Faults and Fault-Tolerance in Distributed Computing Systems: the Election Problem

A THESIS
Presented to
The Academic Faculty

by

Byungho Yi

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
January 1994

Faults and Fault-Tolerance in Distributed Computing Systems: the Election Problem

APPROVED:

Gil Neiger, Chairman

Gary L. Peterson
Spelman College

Kenneth L. Calvert

Ellen Witte Zegura

James Calvin
School of Industrial & Systems Engineering

Date Approved by Chairman 1/11/94

Acknowledgment

Most of all, I would like to thank Gary L. Peterson for all he has given to me during this research. Not only did he provide immeasurable academic guidance but also understanding of the life of a graduate student. I deeply indebted to Gil Neiger. His exceptional talent for writing greatly enhanced the quality of this dissertation. I also appreciate the interest shown in my dissertation by other members of my committee. A special thanks goes to H. Venkateswaran for his confidence in me and his leadership.

I am grateful to College of Computing for its financial support during the study. Its support staff and excellent facilities have been very helpful.

I have had many officemates over the course of this research who have made coming to work pleasant: Hernan Astudillo, Jorg Liebeherr, Wei Liu, Mark Pearson, Hongyi Zhou, Rida Bazzi, Venkataraman Ramanathan, and Rimli Sengupta. All members of Korean Students Community of College of Computing have also made this journey possible and enjoyable.

I acknowledge the support my parents provided during this study. Their love and patience made all this possible. My wife, Hyeryeong, deserves many thanks for her tolerance and patience during this ordeal. Also, I thank my son, Tacki, for sacrificing many hours to play with his daddy.

Contents

List of Tables

List of Figures

Summary

This dissertation examines some issues concerning fault tolerance in distributed computing systems using the election problem as a test bed. The first problem investigated is the average-case behavior of algorithms for election on asynchronous rings of processors. An algorithm with good worst-case and good average-case message complexity is obtained. It is demonstrated by extensive simulations that the average-case message complexity of the algorithm appears to be very close to the theoretical optimal. The availability of such algorithms is important for practical applications and their existence is interesting since it contradicts the common belief that algorithms with better worst-case message complexity perform less well in the average case.

The impact of inexact knowledge by processors is examined. Specifically, the election problem is considered for asynchronous rings with one possible fail-stop link failure when a lower bound and/or an upper bound on ring size are known to all processors. It is shown that a good lower bound is most useful in designing algorithms with good worst-case message complexity. However, the availability of upper bound is only useful if the upper and lower bounds are sufficiently close. Even a very tight upper bound is not helpful if it is not combined with a good lower bound.

The impact of the additional knowledge of the identifiers of two neighbors is also examined. It is shown that this knowledge affects the solvability of the problem but is not helpful in improving the worst-case message complexity if the problem is solvable without that knowledge.

Tolerating link failures on square meshes of processors is studied, again using the election problem. While conceptually simpler algorithms are obtained using election algorithms on rings, a more sophisticated algorithm with better worst-case message complexity is also obtained for the case with a smaller number of faulty links.

Several interesting open problems are discussed for each issue investigated.

Chapter 1

Introduction

A distributed system is a set of autonomous processors that communicate using a communication network. There are many advantages to distributed systems: Some resources can be shared by many processors (e.g., printers). Computation speed can be improved by load sharing. Furthermore, failures of system components (such as processors and communication links) can be tolerated using redundancy of those components [37].

A distributed system can adapt to failures in two ways. One way is having fault-tolerant software that can operate continuously and correctly even if failures occur. The second alternative is temporarily halting normal operation and reconfiguring the system. This reconfiguring can be managed by a single processor called the "leader". The procedure that elects a leader is called an *election* [18] and is the focus of this dissertation.

The problem of election has been studied extensively since it is one of fundamental problems of distributed computing systems. This dissertation examines some issues concerning fault tolerance in distributed computing systems using the election problem as a test bed.

Brief descriptions of the contents of the chapters in this dissertation are as follows. Chapter 2 gives some definitions that are used through out the dissertation.

Chapters 3 summarizes previous work.

Chapter 4 considers the average-case message complexity (i.e., expected number of messages for an execution of algorithms) of election algorithms for asynchronous rings. It is important for practical applications to have algorithms with good average-case message complexity. It is especially desirable for algorithms to have good average-case *and* worst-case message complexity. Chapter 4 considers the question of whether there exist election algorithms that are optimal (or near optimal) in average-case message complexity and whose worst-message complexity is also near optimal. The existence of such algorithms is very interesting, since it is commonly believed that algorithms with good worst-case message complexity perform worse in the average case [31]. Chapter 4 considers the above question in failure-free asynchronous rings of processors. The question is answered positively by presenting an algorithm whose average-case and worst-case message complexities are both near optimal.

Chapter 5 considers processor's knowledge of ring size on the possibility and complexity of ring election algorithms. The number of messages needed for an execution of a distributed algorithm depends on parameters such as the assignment of the identifiers to processors, characteristics of the communication system (e.g., synchrony and topology), and the local knowledge of processors in distributed systems such as the number of processors in a distributed system. There have been many studies on the effect of these parameters on the message complexity [15, 24, 28, 32]. Chapter 5 considers the case where each processor's knowledge of the number of processors in distributed systems is inexact. Each processor knows a lower bound and an upper bound of the number of processors instead of an exact number. Also, knowledge of identifiers of two neighbors are considered. Asynchronous rings of processors with one

link failure are used as an example. Lower bounds on worst-case message complexity and two asymptotically tight upper bounds are obtained. An impossibility is also presented.

Chapter 6 considers election algorithms for asynchronous square meshes of processors in which some links may fail undetectably. Several cases on the relation of t, the maximum number of faulty links, to the number of processors are considered with assumptions that t and its relation to the number of processors in the network are known to all processors. Several algorithms and an impossibility result are presented for the election problem in such systems.

Chapter 7 gives conclusions and lists several open problems.

Chapter 2

Definitions and a Model

2.1 Distributed Systems

A *distributed system* is a set of processors and a set of communication links that connect them. Each processor has its own computing unit and local memory that are not shared with any other processor. Processors communicate with each other by passing messages along communication links. Since a distributed system can be considered as a network of processors, the term *network* is used instead of distributed system in much of literature.

A communication link can be either *bidirectional* or *unidirectional*. If both processors connected to a communication link can send and receive messages over the link, the link is bidirectional. If one processor can only send messages and the other can only receive them, the link is unidirectional.

Distributed systems are either *synchronous* or *asynchronous*. A distributed system is synchronous if there is an *a priori* known (to all processors) bound on the delivery time for all messages that are delivered. (Some messages might not be delivered in the presence of faults.) If there is no such bound for a distributed system, the distributed system is asynchronous.

The underlying graph of a distributed system is the *topology* of the system. Typical

topologies of distributed systems include rings, complete graphs, and meshes.

A distributed system is said to have a *global sense of direction* if links are labeled to capture some amount of topological information [39]. A ring has a global sense of direction if links are labeled as follows: all links are labeled as "left" and "right". Let p_i, p_j, p_k be any three consecutive processors in a ring, then p_j's "left" link is the "right" link of p_i and p_j's "right" link is the "left" link of p_k. A square mesh has a global sense of direction if a processor can distinguish its four links by its names (such as up, down, left, right) in uniform fashion. Let p_1, p_2, p_3, and p_4 be the processors that are connected to a processor p_i with p_i's "up", "right", "down", and "left" links, respectively. Then, p_i's "up" link p_1's "down" link, p_i's "right" link is p_1's "left" link, p_i's "down" link is p_1's "up" link, and p_i's "left" link is p_1's "right" link.

2.2 Faults

Both processors and links in a distributed system can fail in various ways. Link failures, which are considered in this dissertation, can be *fail-stop*, *intermittent*, or *Byzantine*. The fail-stop failure is the most benign failure type [40]. A failed link stops delivering messages and never delivers messages again. A faulty link of fail-stop failure fails before the start of an execution of an algorithm. The Byzantine failure is most malicious failure type. A failed link can perform any malicious behavior such as altering messages or sending false information. The intermittent failure is more malicious than fail-stop failures but less malicious than Byzantine failures. In the intermittent failure, a failed link stops delivering messages and never delivers messages again like fail-stop failure. But links of intermittent failure can fail at any

time of an execution of an algorithm.

In the presence of faults, distributed algorithms are hard to design. The impossibility result by Fisher, Lynch, and Paterson [14] implies that the election problem is unsolvable on asynchronous systems with one processor failure that may fail during a execution. Link failures are hard to tolerate if communication is asynchronous because failed links cannot be distinguished from slow ones.

2.3 Distributed Algorithms

A *distributed algorithm* for a distributed system consists of n copies (where n is the number of processors in the system) of a deterministic local program, each of which is assigned to one processor in the system. The programs are ordinary sequential programs with *communication statements*. The communication statements of a processor are of the form of "send a message M over the link l" or "receive a message M' from the link l", where l is a link that connects the processor to another processor.

2.4 The Election Problem

Election is the problem of choosing a unique processor from the processors in a distributed system of n processors. Each processor has a unique *identifier* or *id* chosen from a totally ordered set. It is assumed that all processors are identical except for their identifiers.

A distributed algorithm $\mathcal{A}$ solves the election problem if all executions of the programs terminate and the following conditions are satisfied after they do:

- exactly one processor (called a *leader*) in the network is in a distinguished state called *elected*; and
- the identity of the leader is known to all processors connected to the leader.

2.5 Measures

There are several measures for the complexity of distributed algorithms. One of the commonly used measures is *worst-case message complexity*, which is an upper bound on the number of messages sent during any execution of an algorithm.

The maximum number of bits necessary to represent a message is another frequently used measure. In some situations, a trade-off between the number of messages and the size of the largest message is possible by encoding more information into a larger size message, and fewer messages can be sent. Therefore, it is important to try to minimize both measures. In this dissertation, worst-case message complexity is used as a principal measure and size of the longest message is also analyzed.

Besides the worst-case complexity and the message size, the *average-case message complexity* is also of interest. The average-case message complexity of an algorithm is the expected number of messages for an execution of the algorithm. More formally, it is defined as follows. Let $\mathcal{A}$ be an algorithm that is executed on a distributed system of n processors. Let a random variable $\mu_n[\mathcal{A}]$ be the number of messages sent during an execution of $\mathcal{A}$. Let $\boldsymbol{I}_n$ be a subset of inputs to the algorithm. Then average-case message complexity of algorithm $\mathcal{A}$ is defined by

$$\overline{\mu_n}[\mathcal{A}] = \boldsymbol{E}\{\mu_n[\mathcal{A}]\} = \sum_k k \cdot Pr\{\mu_n[\mathcal{A}] = k\},$$

where $\boldsymbol{E}\{\cdot\}$ denotes expected value and $Pr\{\cdot\}$ denotes probability with respect to a probability distribution over $\boldsymbol{I}_n$ [42]. Note that a typical $\boldsymbol{I}_n$ for an asynchronous distributed system is the set of all assignment of identifiers to the n processors.

The following defines more precisely the average-case message complexity of algorithms that elects a leader in an asynchronous rings of processors. For asynchronous deterministic algorithms, the behavior of a processor depends on inputs to the algorithm and the order in which messages are received. While inputs may be fixed, the order in which messages are received may vary in every execution. However, the order in which messages are received by a processor does not vary for unidirectional rings of processors where messages on every link are delivered in FIFO order. Therefore, the behavior of a processor in algorithms that elects a leader in an asynchronous rings processors depends only on the assignments of identifiers. Similarly, the exact number of messages required to elect a leader with algorithm $\mathcal{A}$ depends on permutations of identifiers of all processors in the ring.

Assume that all ring permutations are equally likely. Then,

$$\overline{\mu_n}[\mathcal{A}] = \frac{1}{|\boldsymbol{I}_n|} \sum_k k J_{nk} = \frac{1}{(n-1)!} \sum_k k J_{nk},$$

where J_{nk} is the number of ring permutations of size n in which k messages are exchanged by algorithm $\mathcal{A}$.

2.6 A Model of Distributed Systems

A communication network consists of a set of n processors $P = \{p_1, p_2, \cdots, p_n\}$ and a set of links, each of which connects two processors. A network is modeled as a

graph $G = (V, E)$, where $|V| = n$, each vertex represents a processor, and each edge represents a link between two processors.

Each processor p_i has an unique identifier from a totally ordered set. Every processor p_i also has two buffers ($SendBuffer_i(l)$, $ReceiveBuffer_i(l)$) for every link l that connects one processor to another. It is assumed that all buffers are first-in first-out (FIFO).

An execution of a communication statement "send a message M over link l" by processor p_i results in two *communication events*: a *Message-Send* event that places message M into the $SendBuffer_i(l)$ and a *Message-Transfer* event or a *Message-Loss* event for message M. A *Message-Transfer* event removes a message from $SendBuffer_i(l)$ and places the message into the $ReceiveBuffer_j(l)$ (if link l connects processors p_i and p_j). A *Message-Loss* event removes a message from $SendBuffer_i(l)$ and discards it. Either of a *Message-Transfer* event or a *Message-Loss* event for message M will occur eventually after *Message-Send* event for message M. This takes indefinite amount of time. This fact captures the asynchronous nature of the communications considered.

This dissertation considers fail-stop link failures that occur before execution of a distributed algorithm begins. If a *Message-Transfer* event occurs for a link, then there will be no *Message-Loss* events for the link during the execution of an algorithm. Also, if a *Message-Loss* event occurs for a link, then there will be no *Message-Transfer* events for the link. This captures the nature of fail-stop link failures.

An execution of a communication statement "receive a message M over link l" by processor p_i results in a communication event *Message-Receive* that removes one

message in $ReceiveBuffer_i(l)$ if there is one available. If there is more than one message, the messages are removed in FIFO order. If there is no message, the execution has no effect. Contents of message M are available in local memory after the message is removed from the buffer.

A link l is said to be *faulty* if there is one or more *Message-Loss* events for the link l during an execution of an algorithm.

Chapter 3

Literature Survey

The problem of election has been studied extensively on many different topologies with various settings of parameters such as synchrony [16] and the availability of a sense of global direction [28, 39]. Some important results that are related to this dissertation are summarized in the following sections.

