n_{i}(H)$, subject to the constraints $n_{i}(H)$). The Lemma would have yield the desired $\Omega(n \log n)$ lower bound, in case

$$\sum_{i=1}^{n} n_{i}(H) = \Omega\left(\sum_{i=1}^{n} n_{i}(H)\right).$$

However, this may not be the case (in fact one may show that under the above constraints, $\sum_{i=1}^{n} n_{i}(H) = \Theta(n \log \log n)$). Thus, a more involved adversary argument is required.

We now introduce the adaptive adversary sequence $H$. Intuitively, this sequence adopts the expanding policy introduced by Lemma $\ref{lemma:adaptive}$ (i.e. expands to the smaller distance) during phases at which the interval is of size less than $\frac{n}{2}$ or bigger than $\frac{15n}{16}$. During the other phases (called the biased phases) a biased expansion policy is adopted. This policy either creates a more balanced (w.r.t. the position of $P_{0}$) interval or causes many small expansions to the more distant direction. This is done by preferring to expand to the direction closer to $P_{0}$ unless
The expansion is \( k = O(\log n) \) times larger than the expansion suggested to the other direction, by the algorithm. That is, at the \( i \)-th biased phase, a right expansion takes place if \( r_i(H) \leq \frac{1}{k-1} \cdot 4_i(H) \); and a left expansion takes place otherwise.

For simplicity, we write \( n_i \) instead of \( n_i(H) \), \( n'_i \) instead of \( n'_i(H) \), \( n_i \) instead of \( n_i(H) \), \( n'_i \) instead of \( n'_i(H) \), and \( t \) instead of \( t(H) \).

The Adaptive Adversary (for phase \( i, 1 \leq i \leq t \))

If \( n_{i-1} < \frac{n}{2} \), then apply the minimizing expansion policy (suggested by Lemma 18).

It follows that \( n_i \leq n_{i-1} + \frac{n - n_{i-1}}{2} \).

Let \( p \) be the smallest integer such that \( n_p \geq \frac{n}{2} \).

If \( n_p \geq \frac{n}{2} \), then (consider the following two cases):

**Proof.**

Case 1: Both \( n_p \) and \( n'_p \) are greater than (or equal to) \( \frac{n}{4} \); then continue with the minimizing expansion policy.

Case 2: Without loss of generality, \( n_p < \frac{n}{4} < n'_p \).

Let \( m, n = n' - n_p \) and \( k = \log \frac{n}{m} \).

Case 1: If \( n - n'_p > \frac{n}{4} \), then expand to the left.

Case 2: Otherwise apply the minimizing expansion policy.

(Note that in this case, \( n_p - n_p' \leq \frac{n - n_{i-1}}{k} \).)

End of case 2
Similarly to Lemma 16, we have

Lemma 17: There exists an execution sequence $H'$ (of $A$ on some path $p$) such that

1) The sequence of $m_i(H')$, exists $(H')$, results from the above adaptive adversary.

2) The message complexity of $H'$ is at least

$$2n + \sum_{i=1}^{t(H')} m_i'(H')$$

We now demonstrate an $\Omega(n \cdot \log n)$ lower bound on the sum $\sum_{i=1}^{t(H')} m_i'(H')$.

Lemma 17: Under the above adaptive adversary,

$$\sum_{i=1}^{t(H')} m_i'(H') = \Omega(n \cdot \log n)$$

Proof: Let us again simplify our notation by omitting $H'$ from it. Let $p$ and $m$ be as defined by the adaptive adversary (i.e., $p$ is the smallest integer such that $n_p > \frac{n}{2}$, and $m = n - n_p$). Note that $n_p < \frac{3m}{4}$ and $m > \frac{n}{4}$ (since $n_p \leq n_{p-1} + \frac{n - n_{p-1}}{2}$ and $n_{p-1} < \frac{n}{2}$).

Case 1: $n_p$ and $m_p$ are both larger than (or equal to) $\frac{n}{4}$.

As the minimizing expansion policy is adopted for each phase $i$, $p < i \leq t$, we have at least $\log_2(m - n_p)$ such phases (at each such phase $n - n_i > \frac{n - n_{i-1}}{2}$).

Therefore, $\sum_{i=1}^{t(H')} m_i'(H') > \frac{n}{4} \cdot \log \frac{n}{4} = \Omega(n \cdot \log n)$.

Case 2: Without loss of generality, $n_p < \frac{n}{4} < m_p$.