3.1 Reliable Networks

This section presents some results for the election problem in various topologies without any failures.

3.1.1 Ring Networks

A ring of processors is said to be *bidirectional* if all links in the ring are bidirectional. A ring of processors is said to be *unidirectional* if all links in the ring are unidirectional and a message sent by a processor can be delivered to its originator only by passing through all other processors in the ring.

The election problem for rings of n processors has received considerable attention since the first algorithm by LeLann for unidirectional rings [27]. The problem has been studied for bidirectional as well as unidirectional rings.

The first lower bound of $\frac{1}{4}n \log n + O(n)$ on worst-case message complexity was established by Burns [7] for bidirectional asynchronous rings when the size of the ring is not known to processors. Pachl et al. [33] showed average-case and worst-case lower bounds of $nH_n \approx .693n \log n + O(n)$[1] for asynchronous unidirectional rings when the size of the ring is not known to processors. For comparison-based algorithms, i.e., algorithms are restricted to using only comparisons between identifiers of processors, Frederickson and Lynch [16] proved a lower bound of $\frac{1}{2}n \log n + O(n)$ on the worst-case message complexity for synchronous bidirectional rings. This lower bound also applies to the asynchronous systems. Pachl et al. [33] showed an average-case lower bound of $\frac{1}{8}n \log n + O(n)$ for bidirectional rings. This was later improved by Bodlaender [5] to $\frac{1}{2}nH_n \approx .346n \log n + O(n)$.

An algorithm for the unidirectional case that asymptotically meets the worst case lower bound was first obtained by Peterson [36] with $1.440n \log n + O(n)$. It is later improved by Dolev et al. to $1.356n \log n + O(n)$ [12]. The average-case upper bound of $0.693n \log n + O(n)$ for the unidirectional case was achieved by Chang and Roberts [8]. However, its worst-case message complexity is $O(n^2)$.

Tables 1 and 2 present some significant results for both unidirectional and bidirectional cases.

3.1.2 Complete Networks and Bounded Degree Networks

Korach et al. [25] obtained an $\Omega(n \log n)$ lower bound on worst-case message complexity for the election problem on a complete network of processors.

Afek and Gafni [3] and Peterson [35] presented algorithm for the election problem

[1] H_n is the n^{th} Harmonic number.

Upper Bounds

	Average	Worst
LeLann (1977)	$O(n^2)$	$O(n^2)$
Chang & Roberts (1979)	$nH_n(\approx .693n \log n + O(n))$	$O(n^2)$
Peterson (1982)	$.943n \log n + O(n)^{\dagger}$	$1.440n \log n + O(n)$
Dolev et al. (1982)	$.967n \log n + O(n)^{\dagger}$	$1.356n \log n + O(n)$

Lower Bounds

	Average	Worst
Burns (1980)		$\frac{1}{4}n \log n + O(n)^{\ddagger}$
Pachl et al. (1984)	$nH_n(\approx .693n \log n + O(n))$	

†Empirical results by Everhardt (1984).

‡Also holds for bidirectional rings.

Table 1: Previous Work for Unidirectional Rings

Upper Bound

	Average	Worst
Hirschberg & Sinclair (1980)		$8n \log n + O(n)$
Santoro et al. (1982)		$1.89n \log n + O(n)$
van Leeuween & Tan (1985)		$1.440n \log n + O(n)$
Lavault (1989)	$\frac{1}{2}\sqrt{2}nH_n(\approx .490n \log n + O(n))$	$\frac{1}{4}n^2$

Lower Bound

	Average	Worst
Burns (1980)		$\frac{1}{4}n \log n + O(n)$
Pachl et al. (1982)	$\frac{1}{8} \log n + O(n)$	
Bodlaender (1988)	$\frac{1}{2}nH_n(\approx .347n \log n + O(n))$	

Table 2: Previous Work for Bidirectional Rings

on synchronous and asynchronous complete networks that require $O(n \log n)$ messages in the worst case. Loui et al. [28] showed that $O(n)$ messages suffice for election on asynchronous complete networks if a global sense of direction is available. (A complete network has a global sense of direction if the links of every processors are labeled as follows: A directed Hamiltonian cycle H is fixed and each link of every processor u is labeled according to the distance in H from u to processor adjacent via the link.)

For asynchronous *square meshes* of n processors (a square of n processors, with $\sqrt{n}$ processors on each side, where each column and each row form a ring), Peterson [35] showed that election is possible with $O(n)$ messages.

3.1.3 Arbitrary Networks

For an arbitrary connected asynchronous network with n processors and e communication links, it has been shown that $O(n \log n + e)$ messages are sufficient to elect a leader [17]. It has also been shown that any algorithm that solves the election problem for asynchronous networks whose topologies are not known to processors must use each communication link at least once [17, 25]. This lower bound holds even if synchrony is assumed [38].

3.2 Unreliable Networks

The impossibility result of Fisher et al. [14] implies that, if a processor may fail by stopping during an execution of an algorithm, then no election algorithm exists for asynchronous networks even if all links are reliable. On the other hand, the algorithms of Pease et al. [34], Dolev et at. [10], Dolev and Strong [11], and Coan [9] can be

modified to obtain election algorithms for synchronous complete networks with any type of processor failures.

The following sections summarize some results for networks with link failures.

3.2.1 Ring Networks with Link Failures

Goldreich and Shrira [19, 21] studied the election problem in asynchronous rings with one undetectable fail-stop link failure. (More than one such failure will disconnect the network.) For the case in which the size n of the ring is known to all processors, they presented an algorithm with worst-case message complexity of $\Theta(n \log n)$. For the case in which the size of the ring is not known to processors, they obtained an algorithm of worst-case message complexity of $\Theta(n^2)$ with the additional assumption that each processor knows the identifiers of the two processors adjacent to it.

3.2.2 Complete Networks with Link Failures

Abu-Amara [1] considered asynchronous complete networks with t undetectable fail-stop link failures and obtained an algorithm with worst-case message complexity $O(nt + n \log n)$. Masuzawa et al. [29] studied asynchronous complete networks with t fail-stop link failures with the assumption of a global sense of direction. They presented an algorithm whose worst-case message complexity is $\Theta(nt + t \log t)$, provided that $t < n - 1$.

Chapter 4

Average-Case Behavior of Election Algorithms on Rings

4.1 Introduction

As shown in the Chapter 3, there has been much research on the election problem for rings of processors. For unidirectional asynchronous rings, asymptotically optimal average-case message complexity algorithm and asymptotically worst-case message complexity algorithms have been presented [8, 12, 36].

The worst-case message complexity of Chang and Roberts's algorithm is $O(n^2)$ [8] but it is optimal for average-case message complexity for unidirectional asynchronous rings. The algorithm by Lavault [26], whose average-case message complexity is asymptotically optimal for bidirectional rings, also has worst-case message complexity $O(n^2)$ [6].

Average-case behaviors of asymptotically optimal worst-case algorithms were studied by Everhardt [13] with an empirical method. (The average-case message complexity was obtained by applying least-square method on the average number of messages for ring sizes ranging 5 to 200. The average number of messages for a size of ring is obtained by averaging the number of message over different assignment of identifiers to processors in the ring.) Everhardt's empirical results gave the average-case message

complexities of algorithms by Dolev et al. [12] and Peterson [36] as $.967n \log n + O(n)$ and $.943n \log n + O(n)$, respectively.

As observed above, known algorithms with "good" average-case message complexity (those of Chang and Roberts and of Lavault) behave poorly in the worst case. Also, the algorithms with the best known worst-case message complexity behave poorly in the average case. The availability of algorithms that have good average-case as well as worst-case behavior has significant meaning because of their practical importance. Furthermore, the existence of such algorithms is interesting because it is commonly believed that algorithms with better worst-case message complexity perform less well in the average case [30].

This chapter presents an algorithm for unidirectional rings and reports on sequential simulations that were used to analyze the algorithm's average-case behavior with statistical methods. A mathematical analysis of its average-case complexity would involve complicated techniques from the theory of combinatorial enumeration; however a statistical analysis suggests that the algorithm behaves nearly optimally in the average case. Also, it is shown by mathematical analysis that worst-case message complexity of the algorithm is approximately $1.440n \log n + O(n)$.

This chapter considers the election problem on asynchronous unidirectional rings of processors. A processor receives messages from one link and sends messages on the other link. A message sent by a processor can return to its sender after passing through all other processors in the ring. The size of the ring is not known to any processor, but the topology of the network is known to every processor. An algorithm is assumed to start up spontaneously. This is reasonable for ring networks, because the first message sent by initiator(s) of an algorithm can serve as a "wakeup" message

without increasing the message complexity.

The next section isolates the technique by which optimal average-case message complexity is achieved in the algorithm by Chang and Roberts. An improved algorithm is developed by applying similar techniques in Section 4.3. Section 4.4 analyzes by statistical methods the average case behavior of several algorithms, including the proposed algorithm.

4.2 Previous Algorithms and the Saving Technique

Electing a leader includes reducing the size of the number of candidates processors down to one and detecting the termination of the algorithm [4].

Termination detection for rings of processors is simpler than other networks. An elected processor sends a special declaration message that carries its identifier to one of its adjacent processor and terminates its execution of the algorithm. Upon receiving the message, a processor relays the message to another adjacent processor and terminates its execution of the algorithm. As long as the ring is connected, all processors in the ring eventually receive the special message and execution of the algorithm terminates.

In some algorithms for unidirectional rings, reducing the number of candidates processors is done as follows. Initially, all candidates processors are in *active* state and may later become *passive*; only one processor remains active through the algorithm. A processor maintains a temporary identifier (*tid*) that is initially its own. An active processor compares its *tid* with its adjacent processor's *tid* (called *nid*) and

determines whether to remain active and which *tid* to use according to some subset of the following rules:

D An active processor remains active if *tid* is less than *nid* and sets *tid* to *nid*.

A An active processors remains active if *tid* is greater than *nid* and keeps same *tid*.

G An active processor remains active if *tid* is greater than *nid* and sets *tid* to *nid*.

L An active processor remains active if *tid* is less than *nid* and keeps same *tid*.

The first and the second rules are called "descending (D)" and "ascending (A)" rules, respectively. Note that if all processors observe the first (or second) rule there are some consecutive processors in a ring whose identifiers form a descending (or ascending) sequence, the maximum *tid* in the sequence is compared to all other *tid*'s in the sequence and the processor with the maximum *tid* remains active, respectively. The third and the fourth rules are called "greater than (G)" and "less than (L)" rules, respectively. Several algorithms could be designed using one or two of the above rules. Peterson's algorithm [36] uses A and D rules.

Algorithm D (Figure 1) for unidirectional rings is designed using the "descending" rule. Initially, all processors in the ring are *active*. The number of active processors is reduced in the following way. Every processor maintains a local variable *tid* that is initially its own *id*. Only active processors initiate messages containing their *tid*'s and those are forwarded to the next active processor by passive processors. Upon receiving a message, an active processor compares its *tid* with delivered *id* (stored in a variable *nid* of the receiver). It becomes passive if *nid* is smaller than *tid*; otherwise it remains active, and sets its *tid* to *nid* ("descending" rule). In other words, the *tid*

Algorithm D

```
tid ← id;
state ← active;
send⟨tid⟩;
while (true) do
    receive⟨nid⟩;
    if (nid = id) then
        "Declare elected"
    else if (the received message is the declaration message) then
        "Set leader's identifier, and forward the identifier, and exit"
    else
        case state of
        active:
            if (nid > tid) then
                tid ← nid;
                send⟨tid⟩;
            else
                state ← passive;
        passive:
            send⟨nid⟩;
```

Figure 1: Algorithm D

of an active processor is forwarded to the next active processor and is then compared to that processor's *tid*.

The termination of the algorithm is detected by checking if the message received is the one sent by itself or a declaration message. Note that only active processors do the other comparisons; passive processors only relay messages. Also, note that any set of adjacent processors that form a descending chain of *id*'s have same *tid* at the end of execution of the Algorithm D.

The u^{th} *phase* of an active processor begins after it receives its u^{th} message. The u^{th} phase of a passive processor begins immediately before it receives its $(u+1)^{st}$ message. Note that a message that is delivered to an active processor in its phase u is originally sent by another active processor that enters its u^{th} phase by sending the message. This is clear since passive processors do not initiate messages.

The following shows it is shown that the *tid* of a passive processor is less than that of the next active processor to its right. Let $p_1, \ldots, p_i, \cdots, p_n$ be processors that forms a ring of size n. Let $tid_u(p_i)$ be the *tid* of an active p_i in its phase u and let $tid(p_i)$ be the $tid_u(p_i)$ if the phase p is the last phase in which p_i was in active. Consider a segment $p_i, \cdots, p_k, \cdots, p_j$ of a ring during an execution of algorithm D, where processors p_i and p_j are active in phase u, while all other processors in the segment are passive. Then, $tid(p_k) \leq tid_u(p_j)$ for $i < k \leq j$. This is obvious when $u = 1$. Assume this is true for the phase $u-1$. Let $p_{k_1}, \ldots, p_{k_m}$ ($i \leq k_h \leq j$ for $1 \leq h \leq m$) be the processors that become passive in the phase $u-1$. Then, $tid(p_{k_1}) < tid(p_{k_2}) < \cdots < tid(p_{k_m}) = tid_u(p_j)$, since $p_{k_1}, \cdots, p_{k_m}$ become passive and p_j is active in phase $u-1$. Therefore, the claim is true for the phase u. Also, it is true for later phases since passive processors never changes their *tid*.

With this observation, algorithm D can be modified to eliminate some messages by selectively forwarding messages at passive processors. A passive processor relays only messages with *nid* that is greater than its *tid* instead of always relaying incoming messages. This technique is called the "saving technique".

Since the initial value of *tid* of a processor is its *id* and *tid* is not changed if $nid < tid$, all messages are forwarded up to the processor whose *id* is greater than that of the original sender. Algorithm D with the saving technique is exactly Chang and Roberts's algorithm, which is optimal in average-case message complexity. Figure 2 shows Chang and Roberts's algorithm.

Chang and Roberts's algorithm (algorithm D with the saving technique) has optimal average-case message complexity. The following shows that the saving technique does not increase the worst-case message complexity.

Consider executions of algorithms D and Chang and Roberts's algorithm on same ring. If a processor is active and sends a message in phase p during an execution of Chang & Roberts's algorithm, then the processor is active in phase u during an execution of algorithm D. Also, a message sent by an active processor in phase u in Chang and Roberts's algorithm travels at most as far as the message sent by the same processor in the same phase of algorithm D. Thus, the saving technique does not increase the worst-case message complexity of the original algorithm.

4.3 The New Algorithm

As shown in the previous section, the saving technique is useful in achieving good average-case complexity while it does not increase the worst-case message complexity.

Algorithm Chang & Roberts

```
tid ← id;
state ← ACTIVE;
send⟨tid⟩;
while (true) do
    receive⟨nid⟩;
    if (nid = id) then
        "declare elected"
    else if (the received message is the declaration message) then
        "Set leader's identifier, and forward the identifier, and exit"
    else
        case state of
        active:
            if (nid > tid) then
                tid ← nid;
                send⟨tid⟩;
            else
                state ← PASSIVE;
        passive:
            if (nid > tid) then
                send⟨nid⟩;
```

Figure 2: Chang & Roberts's Algorithm

This section first presents a simple algorithm (called DG; see Figure 3) that is similar to algorithm D, but its worst-case message complexity is $O(n \log n)$ instead of $O(n^2)$. The algorithm is then improved by applying the saving technique.

Algorithm DG

```
tid ← id;
state ← active;
parity ← true;
send⟨tid⟩;
while (true) do
    receive⟨nid⟩;
    if (nid = id) then
        "declare elected";
    else if (the received message is the declaration message) then
        "Set leader's identifier, and forward the identifier, and exit"
    else
    case state of
        active:
            if ((nid < tid) ⊕ parity)† then
                tid ← nid;
                send⟨tid⟩;
            else
                state ← passive;
            parity ← ¬parity;
        passive:
            send⟨nid⟩;
```

---

† $\oplus$ denotes *exclusive or*.

Figure 3: Algorithm DG

4.3.1 Algorithm DG

In algorithm D, the rule for a processor to remain active is that the received *tid* is should greater than its own *tid* (D rule). Let $p_1, \cdots p_n$ be processors such that

processors p_i and p_i are adjacent to each other if $j = (i+1) \bmod n$ in a ring of n processors. If $id(p_1) < id(p_2) < \cdots < id(p_n)$, an execution of algorithm D on the ring uses $O(n^2)$ messages. To avoid this, algorithm DG adopts another rule by which a processor may remain active even if the received *id* is less than its own *tid* (G rule). Algorithm DG applies these two rules in alternate phases. Since both rules require to set *tid* to *nid*, both rules can be adopted in algorithm DG using a variable *parity*. (See Figure 3.)

This reduces the worst-case message complexity to $O(n \log n)$. Note that active processors set their *tid*'s to a received *id* using both rules.

4.3.2 Algorithm DGS

To apply the saving technique, the following uses an observation that is similar to that for algorithm D. Let $p_i, \cdots, p_{k_1}, \cdots, p_{k_m}, \cdots, p_j$ be a segment of a ring, where p_i and p_j are active at the end of phase $u-1$, the processor p_{k_l} $(1 \leq l \leq m)$ are processors that became passive in phase $u-1$, and all other processors in the segment are passive from the beginning of that phase. Assume that *parity* is false ("greater than" rule applied) in phase $u-1$. Since $p_{k_1}, \cdots, p_{k_m}$ became passive and p_j is active in phase $u-1$, $tid(p_{k_1}) \geq \cdots \geq tid(p_{k_m}) = tid_{u-1}(p_j)$. Then $tid(p_{k_l}) \geq tid_u(p_j)$ holds for all $1 \leq l \leq m$ at the beginning of phase u. If $tid_{u-1}(p_i) > tid(p_{k_l})$ for some $1 \leq l \leq m$ at the beginning of the phase u, then $tid_{u-1}(p_i) > tid_u(p_j)$ and p_j remains active in the phase u. Thus, the saving technique can be applied. The message sent by p_i can be stopped at p_{k_1} in phase u. Note that, if the message sent by p_i stops earlier in phase u, then processor p_j remains active in phase u since it does not receive a message in phase u.

With this observation, algorithm DG can be improved as follows. Since messages are stopped early only at processors that became passive in the last phase, another state *recent* is introduced to distinguish such processors from other passive processors. (Processors p_{k_l} $(1 \leq l \leq m)$ become recent in the above example.) Recent processors become active if they receive an *id* greater than *tid* when the value of *parity* is true. Passive processors relay messages as always.

A message that stops at active processor p_i in algorithm DG stops at a recent processor p_{k_1} in the modified algorithm. Thus, the recent processor p_{k_1} needs to act as if it were the active processor p_j. The active processor p_j needs only to relay messages in the modified algorithm. Since p_j does not receive messages in phase u, p_j remains active and the parity of p_j does not change in phase u. Thus, the algorithm is modified so that every message contains its sender's parity as part of a message. An active processor compares its parity with that contained in the message received. If the two parities are different, the processor becomes passive. It is possible that some recent processors do not receive a message in a phase. These processors become passive in the following phase. Thus, every recent processor compares its parity with one contained in the message received (if it receives one) and become passive if the values are different.

A similar modification is also possible for phases in which the value *parity* is false. For phases with false parity, recent processors become active if *nid* is less than *tid*; otherwise they become passive. Algorithm DG with the saving technique is the algorithm DGS (Figure 4).

Note that some messages could be saved in every phase (this contrasts with algorithm DG). A similar technique is used in the algorithm of Dolev et al. [12], but

Algorithm DGS

```
state ← ACTIVE;
tid ← id;
parity ← true;
send⟨tid, parity⟩;
while (true) do
    receive⟨nid, nparity⟩;
    if (nid = id) then
        "declare elected";
    else if (the received message is the declaration message) then
        "Set leader's identifier, and forward the identifier, and exit"
    else
        case state of
        active:
            if (nparity ≠ parity) then
                state ← passive;
                send⟨nid, nparity⟩;
            else if ((nid < tid) ⊕ parity) then
                state ← recent;
            else
                parity ← ¬parity;
                tid ← nid;
        recent:
            if ((nparity ≠ parity) ∧ ((nid > tid) ⊕ nparity)) then
                tid ← nid;
                state ← active;
                send⟨tid, nparity⟩;
            else
                send⟨nid, nparity⟩;
                state ← passive;
            endif;
        passive:
            send⟨nid, nparity⟩;
```

Figure 4: Algorithm DGS

saving is possible only in every other phase in that algorithm. Recently, Higham presented an algorithm where messages are stopped earlier in every phase [22]. This algorithm is similar to algorithm DGS. It was claimed that the worst-case message complexity is $1.272n \log n + O(n)$. Unfortunately, this algorithm contains a non-trivial error. has been

Figure 5 shows executions of algorithms DG and DGS on a ring of 13 processors. In both tables, the first lines shows the *id*'s of processors in the ring. Each of following

	9	1	8	11	5	7	13	4	10	3	6	12	2
T	-	9	-	-	11	-	-	13	-	10	-	-	12
F		-			9			11		-			10
T					10			-					11
F					-								10
T									10				

An Execution of Algorithm DG

	9	1	8	11	5	7	13	4	10	3	6	12	2
T	R	9	R	R	11	R	R	13	R	10	R	R	12
F	-	R	-	9		-	11		-	R	-	10	
T		10			-		R	-		11			-
F		R		-			10					-	
T		10											

An Execution of Algorithm DGS

Figure 5: Sample Executions of Algorithms DG and DGS

lines show *tid*'s of each processor at the end of successive phases. "T" and "F" in the first column of each line represent the values (true and false, respectively) of *parity* used in that phase. If a processor is active at end of a phase, a number (indicating

the processors' *tid*) is shown. Processors in recent state (for algorithm DGS only) are denoted "R" and passive processors are denoted "-". In the execution of algorithm DGS, a blank means the corresponding processor did not receive a message in that phase. (Every processor receive a message in any phase of an execution of algorihtm DG.)

4.3.3 Worst-Case Message Complexity of Algorithm DGS

The worst-case message complexity of algorithm DG will be shown to be $O(n \log n)$, and so is that of the algorithm DGS, since the new saving technique does not increase the worst-case message complexity. (The proof of this is similar to that for Chang and Roberts's algorithm).

An analysis of worst-case message complexity of algorithm DG follows. Let u be the maximum number of phases for an execution of algorithm DG on a given ring. Number the phases in reverse order so that u is the first phase and 1 is the last phase (phase $u+1$ denotes before the start of the algorithm). Let m_k be the number of active processors at the end of the phase k. Then, $m_1 = 1$ and $m_{u+1} = n$. Let p_i and p_j be two active processors at the beginning of the phase k such that p_j receives a message from p_i in the phase. Assume that p_j is active at the end of the phase k. Then there is at least one active processor between p_i and p_j in phase $k+1$. Otherwise, p_j received a message from p_i in the phase $k+1$ and $tid_k(p_j) = tid_{k-1}(p_i)$. Then p_i cannot remain active since the parity of phase k is different from that of phase $k+1$. Thus, a processor may remain active only if there there is at least one processor that became passive in the previous phase. This means that the number of processors remaining active at the end of phase k is at most the number of processors

that became passive during the phase $k+1$. In other words, $m_k \leq m_{k+2} - m_{k+1}$. Or, $m_{k+2} \geq m_{k+1} + m_k$. This gives a Fibonacci progression, so that $m_k \geq F_{k+1}$ where F_k is the k^{th} Fibonacci number. $F_{u+1} = \frac{1}{\sqrt{5}}(\phi^{u+1} - \hat{\phi}^{u+1})$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$. Since $|\hat{\phi}^{n+1}| < 1$ for $n \geq 0$, $u \leq 1.440 \log n + O(1)$ is obtained by taking logarithms. Since every phase requires n messages, the total number of messages is $1.440n \log n + O(n)$, where $O(n)$ messages also includes messages needed to broadcast the *id* of leader to all other processors.

4.3.4 Correctness of Algorithm DGS

Algorithm DGS is correct if algorithm DG is correct. This follows from the fact that the number of active processors which are active in phase i of an execution of algorithm DGS is same as that of algorithm DG, since the there is only one active processors at the end of an execution of algorithm DG.

The correctness of algorithm DG follows from the fact that the number of active processors in each phase decreases as an execution proceeds and that only one processor receives a message that carries its own *id*. The first fact comes by noting that there always is a processor with maximum (or minimum) *tid* among all active processors in any phase. The processor with maximum (or minimum) *tid* in a phase causes at least one processor to become passive in the phase. The processor with maximum (or minimum) *tid* remains active in the next phase.

The second fact can be shown as follows. Assume that there were two or more processors that received messages that carry their own *id*. Let p_i and p_j be such processors and let $id(p_i)$ and $id(p_j)$ be their *id*'s, respectively. Then, p_i and p_j are both active when they receive messages carrying their own *tid*'s in any phase of an

execution of the algorithm. Without loss of generality, it can be assumed that p_j is to the right of p_i and a processor receives a message from its left link. Since messages are delivered in FIFO order on a link, the message carrying $id(p_j)$ will be delivered to p_i before the message carrying $id(p_i)$. Therefore, p_i cannot be active when the message carrying its own *id* is delivered. Thus, only one processor sees the message carrying its own *id*. If there is only one active processor in any phase, that processor will declare itself elected in the next phase.

Theorem 4.3.1 *Algorithm DGS solves the election problem correctly with worst-case message complexity* $1.440n \log n + O(n)$.

4.4 Analysis of Average-Case Message Complexity

As shown in Chapter 2, the average-message complexity $\overline{\mu_n}[\mathcal{A}]$ of an election algorithm $\mathcal{A}$ for unidirectional rings is defined as follows by assuming that all ring permutations are equally likely:

$$\overline{\mu_n}[\mathcal{A}] = \frac{1}{|\boldsymbol{I}_n|} \sum_k k J_{nk} = \frac{1}{(n-1)!} \sum_k k J_{nk},$$

where J_{nk} is the number of ring permutations of size n in which k messages are exchanged by algorithm $\mathcal{A}$.

The function $f(n) = \overline{\mu_n}[\mathcal{A}]$ is of interest. A regression analysis was performed to analyze the average-case behaviors of three algorithms: Chang and Roberts's algorithm, Peterson's algorithm, and algorithm DGS. The regression analysis of Chang

and Roberts's algorithm was performed as an indicator of reliability of the analysis. The function $f(n)$ is modeled with the regression equation $\beta_0 + \beta_1 n + \beta_2 n \log n$. This regression equation is used since the worst-case message complexities of the analyzed algorithms and the average-case message complexity of Chang and Roberts's algorithm are expressed with the function.

As shown in Figures 6 and 7, the standard deviation of the number of messages is not constant with ring size. Thus, the weighted least square method is used for the regression. To increase the reliability of analysis, large sample sizes are chosen so that the result of regression will be very close to the theoretical analysis for Chang and Roberts's algorithm.

The average number of messages for each algorithm is calculated by sequential simulation of each algorithm. Ring size n is sampled from 21 to 2400 (including multiples of 50, powers of 2, and Fibonacci numbers). The average number of messages for each n is the average of $100n$ random permutations. Permutations of identifiers are obtained by an algorithm that generates random cyclic permutations. All simulations were performed on a Sequent Symmetry with 10 processors. The results of the regression analysis are shown in Figures 8, 9 and 10.

The result for Chang and Roberts's algorithm is almost same as that of the theoretical analysis. The simulation gives $.693n \log n + .572n + .852$, while theoretical analysis gives $.693n \log n + .577n + .5$. This suggests that the regression analysis is reliable, since the same number of samplings is done for all algorithms. (Reliability of the regression analysis is heavily dependent on the sample size.)