Let $q$ be the largest integer $i$ such that $n - n_{i-1} > \frac{m}{4}$ (i.e., the $q$-th phase is the last phase in which the biased policy is adopted). Consider the phases in which the
A biased policy is adopted, i.e. consider $\epsilon$ such that $p < \epsilon < q$.

Right expansion takes place in phase $i$ (only) whenever $r_i < \frac{1}{k-1} l_i$.

Let $h$ denote the number of times a right expansion takes place during the biased phases, and recall that $k = 4 \log n$.

**Subcase 2.1:** $h > \frac{k}{4}$.

Note that each of the $h$ right expansions is charged by at least $\frac{n}{4}$ in the corresponding stabilizing phase (since $nr_{i-1} \geq nr_p = n_p - n_p > \frac{n}{2} - \frac{n}{4}$).

The charge for these phases is thus at least

$$\frac{n}{4} h \geq \frac{k}{4} \cdot \frac{n}{4} \cdot \log n = \Omega(n \log n).$$

**Subcase 2.2:** $h \leq \frac{k}{4}$.

We claim that during the biased phases the interval is expanded to the left by at least $\frac{m}{2} = \Omega(n)$. The reason being that each right expansion (occurring at a phase $i$, $p < i < q$) is bounded by

$$\frac{1}{k} (n - n_{i-1}) \leq \frac{1}{k} (n - n_p) = \frac{m}{k}$$

since $r_i = \frac{1}{k} n_{i-1}$. The sum of expansions to the right (i.e. $\sum_{i=p+1}^q (nr_{i-1} - nr_{i-1})$) is thus bounded by

$$\frac{m}{k} \leq \frac{1}{4} m.$$ 

By recalling that the size of part outside the interval is $m$, before the biased phases, and at most $\frac{m}{4}$ after the biased phases, the claim follows.

**Proof:**

- Case 1: $m = \frac{m}{4}$ after the biased phases, and at most $\frac{m}{4}$ after the biased phases.

We conclude by noting that after the biased phases, the size of the part outside the interval, $n - n_q$, is at least $\frac{m}{4k}$ (since $n - n_q \geq \frac{1}{k} (n - n_q)$ and
As the minimization policy is applied for all phases following
the q-th phase, there exist at least \( \log_2 \frac{m}{4k} \) such phases. By the above Claim, the stabilizing phase corresponding to each of these later phases is charged by at least \( \frac{n}{9} < \min \{ \frac{m}{2}, \frac{n}{4} \} \) (recall \( m > \frac{n}{4} \)). Thus,

\[
\sum_{i=1}^{t} n_i \geq \sum_{i=1}^{t} n_i' > \frac{n}{9} \log_2 \frac{m}{4k} = \Omega(n \log n).
\]

The Lemma follows. 

Combining Lemmas 15 & 17, we get

The Main Theorem (Theorem A):

Let \( A_n \) be a consultation algorithm for \( n \)-processors safe rings.

The message complexity of \( A_n \) is \( \Omega(n \log n) \).

3.5 On the Tightness of the Lower Bound for the case in which the Number of Processors is Known

In this subsection, we demonstrate the tightness, up to a multiplicative con-
stant, of the above proven lower bound for the case in which the number of proces-
sors is a-priori known. Tightness is demonstrated by presenting a variant of the
consultation algorithm of Shira and Rodol [SR]. On input \( n \), this variant achieves
consultation on any \( n \)-processors safe ring at the cost of \( n \log n + O(n) \) message transfers. It is interesting to note that the improvement (by a multiplicative fac-
tor of 2) has occurred to us through a closer examination of our (lower bound)
proof technique.

For the sake of simplicity, a special version of the consultation algorithm
(henceforth referred to as the C-Algorithm) is presented. The C-algorithm imple-
ments collecting acknowledgements, which is a special case of consultation (see
section 2.3). It is easy to modify this version so that any consultation is
implemented maintaining the complexity of the algorithm.

The C-algorithm can be viewed as proceeding in phases. Each "covering" phase reduces by half the "yet uncovered" part of the ring (the covered part of the ring is the interval of processors from which acknowledgements have already been collected). The "covering" phases are interleaved by "sharing information" phases, in which information is passed among the end-points of the "currently covered" part of the ring. After \( \log n \) phases the entire ring is already covered.