For Peterson's algorithm, Everhardt [13] obtained $\beta_2 = .943$ by statistical analysis. (This result is obtained by 10 simulations for $n < 20$ and $n/2$ simulations for

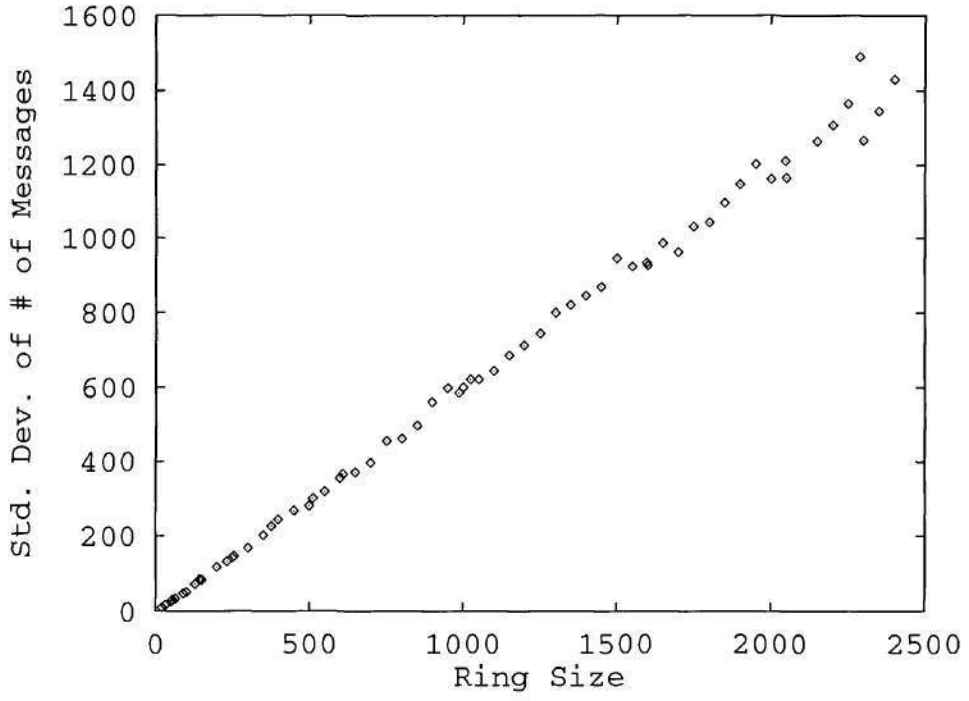

Figure 6: Standard Deviation of Chang and Roberts' Algorithm

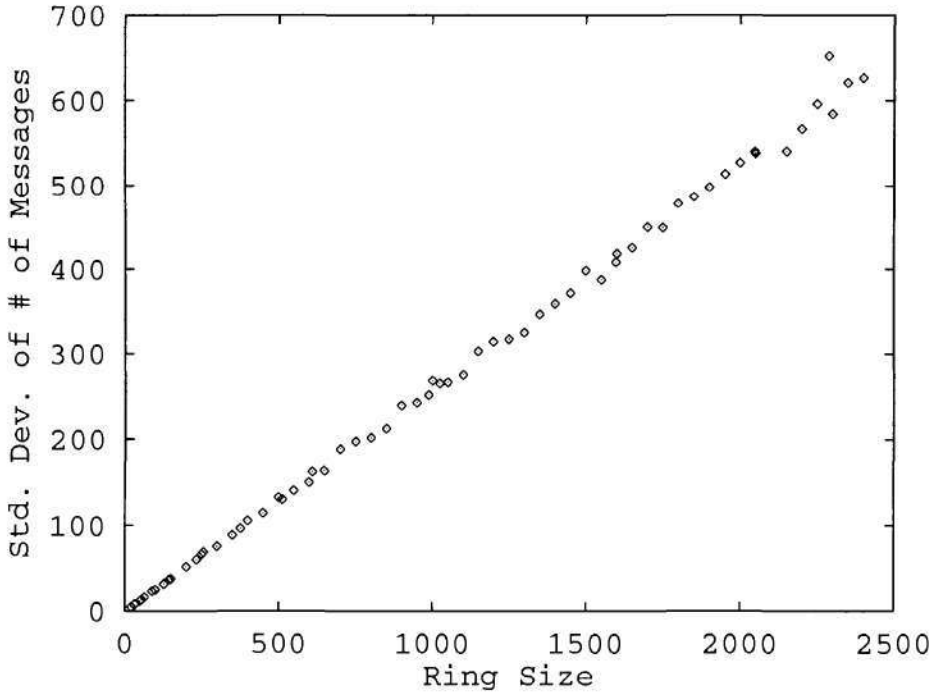

Figure 7: Standard Deviation of Algorithm DGS

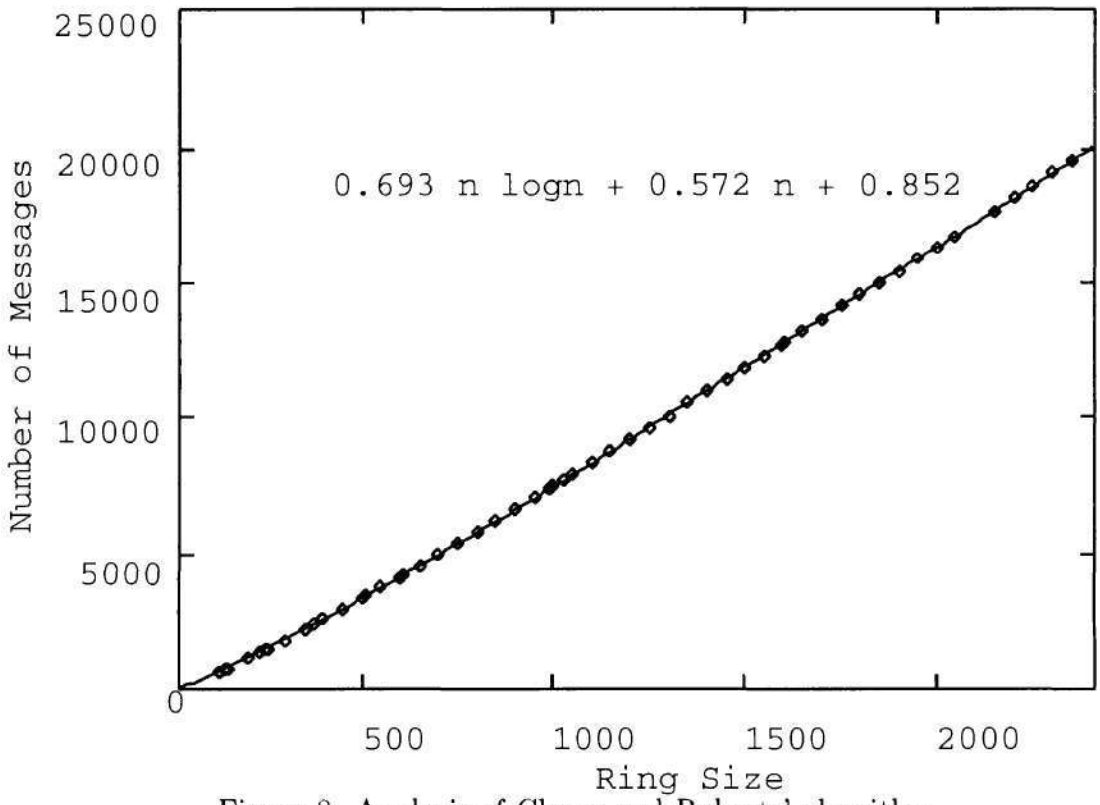

Figure 8: Analysis of Chang and Roberts' algorithm

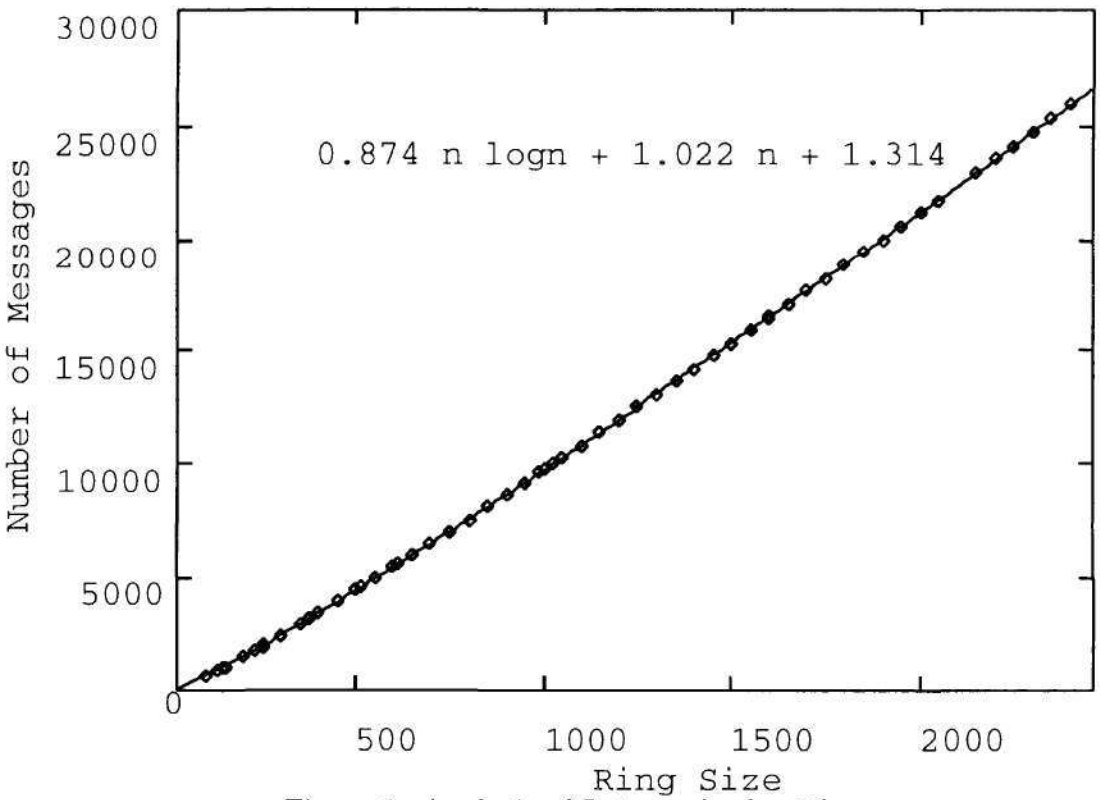

Figure 9: Analysis of Peterson's algorithm

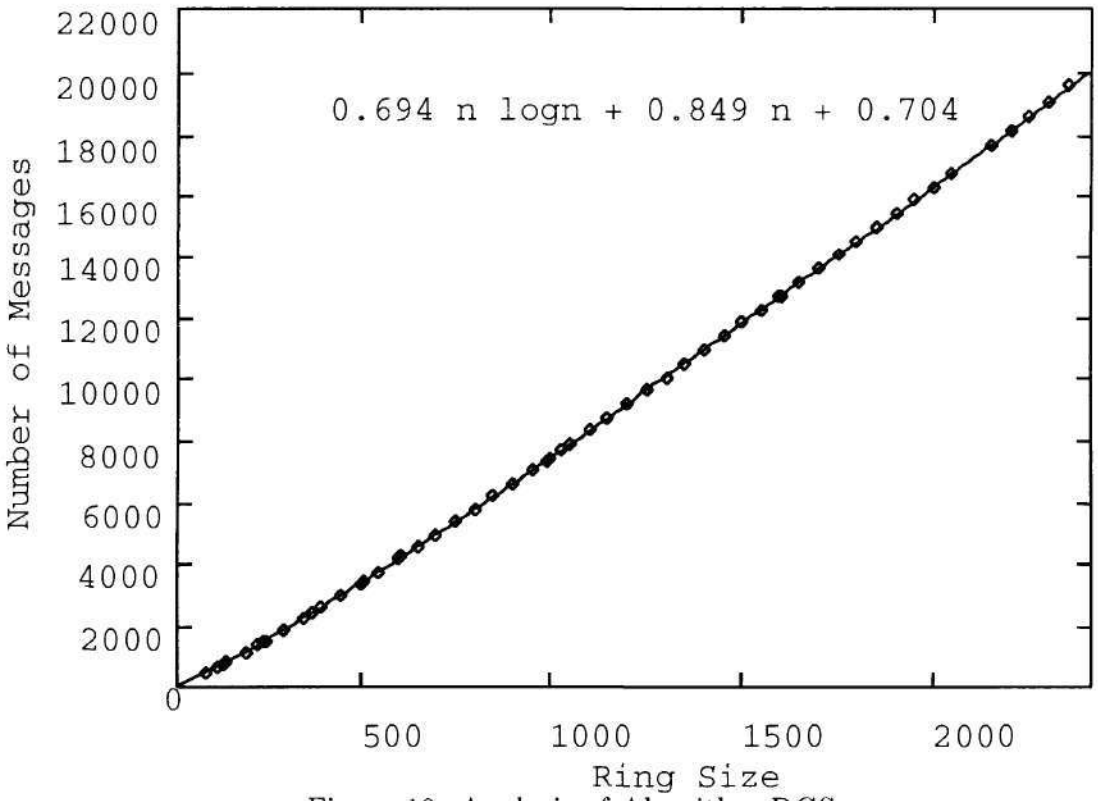

Figure 10: Analysis of Algorithm DGS

Upper Bounds

	Average	Worst
LeLann (1977)	$O(n^2)$	$O(n^2)$
Chang & Roberts (1979)	$.693n \log n + O(n)$	$O(n^2)$
Peterson (1982)	$\boxed{.873n \log n + O(n)}$ †	$1.440n \log n + O(n)$
Dolev et al. (1982)	$.967n \log n + O(n)$‡	$1.356n \log n + O(n)$
Algorithm DGS	$\boxed{.694n \log n + O(n)}$ †	$1.440n \log n + O(n)$

Lower Bounds

	Average	Worst
Burns (1980)		$\frac{1}{4}n \log n + O(n)$§
Pachl et al. (1984)	$.693n \log n + O(n)$	

†Empirical results of this chapter.

‡Empirical result by Everhardt (1984).

§For bidirectional rings.

Table 3: Upper and Lower Bounds for Unidirectional Rings

$n > 20$ for n ranging from 5 to 200.) The result obtained here is $\beta_2 = .873$, which should be more accurate since a larger range of ring size were used and more simulations were performed for each ring size. The results from the regression analysis for algorithm DGS strongly suggest that the algorithm is very close to optimal within lower order terms in the average case complexity. (The simulation of algorithm DGS gives $.694n \log n + .849n + .704$.) The results are summarized in Table 3 with related previous results. (The contributions of this chapter are boxed.)

Every message in algorithm DGS contains the *tid* of its sender and the value of *parity* of the phase. Therefore, the size of each message is $b + 1$ bits where b is the length of longest identifier. Note that any comparison algorithm needs b bits for every message, since identifiers of processors should be exchanged for comparisons.

4.5 Concluding Remarks

This chapter presented an election algorithm on unidirectional rings of processors. While mathematical analysis of the average-case message complexity is an open problem, statistical analysis suggests that the algorithm has essentially the same average-case message complexity as Chang and Roberts's algorithm. Also, the algorithm has $O(n \log n)$ worst-case message complexity while the Chang and Roberts's algorithm has $O(n^2)$. This algorithm is important since it has good average-case message complexity as well as good worst-case complexity. This is done at the cost of one more bit for every message. This result is interesting because it is contrary to the common belief that algorithms with good worst-case complexity perform worse in the average case.

The simulation result for Peterson's election algorithm should be more accurate than the previous simulation result [13], since a larger sample size and a more sophisticated analysis were used.

Chapter 5

Election on Faulty Rings with Incomplete Size Information

5.1 Introduction

In many previous studies of the election problem in the ring network, it has been assumed that every message sent over a link is eventually delivered. This chapter considers rings in which this assumption need not hold. It is assumed that a link may be faulty and messages sent over the link might not be delivered. (If there are two more more faulty links, there are disconnected processors.) This situation is especially interesting in the case of asynchronous rings, since failed links cannot be detected in these networks [20, 23, 41].

Election involves two main tasks: resolving the competition between candidates for a leader (usually all processors participating the election are candidates at the beginning of an algorithm) and detecting termination of the algorithm [4]. For rings of processors without failures, termination can be detected when a message returns to its sender after passing though all other processors. This may not be possible if one or more links may fail.

It has been shown that election is impossible in an asynchronous ring with one fail-stop link failure if the size of ring is not known to processors [20]. Thus, the

knowledge of the size of the ring is important if there are faulty links. This chapter considers cases where the size of the ring is known to processors in inexact form, i.e. the lower bound and/or the upper bound of the size are known to processors instead of the exact value of the size.

Even for the cases in which the size of the ring is not known to processors, the election problem may be solvable if some other information is available: for example, if each processor knows identifiers of two neighbors. Goldreich and Shrira [20] showed that there is an algorithm with worst-message complexity of $O(n^2)$ for this case.

This chapter considers the following cases on a lower bound ℓ and an upper bound u of the size n:

- every processor knows ℓ and u such that $\ell = u$
- every processor knows ℓ and u such that $\ell > \frac{u}{2}$
- every processor knows ℓ and u such that $\ell \leq \frac{u}{2}$
- every processor knows ℓ but does not know u.

(Note that a processor always knows that the lower bound is at least 1, since it knows that it is part of the ring.) As shown in Table 4, there are many cases depending on the relationship between ℓ and u and the availability of two neighbors' identifiers. (For all cases, it is assumed that every processor knows its own identifier.) This chapter examines all possible cases and reports upper bounds, matching lower bounds, and impossibility results.

Goldreich and Shrira [20, 21] considered some of these cases. They showed that worst-case message complexity is $\Theta(n \log n)$ when the exact size of the ring is known

	$u = \infty$	$\ell \leq \frac{u}{2}$	$u > \ell > \frac{u}{2}$	$\ell = u$
Knows Neighbors	$\Theta(n^2)^\dagger$	$\Theta(n \log n + (n - \ell)n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Does Not Know Neighbors	Impossible†	Impossible		

† For the case $\ell = 1$.

Table 4: Cases Considered in this Chapter

to all processors. (This also holds for the case in which the identifiers of its two neighbors are also known to each processor; election is possible without this additional knowledge. Furthermore, the identifiers of neighbors for every processor can be discovered with $O(n)$ messages.)

Goldreich and Shrira showed that worst-case message complexity is $\Theta(n^2)$ if every processor knows its own identifiers and identifiers of its two neighbors but the size of the ring is not known. They proved that it is impossible to elect a leader if the only input to every processor is its own identifier.

This chapter presents an algorithm with worst-case message complexity $O(n \log n)$ that solves election problem for all cases with $\Theta(n \log n)$ in the Table 4. Note that the algorithm with worst-case message complexity $O(n \log n)$ by Goldreich and Shrira does not work for all of those cases. An algorithm with worst-case message complexity $O(n \log n + (n - \ell)n)$ is also presented in this chapter. This algorithm solves election problem for the two cases; $u = \infty$ with the knowledge of two neighbors' identifiers and $\ell \leq \frac{u}{2}$ with the knowledge of two neighbors' identifiers. Note that the algorithm by Goldreich and Shrira with $O(n^2)$ worst-case message complexity does not work for all cases that the $O(n \log n + (n - \ell)n)$ algorithm covers. It is shown that election is impossible for two other cases.

5.2 Preliminaries

This section describes the assumptions made in this chapter. The definition of the election problem is given in Chapter 2.

It is assumed that rings are asynchronous and bidirectional. It is also assumed that a processors can distinguish its two links. Thus, a processor can relay a message by receiving a message from a link a send it over the other link. Also, a processor can return a message received over the link from which it is receive. All message over a link are subject to delivery in FIFO order.

The type of link failure considered in this chapter is fail-stop link failure. Since the communication is asynchronous, faulty links are not detectable [20]. It is assumed that there is at most one faulty link in a ring. Thus, all processors in a ring remain connected by non-faulty links.

5.3 Algorithms with Worst-Case Message Complexity $O(n \log n)$

This section presents algorithms that solves the election problem for the following four cases:

1. A lower bound ℓ and an upper bound u of the size of ring such that $\ell > \frac{u}{2}$ are known to all processors;

2. the exact size of ring is known to all processors;

3. same as the case 1 with additional knowledge of neighbors' identifiers;

4. same as the case 2 with additional knowledge of neighbors' identifiers.

An algorithm (called algorithm $R1$) for the first case is presented. It will be shown that the algorithm can be used for other cases.

5.3.1 Description of Algorithm $R1$

Algorithm $R1$ is shown in Figures 11 and 12.

In describing algorithm $R1$, the following conventions are used. Two links of a processor are referred with names *left* and *right*. The statement "send $\langle var_1, \cdots, var_k; link \rangle$" is to be interpreted as "send a message whose content is *var* in direction *link* (*link* is *left*, *right*, or *both*). The statement "receive $\langle var_1, \cdots, var_k; link \rangle$" is to be interpreted as "receive a message and store the contents of the message to variables $var_1, \cdots, var_k$, and store the link from which the message is received into *link*.

Before describing the algorithm, some concepts should be defined. Throughout an execution of algorithm $R1$, processors can be in an *active*, *passive*, or *elected* state. Initially, all processors are active. As the algorithm proceeds, active processors become passive. Eventually, one active processor remains active, and this processor becomes elected. During an execution of the algorithm, each processor is in some local phase. The value of the current phase is stored in a local variable *phase*.

The algorithm operates in three stages. In the first stage, the number of active processors is decreased to some constant by sending $O(n \log n)$ messages. In the second stage, the number of active processors is further decreased to some smaller constant by sending another $O(n \log n)$ messages. Finally, the number of active processors becomes one in the third stage with $O(n)$ messages. The only active processor

Algorithm $R1$

$state \leftarrow active$; $myId \leftarrow id$; $phase \leftarrow 0$; $stage3 \leftarrow false$;
$lSize \leftarrow 0$; $rSize \leftarrow 0$; $lLinkOk \leftarrow false$; $rLinkOk \leftarrow false$;
$nSent \leftarrow 0$; $received \leftarrow 0$;
NewPhase:
if ($stage3$) then
  $received \leftarrow received + 1$;
  if ($received = nSent$) then
    Declare "elected";
  else if (the received message is the declaration message) then
    "set leader's identifier and exit"
  else /* $received < nSent$ */
    goto Wait;
$phase \leftarrow phase + 1$;
if ($phase > 1$) then
  if ($receivedLink = left$) then
    $lSize \leftarrow |otherDist| + 1$;
  else /* $receivedLink = right$ */
    $rSize \leftarrow |otherDist| + 1$;
if ($phase < \lfloor \log \ell \rfloor$) then /* the 1^{st} stage */
  send $\langle 1, phase, myId, 2^{phase} - 1; both \rangle$;
else if ($lSize + rSize + 1 < \ell$) then /* the 2^{nd} stage */
  send $\langle 2, phase, myId, lSize + \lceil \frac{\ell - (lSize + rSize + 1)}{2} \rceil; left \rangle$;
  send $\langle 2, phase, myId, rSize + \lceil \frac{\ell - (lSize + rSize + 1)}{2} \rceil; right \rangle$;
else /* $lSize + rSize + 1 = \ell$ */ /* the 3^{rd} stage */
  $stage3 \leftarrow true$;
  if ($lLinkOk$) then
    send $\langle 3, phase, myId, \infty; left \rangle$;
    $nSent \leftarrow nSent + 1$;
  if ($rLinkOk$) then
    send $\langle 3, phase, myId, \infty; right \rangle$;
    $nSent \leftarrow nSent + 1$;

Figure 11: Algorithm $R1$

```
Wait:
receive ⟨otherRound, otherPhase, otherId, otherDist; receivedLink⟩;
if (otherPhase ≥ phase) then
    if (receivedLink = left) then
        lLinkOk ← true;
    else /* receivedLink = right */
        rLinkOk ← true;
if (((otherPhase, otherId) = (phase, myId)) ∧ (state = active)) then
    goto NewPhase;
else if ((otherPhase, otherId) > (phase, myId)) then
    state ← passive;
    /* forward the message */
    if (otherRound ≤ 2) then
        if (otherDist = 1) then
            if (receivedLink = left) then
                receivedLink ← right;
            else /* receivedLink = right */
                receivedLink ← left;
    else /* otherRound = 3 */
        if ((receivedLink = left) ∧ ¬lLinkOk) then
            receivedLink ← right;
        else if ((receivedLink = right) ∧ ¬lLinkOk)
            receivedLink ← left;
    if (receiveLink = left) then
        receivedLink ← right;
    else
        receivedLink ← left;
    send ⟨otherRound, otherPhase, otherId, otherDist − 1; receivedLink⟩;
goto Wait;
```

Figure 12: Algorithm $R1$ (continued)

at the end of the third stage declares itself elected by broadcasting its identifier to all processors in the ring.

The following mechanism for "forwarding" messages is used in first two stages. Let p_{i-1}, p_i, p_{i+1} be three consecutive processors in a ring; *forwarding a message* is defined as follows: Upon receiving a message M that contains distance d (that is a part of a message) from p_{i-1}, processor p_i sends M with new distance $d-1$ to p_{i+1} if $d \neq 1$, or sends M with new distance $d-1$ to p_{i-1} if $d = 1$. Note that when a message returns to the processor that originates the message, $d < 0$ and $|d| + 1$ is the number of different processors it passes through. (Similarly, upon receiving a message M that contains distance d from p_{i+1}, processor p_i sends M with new distance $d-1$ to p_{i-1} if $d \neq 1$, or sends M with new distance $d-1$ to p_{i+1} if $d = 1$.)

Throughout an execution of the algorithm, active processors become passive by the following rule (called the "killing rule"). During an execution of the algorithm, each active processor p_i replies to incoming messages. Let *otherPhase* and *otherId* be parts of an incoming message. Let *phase* and *myId* be the local phase and the identifier of processor P_i, respectively. If $(otherPhase, otherId)$ is greater than $(phase, myId)$ (in lexicographic order) then p_i becomes passive and the message is forwarded. If the pair $(otherPhase, otherId)$ is less than the pair $(phase, myId)$, the incoming message is not forwarded but is discarded. If an active processor receives a message carrying $myId$, it enters the next phase. Passive processors always forward a received message.

The first stage operates as follows. Upon entering phase v, an active processor p_i sends messages to both directions to distance $d = 2^v - 1$ and waits for the return of one of these messages. If one such message returns, p_i enters phase $v + 1$.

The concept of *segment* is used in describing the algorithm. Every active processor

has its own segment. At the beginning of the algorithm, the segment of a processor p_i is p_i itself. Assume that an active processor p_i receives a message m that was originated by itself in phase p, and enters the phase $p+1$. The segment of p_i at the end of phase p is defined as the union of the set of all processors that received the message m and the segment of p_i at the beginning of phase p.

Note that active segments are not processor disjoint. Let $p_i, p_{k_1}, \cdots, p_{k_m}, p_j$ be part of a ring, where p_i and p_j are active and all others are passive. Then, $p_i, p_{k_1}, \cdots, p_{k_m}$ and $p_{k_1}, \cdots, p_{k_m}, p_j$ can be two active segments with active processors p_i and p_j, respectively. The forwarding mechanism ensures that there is only one active processor in a segment.

The size of the segment is maintained with two variables *lSize* and *rSize* at every active processor. When a message returns back to its sender by the forwarding mechanism, a variable *lSize* (or *rSize*) at the active processor is updated to $|Dist|+1$ that is the number of processors on the right (or left, respectively) of the active processor in the segment.

At the end of the last phase (phase $\lfloor \log \ell \rfloor$) of the first stage, the size k of a segment (the number of processor in the segment) of an active processor is $\frac{1}{2}2^{\lfloor \log \ell \rfloor} \leq k \leq \frac{3}{4}2^{\lfloor \log \ell \rfloor}$. (The size of the segment of a processors is minimal if all messages return to the processor during the first stage in one direction. It is maximum if the messages return to the processor in the last two phases of the first stage in different directions.) Processors that reach phase $\lfloor \log \ell \rfloor$ start the second stage.

During the second stage, an active processor tries to increase the size of its segment to ℓ. Upon entering phase v in the second stage, an active processor p_i tries to extend the size of its segment by half of $d/2 = (\ell - k)/2$ (where k is the size of its segment)

by sending messages in both directions to distance $\lceil d/2 \rceil$ (starting from processors at the end of the segment) beyond both end processors of the segment. If one of these messages returns, p_i enters phase $v + 1$.