At each expanding phase the end-points of the "currently covered interval" initiate a "covering message" towards the midst processor of the uncovered part. This message is sent through the uncovered part of the ring. When the midst processor receives a "covering message" directed to it, it becomes the new end-point of the covered interval. Upon becoming a new end-point, a processor notifies both old end-points of the old covered interval (by sending a message through the new covered interval), and initiates a new covering phase. Upon being notified of a new end-point, the old end-point closer (in the covered interval) to it ceases to act as an end-point; while the other (old) end-point initiates a new expanding phase. Processors, other than the current midst and end-points, forward the messages in the required direction (i.e., either from the initiator to the designated midst or from the midst to the initiators). In case a processor receives "covering messages" from both sides, it initiates a special interrupt message which is sent in both directions (i.e., an announcement). No processor forwards more than one interrupt message in each direction.

The reader can easily verify that:

1) If the ring contains a fault which persists then the execution terminates after \( \log n \) covering phases and all the processors have been covered.
2) If the faulty link alternatingly loses and transfers messages then it may be in the currently covered interval. In this case, the "sharing information messages" may not get through. However, then it is guaranteed that some processor would get covering messages from both directions and initiate an interrupt message. Thus, also in this case the algorithm terminates after covered covering phases and all processors have been covered.

In order to establish the upper bound on the message complexity of the C-algorithm, the reader is referred to the Appendix. Loosely speaking, let \( n_i \) denote the number of "covered" links after the \( i \)-th covering phase. Note that \( n_0 = 0, n_1 \leq \frac{n}{2}, \) and \( n_{i+1} \leq \frac{n_i}{2^i}, \) for \( 1 \leq i \leq \log \frac{n}{2} \). The message complexity of all the covering phases is \( \frac{\log n}{\log 2} \). The message complexity of the "sharing information" phases is \( n_{\log n} \). The complexity of an interrupt message is at most \( n + 1 \). Thus, the overall message complexity of the C-Algorithm is at most \( 3n + \frac{n}{\log n} \log n + 3.5n \).

This leads to the following theorem.

**Theorem B** There exist a uniform consultation algorithm, that on input \( n \) processors implements consultation on an \( n \)-processor safe ring using at most \( n \log n + \frac{n}{\log n} \log n + 3.5n \) messages.

**Proof**

\[ \text{Details...} \]

The proof is as follows.

### 3.6 Concluding Remarks

**Lower Bound.**

We have proved a tight lower bound on the message complexity of consultation in a faulty ring. The lower bound was demonstrated under a non-uniform complexity measure: the consultation algorithm may be specially designed to operate on an \( n \)-processor ring, still it may require \( \Omega(n \log n) \) messages when running on
this specific network. To the best of our knowledge - our result constitutes the first non-uniformed lower bound which holds only for asynchronous communication. The lower bound fully reflects the nature of the distributed task we have considered, and demonstrates the difficulty of implementing the task in the presence of faults.

We now extend our result to general safe networks. A network is said to be safe if it is guaranteed that during any execution of any algorithm, the graph consisting of all processors and all non-faulty links is connected. Note that there is no point to consider consultation in asynchronous networks which are not safe. Let $G = (V,E)$ be any graph and let $n = |V|$, and $m = |E|$. Suppose that the graph $G$ is Hamiltonian, i.e. $G$ contains a simple circuit on which every vertex of the graph appears only once. Then consultation on a safe network, the underlying graph of which is $G$, may require $m - n + O(n \log n)$ messages. This holds even if the consultation algorithm is specially designed to operate on $G$.

No characterization is known for Hamiltonian graphs, and there is no reason to believe that the underlying graphs of communication networks are more likely to be non-Hamiltonian than Hamiltonian. Furthermore, determining whether or not a given graph is Hamiltonian - is unlikely to be easy even if the given graph is sparse. (The reason being that the problem is NP-Complete. For further details the reader is referred to Even's textbook on graph algorithms [E] and to [GJT] and [W] which present NP-Completeness results for the Hamiltonian problem in restricted classes of sparse graphs.)

We do not believe that the extended lower bound is tight. The gap between this lower bound and the $nm$ obvious upper bound calls for further research. Even presenting a more efficient consultation algorithm for general graphs that contain only one faulty link - constitutes an open problem.
A Lower Bound for the case in which the Number of Processors is Unknown

In this section we prove an $\Omega(n^2)$ lower bound on the message-complexity of consultation on safe rings, when $n$ - the number of processors in the ring - is not a priori known. This implies that the simple consultation algorithm, derived by "securing via announcement" [SR], each message transfer of the "one-round" algorithm, is optimal.