Since the segments of active processors are not processor disjoint, there can be more than one active segment of size ℓ with an active processor even if $\ell > u/2$. (This is discussed in detail in the proof of the correctness.) The active processor of a segment of size ℓ cannot declare itself as elected since there could be one more such processor, even though the number of segments of size ℓ is bounded by some constant. If the active processors with segments of size ℓ simply broadcast their id's, the processors that receive these id's cannot determine whether there is more than one such processor. Thus, it is necessary to further reduce the number of active processor to one. This the task of the third stage.

The third stage adopts a forwarding mechanism different from the one used in the first two stages. During the first stage, a message returns to its sender after traveling a specified distance. In the third stage, a message returns back to its sender if the message reaches a link that was not specially marked during the second stage. A link is said to be "proven non-faulty", if it has delivered at least one message that is sent by a processor in the phase greater than or equal to that in which the message was initiated.

Let p_{i-1}, p_i, p_{i+1} be three consecutive processors in a ring, then *forwarding a message* in the third stage is defined as follows: Upon receiving a message $\langle m \rangle$ from p_{i-1}, processor p_i sends $\langle m \rangle$ to p_{i+1} if its left (or right) link is proven non-faulty and the message is received from its right (or left, respectively). Otherwise, p_i sends $\langle m \rangle$ back to p_{i-1}. (Similarly for messages received from p_{i+1}.)

A variable *lLinkOk* (*rLinkOk*) at every processors is set to true if a processor receives a message that is sent by another processor in the phase greater than or equal to its own phase from its left (or right, respectively) link. These variables are used by the forwarding mechanism in obvious way.

The same killing rule as the one used in the first two stages is used for the third stage. Upon entering the third stage, an active processor sends messages over the links that are proven non-faulty. Since all active processors are connected by proven non-faulty links in the third stage (this is proved in the following section), the forwarding mechanism ensures that messages sent by an active processor are delivered to its nearest active processor. Also, the killing rule ensures that messages sent by the processor with the largest *id* among those that enter the third stage are returned to its sender either by being echoed or by circling whole ring.

5.3.2 Correctness of Algorithm $R1$

This section proves the correctness of algorithm $R1$. The first two lemmas show that there is at least one active processor that starts the third stage. The following lemma proves that at least one active processor is not prevented from entering the next phase by the faulty link during the first two stages.

Lemma 5.3.1 *Let L be the last phase of the second stage. Then, there is at least one of the messages sent by an active processor in phase v ($1 \leq v \leq L$) does not encounter the faulty link.*

Proof. Let p_i be an active processor in its phase $1 \leq v \leq L$. If there is no faulty link in the ring, then the lemma is trivial. Assume that there is one faulty link in the

ring. There are two cases determined by v.

- If $1 \leq v < \lfloor \log \ell \rfloor$, then processor p_i sends messages in both directions to distance $d = 2^v - 1 < 2^{\lfloor \log \ell \rfloor - 1} \leq \lfloor \ell/2 \rfloor$. Let f_l and f_r be the numbers of processors connected by non-faulty link on the left-hand side and on the right-hand side of the active processor p_i, respectively. Since there are at least ℓ processors in the ring and all processors are connected by non-faulty links, $f_l + f_r \geq \ell - 1$. Thus, either

$$\begin{aligned} f_l &\geq \left\lceil \tfrac{\ell-1}{2} \right\rceil > \left\lfloor \tfrac{\ell}{2} \right\rfloor \geq d, \text{ or} \\ f_r &\geq \left\lceil \tfrac{\ell-1}{2} \right\rceil > \left\lfloor \tfrac{\ell}{2} \right\rfloor \geq d. \end{aligned}$$

  Since d is the maximum distance that a message that is sent by p_i can travel, the lemma follows.

- If $\lfloor \log \ell \rfloor \leq v \leq L$, then an active processor p_i sends messages in both directions to distance $d = \lceil (\ell - k)/2 \rceil$ (k is the size of the segment of p_i) starting from processors at the end of its segment. Let f'_l (or f'_r) be the number of processors connected by non-faulty links on the left-hand (or right-hand-side, respectively) of p_i but do not belong to its segment. Since there are at least ℓ processors in the ring and all processor are connected by non-faulty links, $f'_l + f'_r \geq \ell - k$. Since f'_l and f'_r are integers, either

$$\begin{aligned} f'_l &\geq \left\lceil \tfrac{\ell-k}{2} \right\rceil = d, \text{ or} \\ f'_r &\geq \left\lceil \tfrac{\ell-k}{2} \right\rceil = d. \end{aligned}$$

  Since d is the maximum distance that a message that is sent by p_i can travel outside its segment, the lemma follows. □

This lemma demonstrates the importance of the second stage. If the size of a segment is greater than $\ell/2$, both f_l and f_r could be less than d and the first stage cannot guarantee that there exists at least one message that returns to its sender.

The next lemma proves that there is at least one processor that starts the third stage.

Lemma 5.3.2 *If there is one or more processors active in phase $v \leq L$, there is at least one active processor that enters phase $v+1$.*

Proof. The lemma follows from Lemma 5.3.1 if there is only one active processor in phase v. Assume that there is more than one processor active in phase u and that there is currently no processor active in a phase greater than v. The forwarding mechanism ensures that the message containing $(OtherPhase, OtherId)$ is not stopped by any processor whose $(phase, id)$ is less than $(OtherPhase, OtherId)$. Let p_i have the largest id among the processors active in phase v. Then Lemma 5.3.1 implies that the messages sent by p_i are not stopped by any processor in that phase. Also, at least one message does not encounter the faulty link. Thus, at least one message returns to processor p_i and the perocessor enters phase $v+1$. □

Lemma 5.3.2 implies that there are at least one processor that starts the third stage. The following lemma proves that at most three processors do so.

Lemma 5.3.3 *At most three processors start the third stage.*

Proof. Assume that four processors start the third stage. Let $p_{x_1}, \cdots, p_{x_2}, \cdots, p_{x_3}, \cdots, p_{x_4}, \cdots$ be a ring and that only processors p_{x_i} $(1 \leq i \leq 4)$ start the third stage. Let a_j $(1 \leq j \leq 4)$ be the number of processors between p_{x_i} and p_{x_j} $(j = (i \bmod 4) + 1)$ for $1 \leq i \leq 4$. Since there are at most $2\ell - 1$ processors in the ring (recall $\ell > \frac{u}{2} \geq \frac{n}{2}$),

$a_1 + a_2 + a_3 + a_4 \leq 2\ell - 5$. Since the size of the segment of each p_{x_i} is ℓ at the beginning of stage 3,

$$a_4 + a_1 \geq \ell - 1 \quad (1)$$

$$a_1 + a_2 \geq \ell - 1 \quad (2)$$

$$a_2 + a_3 \geq \ell - 1 \quad (3)$$

$$a_3 + a_4 \geq \ell - 1. \quad (4)$$

By adding up (1), (2), (3) and (4),

$$2(a_1 + a_2 + a_3 + a_4) \geq 4\ell - 4 = 2(2\ell - 2).$$

This is a contradiction to $a_1 + a_2 + a_3 + a_4 \leq 2\ell - 5$. Thus, it is clear that no more than three processors can start the third stage. Thus, the lemma follows. □

Lemmas 5.3.2 and 5.3.3 imply that at least one and at most three processors start the third stage. The following lemmas prove that there is only one active processor at the end of the third stage.

Lemma 5.3.4 *Any two processors that enter the third stage are connected by proven non-faulty links.*

Proof. All links within the segment of any active processor are proven non-faulty, since the active processor has received messages from processors at both ends of the segment.

By definition, the size of the segment of any active processor that enters the third stage is ℓ. There are at most $2\ell - 1$ processors in the ring. Thus, at least one processor belongs to both segments. Since both links of any processor that belongs to both segments are non-faulty links, the lemma follows. □

Since the faulty link fails before the beginning of the execution of the algorithm and any two active processors that enter the third stage are connected by non-faulty links, any message sent by an active processor in the third stage can reach all active processors.

The following lemma can now be proved.

Lemma 5.3.5 *There is always exactly one processor at the end of the third stage.*

Proof. Consider the processors in the highest phase during an execution of the algorithm. By Lemma 5.3.4, any two processors that start the third stage are connected by proven non-faulty links. Let p_i be the active processor with the largest *id* among the active processors at the beginning of the third stage. The killing rule ensures that all other processors become passive by receiving message(s) from p_i. Also, p_i receives all messages that it sent at the beginning of the stage. (Note that, if all links in a ring are proven non-faulty, the two messages return back to p_i without changing their directions.) □

The following correctness theorem follows the above lemma.

Theorem 5.3.1 *Let R be a asynchronous bidirectional ring with at most one fail-stop link failure that fails before the start of an algorithm (if it ever fails). If every processor in R knows an upper bound u and a lower bound ℓ such that $\ell > u/2$, and u and ℓ are same for all processors, then algorithm $R1$ solves election on R.*

Proof. It is clear that there is only one processor that can declare itself as a leader by the above lemma 5.3.5. Since all processors are connected even if there is one link failure, the elected leader can send its *id* to all other processors in the ring. Every

processor can terminate its execution of the algorithm when it is informed of the leader's *id*. □

5.3.3 The Message Complexity of Algorithm $R1$

This section analyzes the worst-case message complexity of algorithm $R1$ and the size of the largest message used in the algorithm.

Lemma 5.3.6 *Let L be the last phase of the second stage. Let k be the size of an active segment at the end of phase v ($1 \leq v < L$). Let I be an interval of k consecutive non-faulty links and $k+1$ processors. Then at most two processors in I enter phase $v+1$.*

Proof. Let $p_1, \cdots, p_i, \cdots, p_{k+1}$ be an interval I. Assume that processor p_i ($1 < i < k+1$) enters phase $v+1$. Then the message that contains phase v and $id(p_i)$ (where $id(p_i)$ is the *id* of processor p_i) has been forwarded by either p_1 or p_{k+1}, since the size of the segment is k. Then, the processor that forwards the message is not in a phase greater than v and the processor is in passive state after forwarding the message. The lemma follows. □

The following lemma counts the number of processors entering phase v (denoted by n_v) during the first stage of an execution of algorithm $R1$.

Lemma 5.3.7 *Let $1 \leq v \leq \lfloor \log \ell \rfloor$. Then, $n_v \leq 2\lceil n/2^v \rceil$.*

Proof. Consider a partition of the ring into processor-disjoint intervals each consisting of $2^v - 1$ non-faulty links, and an interval consisting of less than or equal to $2^v - 1$ non-faulty links. The faulty link (if it exists) does not belong to any of these intervals.

The total number of such intervals is $\lceil n/2^v \rceil$. Since the size of an active segment in phase $v < \lfloor \log \ell \rfloor$ is 2^v, the lemma follows by Lemma 5.3.6. □

The following corollary follows immediately from the lemma.

Corollary 5.3.1 *There are constant number of processors that enter the phase* $\lfloor \log \ell \rfloor$ *(in the second stage).*

Proof. By Lemma 5.3.7, the number of processors that enter phase $\lfloor \log \ell \rfloor$ is at most

$$\begin{aligned} 2\left\lceil \frac{n}{2^{\lfloor \log \ell \rfloor}} \right\rceil &< 2\left(\frac{n}{2^{\lfloor \log \ell \rfloor}} + 1\right) \\ &\leq 2\left(\frac{n}{2^{\log \ell - 1}} + 1\right) \\ &= 4\frac{n}{\ell} + 2. \end{aligned}$$

Since it is assumed that $\ell > \frac{u}{2}$, the inequality $\ell \leq n \leq u < 2\ell$ holds. Thus, at most 10 processors enter the second stage. □

Corollary 5.3.1 shows that only constant number of active processors enter the second stage. Lemma 5.3.3 implies that at most three processors start the third stage. The following lemma proves that the worst-case message complexity of algorithm $R1$ is $O(n \log n)$.

Lemma 5.3.8 *Let R be a asynchronous bidirectional ring with one fail-stop link failure that occurs before the beginning of the algorithm (if ever). Every processor in the ring knows the upper bound u and the lower bound ℓ of the ring size, and $\ell > u/2$. Then algorithm $R1$ solves election on R with worst-case message complexity $O(n \log n)$.*

Proof. It is first shown that the number of messages sent in the first stage of any execution of algorithm $R1$ is $O(n \log n)$. Let m_v be the number of message sent in

phase v ($1 \leq v \lfloor \log \ell \rfloor$). Every processor that enters phase v sends messages to distance $2^v - 1$ to both directions. Then, $m_v \leq n_v \cdot 4 \cdot 2^v$, where n_v is the number of processors active in phase v. By Lemma 5.3.7,

$$\begin{aligned} m_v &\leq 2\left\lceil \frac{n}{2^v} \right\rceil 2^{v+2} \\ &= 2^{v+3}\left\lceil \frac{n}{2^v} \right\rceil \\ &< 2^{v+3}\left(\frac{n}{2^v} + 1\right) \\ &= 8n + 2^{v+3}. \end{aligned}$$

Since the first stage has $\lfloor \log \ell \rfloor - 1$ phases, the number of messages sent during the first stage in an execution of algorithm $R1$ is $O(n \log n)$.

In each phase of the second stage, every active processor sends messages to distance less than ℓ in both directions. Let L be the last phase of the second stage. By Corollary 5.3.1, only a constant number of processors enter each phase v ($\lfloor \log \ell \rfloor \leq v \leq L$). Thus, the number of messages sent in every phase of the second stage is $O(n)$.

In the second stage, the size of a segment increases up to ℓ starting from $2^{\lfloor \log \ell \rfloor}/2$. Let k_v be the size of an active segment in phase v ($\lfloor \log \ell \rfloor < v < L$). Then, $k_{v+1} = k_v + \left\lceil \frac{\ell - k_v}{2} \right\rceil = \left\lceil \frac{\ell + k_v}{2} \right\rceil$. Thus, there are $c \log n$ (c is some constant) phases in the second stage. Therefore, the number of messages sent during the second stage of an execution of algorithm $R1$ is $O(n \log n)$.

As shown above, $O(n \log n)$ messages are sent during the first and the second stage. There are at most three processors that initiate at most two messages in the third stage. The messages in the third stage travel distance at most n. Thus, $O(n)$ messages are sent during the third stage. The lemma follows. □

Note that messages sent by an active processor in a phase of the second stage travels distance less than ℓ.