4.1 Proof of the $\Omega(n^2)$ Lower Bound

In the proof of the $\Omega(n^2)$ lower bound on consultation in the case that $n$ is not a priori known, we use some of the ideas and notions introduced in section 3. In this sequel, we briefly present a proof for the $\Omega(n^2)$ lower bound.

Throughout the proof of the $\Omega(n \log n)$ lower bound, we have simultaneously considered $n$ paths corresponding to the $n$ possible locations of the faulty link. Recall that each of these paths consists of $n$ processors. In this section, we simultaneously consider paths of various length, the left endpoint of which is $P_0$.

Clearly, the consultation algorithm - not knowing the number of processors - must succeed on all these paths.

The path-independence of the notions of first-affecting, enabling, $R_i$, and $L_i$ is preserved also with respect to the paths we consider (Compare Lemmas and Corollaries 2, 3, 4, 4', 4", 6, 7, 9, 10, 10', 10", 12, 13, 14, 15, and 18).

The key observation is that, considering paths of different length, one can derive much stronger upper bounds on the growth of the $r_i$'s. Namely, $r_i(H) = 1$, for all $i$ (recall that no processor lies on the left of $P_k$ in any of the paths we consider). [Assume on the contrary that $r_i(H) > 1$. Consider the path consisting of the $(i-1)$-st interval and the first link and processor on its right. By Corollary 4 or 10, there would be no receive event in the right endpoint of the $(i-1)$-st interval.

The algorithm does not terminate on this path, in contradiction to the requirement that it operates on all rings.]

We conclude that \( m_H = 1 \), and the message complexity of \( H \) is at least \( \frac{n-1}{2} \cdot \frac{n}{2} \). This yields

**Theorem C:**

Let \( A \) be a consultation algorithm for safe rings, operating without a-priori knowledge of the number of processors. The message complexity of \( A \) on an \( n \)-processors ring is at least \( \frac{1}{2} \cdot (n^2 - n) \).

### 4.2 On the Possibility of Consultation and the Tightness of the Lower Bound

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, consultation can not be achieved. The reason being that there is no way to detect whether or not the active processors have participated in the consultation. Assume, on the contrary, that there exist an algorithm which achieves consultation 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 the initiator \( P_0 \). These processors only know the identity of processors on the path. Embedding the processors of the above path on one larger ring, we derive a contradiction.

In case each processor a-priori knows the identities of its neighbours, a consultation can be achieved with \( n^2 \) messages using a "link securing" methodology [SR]. A link, in a safe ring, can be secured by announcing each message sent on the link (i.e. instead of sending \( m \) from \( P_i \) to \( P_{i+1} \), the message is sent from \( P_i \) to \( P_{i+1} \), via both possible directions). Using this technique, the simple "one round traversing of the ring" algorithm (implementing collecting acknowledgements in
non-faulty rings) can be secured to operate on safe rings. It is interesting to note
that the resulting algorithm is optimal up to a multiplicative factor of 2.

An algorithm that achieves consultation in \( \frac{1}{2} n^2 + O(n) \) is hereby presented.

For the sake of simplicity, we again refer to the special version of the consultation
algorithm - C-Algorithm).

Processor \( P_0 \) issues an awaking message to each of its neighbors. Upon
receiving, for the first time, an awaking message \( P_i \) sends an awaking message to
\( P_{i+1} \) and an acknowledging message to \( P_{i-1} \), when the awaking message received
at \( P_i \) is sent by \( P_{i+1} \). This acknowledging message consists of the ordered pair
\((ID_{ack}, ID_{next})\), where \( ID_{ack} \) is the identity of \( P_i \) (the acknowledging processor) and
\( ID_{next} \) is the identity of \( P_{i+1} \) (the next processor). For \( j \neq 0 \), when \( P_j \) receives an
acknowledging message from \( P_{j+1} \), it forwards the message to \( P_{j-1} \). Symmetric
behaviour occurs for the opposite direction. Note that in case \( P_j \) receives a
second awaking message it does nothing; and that \( P_0 \) does not forward ack-
nowledging messages.

The reader can verify that the message complexity of this algorithm on a \( n \)-
processor ring is at most \( \frac{1}{2} (n^2 + n + 2) \).

4.5 Implication on the Complexity of Electing a Leader in a Safe Ring

The lower bound on the uniformed message-complexity of consultation
(presented above) implies a similar lower bound for the general problem of electing
a leader, in which the number of processors on the ring is not a-priori known.
The reason being that a leader electing algorithm operating on a safe ring, with an
unknown number of processors, must collect acknowledgements concerning the
leadership from all other processors. (Otherwise, consider an execution of the
leader electing algorithm in which the leader did not get messages from all

Technion - Computer Science Department - Technical Report CS0355 - 1985
processors. Without loss of generality, the links which transfer messages during this execution constitute a simple path. Since the number of processors is not known, we may consider the same execution when taking place on a larger ring. By symmetric arguments and using the assumption that the processors IDs are taken from an arbitrary set, the "larger ring" may contain another path and another leader.