By recalling a result (by Goldreich and Shrira [21]) that the election problem on asynchronous bidirectional rings with at most one fail-stop faulty link requires $\Omega(n \log n)$ messages in the worst case, this section concludes with the following theorem.

Theorem 5.3.2 *Let R be a asynchronous bidirectional ring with one fail-stop link failures that occurs before the start of an algorithm if it ever fails. Assume that every processor in the ring knows an upper bound u, a lower bound ℓ of the ring size and the relation $\ell > u/2$, while the exact size of the ring is not known to any processor. The worst-case message complexity of any algorithm that solves election on R is $\Theta(n \log n)$.*

An analysis of the size of the largest message follows. There are four fields in every messages (except the one that carries the leader's identifier). A message field *stage* that distinguishes messages used in different stages need at most two bits, since there are four stages (including the stage that is used to broadcast the *id* of the elected leader to all other processors). Let b be the length of the longest identifier. Then, a message field $myId$ requires b bits. Since there are $c \log \ell$ phases in every execution of the algorithm (for some constant c), a message field *phase* requires $O(\log \log n)$ bits. Clearly, every distance that a message is sent is bounded by the size of the ring n. Thus, a message filed *dist* requires $O(\log n)$ bits. Therefore, the total number of bits for a message is $b + O(\log n)$.

5.3.4 Other Cases with $\Theta(n \log n)$ Worst-Case Message Complexity

This section shows that algorithm $R1$ can be used for the other three cases that also require $\Theta(n \log n)$ messages in the worst case.

It is clear that algorithm $R1$ can be used in the case where the exact size n of the ring is known to all processors by setting $l = u = n$. Goldreich and Shrira [20] presented an algorithm for this case. But their algorithm does not work if lower and upper bounds (ℓ and u) are given instead of the exact size n of the ring, since their algorithm relies on the information of the exact size n. Again, the lower bound $\Omega(n \log n)$ is valid for this case [21]. It is also clear that the algorithm $R1$ works without the knowledge of identifiers of neighbors.

The above upper bounds are asymptotically optimal. When identifiers of its two neighbors are known to all processors, the election problem requires $\Omega(n \log n)$ message in the worst case because the lower bound for election problem with comparison based algorithms by Frederickson and Lynch [16] holds even if the identifiers of two neighbors are known to all processors in a ring. (Note that finding identifiers of two neighbors takes $O(n)$ messages if there are no faulty links in a ring.)

5.4 Algorithms with Worst-Case Message Complexity $O(n \log n + (n - \ell)n)$

This section presents an algorithm that solves election problem in the following two cases:

- the identifiers of both neighbors and a lower bound ℓ are known to all processors (no upper bound is known),
- the identifiers of two neighbors and an upper bound u and a lower bound ℓ such that $\ell \leq u/2$ are known to all processors.

An algorithm (called algorithm $R2$) for the first case is presented in the following section. It is clear that algorithm $R2$ can be used for the second case since that provides more information.

5.4.1 Description of Algorithm $R2$

Algorithm $R2$ is based on algorithm $R1$. Since $\ell \leq u/2$, segments of processors active at the end of the second stage might not overlap when algorithm $R1$ is executed. Therefore, the third stage of algorithm $R1$ might not reduce the number of active processor to one. Algorithm $R2$ use the same first two stages used in the algorithm $R1$ but executes a procedure (called procedure P, see Figures 13 and 14) instead of the third stage of algorithm $R1$.

Let segments be defined as in algorithm $R1$. Then, *left end (or right end) of a segment* is the processor at the left (or right, respectively) end of the segment. Also, *left (or right) neighbor of a segment* is the processor to the left (or right, respectively) of the left (right) end of the segment. Two variables *SegLeftId* and *SegRightId* used to keep left and right neighbors' identifiers. Also, two message fields *leftId* and *rightId* are used to carry those information.

Procedure P relies on the following fact. If the size of a segment is $n - 1$ (for $n > 2$), the left neighbor of the segment is the right neighbor of the segment. Thus,

Procedure P

$last \leftarrow false$;
send $\langle phase, leftId, myId, rightId, lSize; left\rangle$;
send $\langle phase, leftId, myId, rightId, rSize; right\rangle$;
Wait for return of both messages and update *SegLeftId* and *SegRightId*;
send $\langle phase, leftId, myId, rightId, lSize + 1; left\rangle$;
send $\langle phase, leftId, myId, rightId, rSize + 1; right\rangle$;
goto Wait;
NewPhase:
$phase \leftarrow phase + 1$;
if $(receivedLink = right)$ then
    $SegRightId \leftarrow otherRightId$;
else /* $receivedLink = left$ */
    $SegLeftId \leftarrow otherLeftId$;
if $(receivedLink = right)$ then
    $rSize \leftarrow |otherDist| + 1$;
else /* $receivedLink = left$ */
    $lSize \leftarrow |otherDist| + 1$;
if $(last)$ then
    "Declare elected";
else if (the received message is the declaration message) then
    "Set leader's identifier, and forward the identifier, and exit"
else if $(SegLeftId = SegRightId)$ then
    $last \leftarrow true$;
send $\langle phase, leftId, myId, rightId, lSize + 1; left\rangle$;
send $\langle phase, leftId, myId, rightId, rSize + 1; right\rangle$;

Figure 13: Procedure P

```
Wait:
receive ⟨otherPhase, otherLeftId, otherId, otherRightId, otherDist; receivedLink⟩;
if (((otherPhase, otherId) = (phase, myId)) ∧ (state = active)) then
    goto NewPhase;
else if ((otherPhase, otherId) > (phase, myId)) then
    state ← passive;
    /* forward the message */
    if (otherDist = 1) then
        otherLeftId ← leftId;
        otherRightId ← rightId
        if (receivedLink = left) then
            receivedLink ← right;
        else /* receivedLink = right */
            otherDir ← left;
    if (receiveLink = left) then
        receivedLink ← right;
    else
        receivedLink ← left;
    send ⟨otherPhase, otherLeft, otherId, otherRight, otherDist − 1; otherDir⟩;
goto Wait;
```

Figure 14: Procedure P (continued)

it is possible for a processor to decide when to terminate an algorithm by checking the condition $SegLeftId = SegRightId$. Note that, if $n \leq 2$, election is trivial since all processors can determine n from their neighbors' identifiers and also know the identifiers of all processors.

The forwarding mechanism used in procedure P is similar to the one used in the second stage of algorithm $R1$. But both messages sent by the active processor of a segment try to extend the size of the segment by one. When a message travels back to its sender, it carries the identifier of the left (or right) neighbor of the processor from which it starts to travel back. The killing rule used is in procedure P is exactly the same as that of algorithm $R1$.

Procedure P operates as follows: Upon entering the procedure P, every active processor sends two message in both directions to collect identifiers of left and right neighbors of the segment. Upon entering a new phase in procedure P, every active processor sends out messages in both directions to a distance that expands the size of current segment by 1 and waits for return of one of those messages. If such a message returns, the processor enters the next phase.

Eventually, the size of segment grows to $n - 1$ and the left neighbor and the right neighbor of the segment are the same processor. If this condition occurs at a processor, the processor enters the next phase. There are at most two such processors since the size of segment is n The processor that receives one more message declares itself elected.

5.4.2 Correctness of Algorithm $R2$

The correctness of algorithm $R2$ is partly based on that of algorithm $R1$ since the first and second round of the two algorithms are same. Lemma 5.3.2 implies that at least one processor that executes procedure P.

The following lemmas show that at least one active processor declares itself elected during an execution of procedure P.

Lemma 5.4.1 *Let L be the last phase (declaration phase) of procedure P. If there is at least one processor active in phase $v < L$, at least one active processor enters phase $v + 1$.*

Proof. If there is only one processor active in phase v, the lemma is trivially true. Assume that more than one processor is active in phase v and that there are currently no processors in phases higher than v. Let p_i be the processor with the largest *id* among the processors active in phase v. The killing rule ensures that a message sent by p_i is not stopped by any other processor.

Every active processor in phase v sends two message each of which tries to extend the size of the segment by one. It is clear that at least one of these messages does not try to cross the faulty link. Thus, at least one message returns back to its sender. The lemma follows. □

It has been shown that at least one processor enters the last phase of procedure P. The following lemma proves that detecting the termination is possible.

Lemma 5.4.2 *During an execution of procedure P in algorithm $R2$ on a ring of size n, the size of an active processor's segment is $n - 1$ if and only if the processor has $SegLeftId = SegRightId$.*

Proof. Let p_i be such an active processor. Since *SegLeftId* (or *SegRightId*) is the identifier of the left (or right, respectively) neighbor of the segment of p_i, *SegLeftId* = *SegRightId* when the size of segment is $n-1$. The other direction is trivially true. □

The correctness theorem of the algorithm $R2$ follows immediately from above lemmas.

Theorem 5.4.1 *Let R be a asynchronous bidirectional ring with at most one fail-stop link failure that fails before the start of an algorithm (if ever). If every processors in R knows the identifiers of its two neighbors and a lower bound ℓ, then algorithm $R2$ solves election on R.*

Proof. Lemma 5.4.1 implies that at least one active processor enters the last phase of procedure P. Since the size of the segment of the processor that enters the last phase is n, there is only one such processor by the definition of segment. Thus, the theorem follows. □

5.4.3 Analysis of Algorithm $R2$

This section analyzes the worst-case message complexity and the size of the largest message of algorithm $R2$.

Lemma 5.4.3 *The number of messages sent during an execution of procedure P in algorithm $R2$ is $O((n-\ell)n)$.*

Proof. It is clear that the size of the segment of an active processor in the k^{th} phase of procedure P is $\ell + k$. Let n_k be the number of active processors in the k^{th} phase of procedure P. Then, $n_k \leq 2\left\lceil \frac{n}{\ell+k} \right\rceil$ (the proof is similar to that of Lemma 5.3.7).

Since messages sent by an active processor in the k^{th} phase travel distance at most $2(\ell + k)$, the number of message sent during an execution of procedure P is less than or equal to $\sum_{k=1}^{n-\ell} 2(\ell + k)2\left\lceil \frac{n}{\ell+k} \right\rceil = O((n - \ell)n)$. □

Lemma 5.3.8 implies that the first and second stages of algorithm $R2$ require $O(n \log n)$ messages in the worst case. The following theorem follows immediately.

Theorem 5.4.2 *Let R be a asynchronous bidirectional ring with one fail-stop link failure that occurs before the beginning of the algorithm (if ever). Every processor in the ring knows the identifiers of its two neighbors and a lower bound ℓ. There exists an algorithm that solves election on R with worst-case message complexity $O(n \log n + (n - \ell)n)$.*

An analysis of the largest message size is as follows. There are two message fields ($leftId$ and $rightId$) that are only used in procedure P. Since those two fields carry the identifiers, $2b$ (where b is the length of the longest identifier) additional bits are required. Thus, the maximum number of bits required for messages exchanged during an execution of algorithm $R2$ is $3b + O(\log n)$.

5.5 An $\Omega(n \log n + (n - \ell)n)$ Lower Bound

This section proves a lower bound on the worst-case message complexity for the following cases:

- the identifiers of neighbors and an upper bound u and a lower bound ℓ such that $\ell \leq u/2$ are known to all processors,
- the identifiers of its two neighbors and a lower bound ℓ is known to all processors.

The proof of the lower bound for the first case is also valid for the second case, since less information is available in the second case.

Let R be an asynchronous bidirectional ring of size n with at most one fail-stop link failure that fails before the start of an algorithm (if ever). Let a *k-segment* be k consecutive processors ($k \leq n$) connected by non-faulty links from R. Let $\mathcal{A}$ be an algorithm that correctly solves the problem of election on R in which the identifiers of neighbors, a lower bound ℓ, and an upper bound u ($\ell \leq u/2$) are known to all processors but the exact size n is not.

At any point of an execution of an algorithm $\mathcal{A}$, a k-segment is said to *have a potential leader* if there is at least one processor in the k-segment that can correctly determine the identifier of the eventual leader without receiving any more messages if the size of ring is exactly k. It is clear that there should be a k-segment having potential leader(s) at the end of any execution of an algorithm that elects a leader on rings of size k.

The following lemma shows the existence of an ℓ-segment with potential leader(s) during some executions of $\mathcal{A}$ on a ring of size n.

Lemma 5.5.1 *There exists at least one ℓ-segment with potential leader(s) regardless of u and n, during some executions of $\mathcal{A}$ on R.*

Proof. If $n = \ell$, there is only one ℓ-segment in R. Since algorithm $\mathcal{A}$ correctly elects a leader when $n = \ell$, the ℓ-segment has potential leader(s) at termination regardless of u.

Consider an execution of algorithm $\mathcal{A}$ on ring $R' = p_{s_1}, \cdots, p_{s_\ell}$ of size ℓ such that the link between processors p_{s_ℓ} and p_{s_1} is the faulty link and every processor know u

and ℓ. Then, $p_{s_1}, \cdots, p_{s_\ell}$ is a ℓ-segment with a potential leader(s) at some point of the execution. Now, consider a ring $R'' = p_1, \cdots, p_{s_1}, \cdots, p_{s_\ell}, \cdots, p_n$ where the links between processors p_{s_1-1} and p_{s_1} between processors P_{s_ℓ} and $p_{s_\ell+1}$ are very slow (one may be the faulty link). Then, there is an execution of algorithm $\mathcal{A}$ on R'' such that P_{s_ℓ} and $P_{s_\ell+1}$ becomes an ℓ-segment with potential leader since faulty links and slow links are not distinguishable. Thus, lemma holds for any u and n such that $u \geq n \geq \ell$.

□

The following lemma proves that algorithm $\mathcal{A}$ requires at least $(n-\ell)n$ messages.