We conclude by noting that remarks similar to those which appear in subsection 3.5 hold here. Namely, the $O(n^2)$ holds also for Hamiltonian sparse graphs.

Acknowledgements

We would like to thank Shimon Even, Niki Rodeh and Nessim Francoz for useful comments regarding both the contents and form of this paper.
Appendix

This Appendix consists of the local programs for the C-algorithm (the collecting acknowledgements algorithm presented in section 3.5) in case $n$ - the number of processors is a-priori known. The algorithm is described using the ring representation of section 2.4.

The messages passing during the execution of the C-algorithm are of one of the following three types: cover, share and interrupt. Processor $P_0$ can be in one of the following local states: start, active, and finish. Each of the other processors (i.e. $P_i$ where $1 \leq i \leq n-1$) can be in one of the following local states: idle, covered and interrupted. The links incident at processor $P_i$ are locally referred to as the first and second links. Each processor (including $P_0$) has two pairs of internal flags. The first pair is: covered through the first link and covered through the second link. The second pair is: endpoint through the first link and endpoint through the second link.

In the description of the C-algorithm we will use the following primitives: $P_i$ sends a message designated to $P_j$ via $P_{i+1}$; and $P_k$ forwards a message designated to $P_j$. These primitives can be implemented, by appending counters to the messages. For the sake of simplicity, we assume that $n = 2^l$, where $l = \log_2 n$.

Initially, processor $P_0$ is in the start state and both its endpoint flags are on. Each other processor ($P_i$ for $1 \leq i \leq n-1$) is in the idle state and both its endpoint flags are off. The cover flags of all processors are off.

Beginning the first covering phase: processor $P_0$ sends cover messages designated to $P_k$ via both possible directions, and sets its local state to active.

Upon receiving a cover message (through its first link) processor $P_i$ ($1 \leq i \leq n-1$) acts as follows:

1) If $P_i$ is in the idle state then it sets its state to covered and sets its covered through the first link flag on.

In case the message was not designated to $P_i$ then $P_i$ forwards it through its second link. In case the cover message is designated to processor $P_j$ then $P_j$ sets its
endpoint through the first link flag on and complete the following two send
actions: Sends a cover_{π+1} message, designated to the processor at distance
\( \frac{π}{2} \) through \( P_1 \)'s second link. Sends a share_{π}, message designated to the
old endpoints, through the first link.

2) If the covered through the first link flag of \( P_1 \) is on then \( P_1 \) proceeds as in (1).

3) If the covered through the second link flag of \( P_1 \) is on then \( P_1 \) sets its local
state to interrupted, sends interrupt messages to both its neighbours and
halts.

[Remark: No cover message ever gets to \( P_0 \)]

Upon receiving a share message: (through its first link) processor \( P_i \) (0 ≤ i < m – 1)
sends the share message through the second link.

1) If the endpoint through the first link flag of \( P_i \) is on then \( P_i \) sets it off and
 sends the share message designated to the processor at distance
\( \frac{π}{2} \) through \( P_i \)'s second link. Here \( π \) is the index of the share message received at
\( P_i \).

2) If neither endpoint flags of \( P_i \) is on then \( P_i \) forwards the share message
through its second link.

3) In addition, processor \( P_0 \) stores its distance to the furthest endpoints
(from which it has received a share message). \( P_0 \) updates these two values
according to the counters of the share messages passing through it. When the
sum of these values amount to \( π + 1 \), processor \( P_0 \) sets its local state to finish
and halts.

Upon receiving a interrupt message: (through its first link) processor \( P_i \)
(0 ≤ i < m – 1) acts as follows:

1) If \( i = 0 \) then \( P_i \) (≠ \( P_0 \)) sets its local state to finish and halts.

2) If \( i ≠ 0 \) then \( P_i \) sets its local state to interrupted, forwards the interrupt
message through its second link and halts.

The reader can verify, by induction on \( π \), that during the execution of the C-
algorithm at most one processor has received a cover message designated to it
(\( 0 ≤ i < m \)). Furthermore, at most one share_{π} message was issued. The upper bound
on the complexity of the C-algorithm follows.