Lemma 5.5.2 *Let R be an asynchronous bidirectional ring with at most one fail-stop link failure that fails before the start of an algorithm (if ever). Let $\mathcal{A}$ be an algorithm that solves election on R when a lower bound ℓ and an upper bound u ($\ell \leq u/2$) are known to all processors but the exact size n is not. Then some executions of $\mathcal{A}$ on R requires $\Omega((n-\ell)n)$ messages.*

Proof. By Lemma 5.5.1, there is at least one ℓ-segment with a potential leader. If $n = \ell$, the lemma is trivial. Assume $n > \ell$. Since $2\ell \leq u$, there may be more than one ℓ-segment with a potential leader in the ring. Therefore, an ℓ-segment cannot decide the leaders *id*. Thus, at least one ℓ-segment should receive more messages. Let $p_{a_x}, \cdots, p_{a_1}, p_{s_1}, \cdots, p_{s_\ell}, p_{b_1}, \cdots, p_{b_y}$ (where $x + y + \ell = n$ and $|x - y| \in \{0, 1\}$) be a ring such that $p_{s_1}, \cdots, p_{s_l}$ is an ℓ-segment that receives more messages. Let the link between processor p_{a_x} and p_{b_y} be the faulty link. (Note that the faulty link could be any link not in the ℓ-segment.) Consider an execution of algorithm $\mathcal{A}$ in which messages sent by processors p_{a_i} and p_{b_i} are delivered to some processor between those two processors after the ℓ-segment is formed in ascending order of i. Messages that are sent by p_{a_i} and p_{b_i} should be delivered to one of processors $p_{a_i}, \cdots, p_{s_1}, \cdots, p_{s_\ell}, \cdots, p_{b_i}$.

Thus, the number of messages required is $\Omega((n-l)n)$ since $\ell + 2i - 1$ messages are needed for $1 \leq i \leq \left\lfloor \frac{n-\ell}{2} \right\rfloor$. □

By recalling the result (by Goldreich and Shrira [21]) that election requires $\Omega(n \log n)$ messages when the size of ring is known to any processors, the following corollary immediately follows.

Corollary 5.5.1 *Let R be an asynchronous bidirectional ring with at most one fail-stop link failure that fails before the start of an algorithm (if ever). Let $\mathcal{A}$ be an algorithm that solves election on R when a lower bound ℓ and upper bound u ($\ell \leq u/2$) are known to all processors but the exact size n is not. Then, any execution of $\mathcal{A}$ on R requires $\Omega(n \log n + (n-\ell)n)$ messages.*

This section concludes with the following two theorems.

Theorem 5.5.1 *Let R be an asynchronous bidirectional ring with at most one fail-stop link failure that fails before the start of an algorithm (if ever). Then, election requires $\Theta(n \log n + (n-\ell)n)$ messages in the worst case if every processor in R knows identifiers of two neighbors, a lower bound ℓ, and an upper bound u ($\ell \leq u/2$).*

Proof. By Theorem 5.4.2, there is an algorithm that solves the election problem on R when every processor in R knows identifiers of two neighbors and a lower bound ℓ. Thus, the theorem immediately follows from Corollary 5.5.1. □

Theorem 5.5.2 *Let R be an asynchronous bidirectional ring with at most one fail-stop link failure that fails before the start of an algorithm (if ever). Then, election requires $\Theta(n \log n + (n-\ell)n)$ messages in worst case if every processor in R knows identifiers of two neighbors, a lower bound ℓ, and an upper bound u and such that $\ell \leq \frac{u}{2}$.*

Proof. The election requires $\Omega(n \log n + (n-\ell)n)$ message since Corollary 5.5.1 holds even if an upper bound is not available to every processors. Also, election is possible with worst-case message complexity $O(n \log n + (n - \ell)n)$ by Theorem 5.4.2. □

5.6 An Impossibility Result

There are two remaining cases:

- the identifiers of two neighbors, an upper u of the size of the ring, and the exact size n are not known but a lower bound ℓ is known to all processors,
- the identifiers of two neighbors are not known but an upper bound u and a lower bound ℓ of the size of the ring is known such that $\ell \leq u/2$.

Goldreich and Shrira [20] showed that election is impossible in the first case for $\ell = 1$.

The following theorem proves that election is impossible in the second case as well.

Theorem 5.6.1 *Let R be a asynchronous bidirectional ring with at most one fail-stop link failure that fails before the start of an algorithm (if ever). Then there is no distributed algorithm that solves election on R, if every processor knows its own identifier and an upper bound u of the size of R such that $\ell \leq \frac{u}{2}$.*

Proof. Assume to the contrary that there is such an algorithm $\mathcal{A}$. Consider $\mathcal{A}$'s executions on two different rings $p_1 \cdots p_i \cdots p_n$ (where the link between p_n and p_1 is the faulty link) and $p_{n+1}, \cdots, p_j, \cdots, p_{2n}$ (where the link between p_n and p_1 is the faulty link). Assume also that the two rings have disjoint sets of identifiers. Since algorithm $\mathcal{A}$ solves the problem, leaders p_i $(1 \leq i \leq n)$ and p_j $(n+1 \leq j \leq 2n)$ are

elected from each execution. Now consider another execution of algorithm $\mathcal{A}$ on the ring $p_1 \cdots p_i \cdots p_n, p_{n+1}, \cdots p_{2n}$, where the link between p_n and p_{n+1} is very slow and the link between p_1 and p_{2n} is faulty. The algorithm should elect a leader since $\ell \leq n$ and $u \geq 2n$. Since the slow link and the faulty link can not be distinguished, two sets of processors ($p_1, \cdots, p_n$ and $p_{n+1}, \cdots, p_{2n}$) may act as in their original executions and elect two leader p_i and p_j in a ring. This is a contradiction and the theorem follows. □

The above theorem holds even if upper bound u is not known to all processors, since u can be considered as ∞ if u is not known. Note that theorem implies the impossibility result by Goldreich and Shrira. The reverse is not true.

5.7 Concluding Remarks

This chapter considered the effect of incomplete knowledge of network size on the election problem for asynchronous rings of processors with at most one undetectable fail-stop link failure. All possible cases of a lower bound ℓ and an upper u are considered. The availability of two neighbors' identifiers are also considered for each case, since election becomes possible with this additional information for some cases. The results are summarize in Table 5.

The quality of a lower bound ℓ (how close it is to the size n of a ring) directly

	$\ell \leq \frac{u}{2}$	$u > \ell > \frac{u}{2}$
Know Neighbors	$\Theta(n \log n + (n - \ell)n)$	$\Theta(n \log n)$
Does Not Know Neighbors	Impossible	

Table 5: Election with Incomplete Knowledge of Ring Size

affects the worst-case message complexity while an upper bound u does not. On the other hand, election is not possible even if a lower bound is very close to n if the exact size of the ring is not known and the known upper bound is not tight enough (i.e., $u \geq 2\ell$), without additional information such as identifiers of two neighbors. Note that all results by Goldreich and Shrira [19, 20, 21] are subsumed by the above results.

Chapter 6

Election on Square Meshes with Link Failures

6.1 Introduction

This chapter considers the election problem for asynchronous square meshes with fail-stop link failures. Since the communication is asynchronous, the failed links cannot be detected.

As shown in Chapter 3, link failures have been studied by several researchers recently. While Peterson [35] and Abu-Amara [2] considered the election problem for asynchronous and synchronous square meshes without failures, this chapter considers asynchronous square meshes with undetectable fail-stop failures. Note that the fail-stop failure of a link can be easily detected in synchronous systems by sending messages over the link and waiting for the acknowledgment from the processor connected by the link.

Three cases are considered: $t < \sqrt{n}$, $t < 2\sqrt{n}$, and $t \geq 2\sqrt{n}$, where t is the number of maximum faulty links in a square mesh and n is the number of processors in the square mesh. An algorithm of worst-case message complexity $O(n \log t)$ is obtained for the case $t < \sqrt{n}$. When $t < 2\sqrt{n}$, an algorithm of worst-case message complexity $O(n \log n)$ is presented. For the case $t \geq 2\sqrt{n}$, an impossibility result is obtained.

The algorithms are correct even for cases in which some processors are completely disconnected due to faulty links. The algorithm for the case $t < \sqrt{n}$ cannot be used for the case $t < 2\sqrt{n}$. The algorithm for the case $t < 2\sqrt{n}$ can be used for the case $t < \sqrt{n}$ but may have worse performance.

Section 6.2 presents some assumptions made for this chapter and defines square meshes. Section 6.3 presents an algorithm for the case $t < \sqrt{n}$. The algorithm for the cases $t < 2\sqrt{n}$ and the impossibility results are given in Sections 6.4 and 6.5, respectively.

6.2 Preliminaries

A square mesh of n processors is defined as a wrap-around square of n processors, with $\sqrt{n}$ processors on each side, with each row and each column forming a ring. (Figure 15 shows a square mesh of n processors.) Each row of processors is called a horizontal ring and each column of processors is called a vertical ring. As shown in Figure 15, vertical rings are denoted by v_i $(1 \leq i \leq \sqrt{n})$ and horizontal rings are denoted by h_j $(1 \leq j \leq \sqrt{n})$. The processor that belongs to the vertical ring v_i and horizontal ring h_j is denoted by p_{ij}.

The communication in square meshes is asynchronous and bidirectional. Messages sent over a link are delivered in the order they are sent (FIFO). Any links may fail by stopping and but the total number of failed link cannot be greater than t. The value of t and the relation between t and n (such as $t < \sqrt{n}$) are known to all processors, but the value of n is not known. The lower bound on size n based on t (such as t^2) is known to all processors. It is assumed that all faulty links fail before the start of

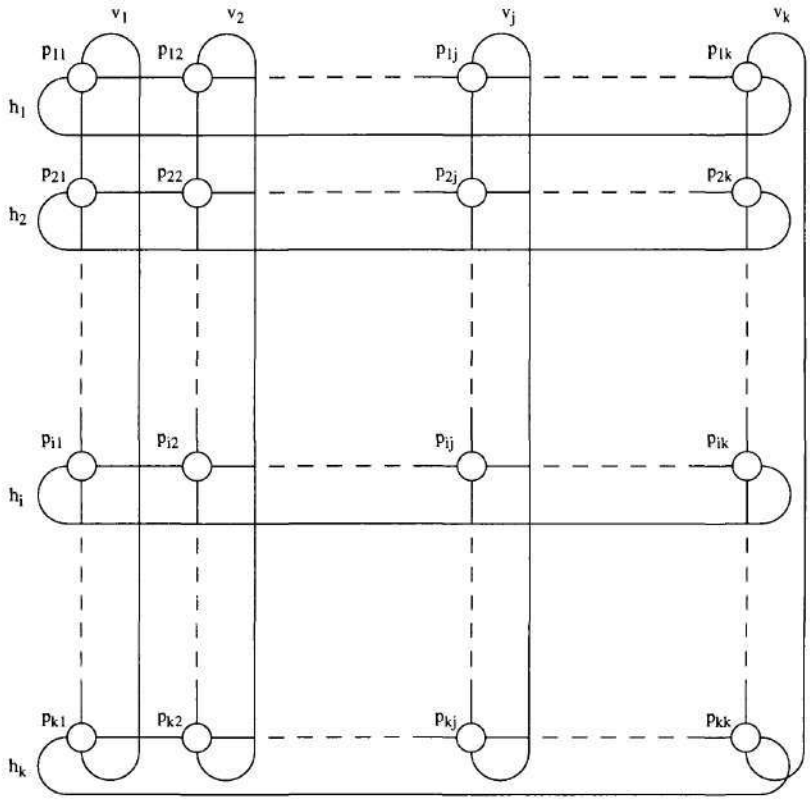

Figure 15: A Square Mesh of Size n

an algorithm.

It is assumed that a sense of global direction is available. The sense of global direction in square meshes means that a processor can distinguish its four links by its names (such as up, down, left, right) in uniform fashion. For example, a processor's right link is the left link of the processor that is connected by that link. Also, the topology of the network is known to all processors. Finally, it is assumed that all processors in the mesh start an algorithm spontaneously. (This assumption can be relaxed as explained later.)

In evaluating algorithms, worst-case message complexity is used as a primary measure. The maximum size of messages is also considered.

6.3 An Algorithm for the case of $t < \sqrt{n}$

6.3.1 Overview of Algorithm $M1$

A high level description of algorithm $M1$, which solves the election problem on square mesh of n processors with $t < \sqrt{n}$, is presented in this section. A detailed description of the algorithm follows in the next section.

Algorithm $M1$ is based on Peterson's election algorithm [35] for square meshes without failures. Peterson's algorithm works as follows: The algorithm proceeds in phases. Each processor in phase i sends a message distance d ($d = \alpha^i$, where α is some constant) to right, then d down, d left, and finally d up back to itself. In other words, each processor is trying to mark off the boundary of a square distance d on a side. Some squares will overlap each other, and only one square among overlapped squares can be completed. Processors that complete their squares move to the next phase. Eventually, there will be a few processors that see "wrap around" (i.e., the message sent by a processor returns back to its sender from its left rather from the below), and one of those processors is elected after constant number of phases.

Peterson's algorithm is not correct in the presence of link failures, since it is possible that the only processor that can move to some phase i cannot complete the square because of a faulty link and the algorithm would not proceed. Since there are at most t link failures, this situation can be avoided if each processor sends $t + 1$ messages that follow different paths. There are two main difficulties in implementing this idea. A processor needs to send messages to $t + 1$ different paths, but each processor has only four links. Also, a naive implementation of this idea could be "send at least $t + 1$ messages following different paths for each message in Peterson's

algorithm". But this results in an algorithm with worst-case message complexity $O(nt)$ since worst-message complexity of Peterson's algorithm is $O(n)$.

Algorithm $M1$ overcomes these problems as follows. First, the algorithm builds groups of $t+1$ consecutive processors in vertical rings. No faulty links are present between processors in such a group (called a "trying segment"). Then, each trying segment tries to mark off $t+1$ squares as shown in Figure 16. In the figure, the thick line denotes a trying segment. All processors in a trying segment share same *id* (called *tid* of the square) when they mark off their squares. All processors in the trying segment try to mark off squares of different side distance (their boundaries are shown as arrowed lines). Since there are at most t faulty links and no boundaries of squares by a processor cross each other, at least one square can be completed if the largest square distance is less than $\sqrt{n}$. (In the figure, the shaded square is completed by the second processor from the bottom with smaller size than the one currently being tried to mark off.) When a square is completed, all processors in the trying segment try to mark off larger squares.

Algorithm $M1$ consists of three stages. Each stage is implemented as a separate procedure: procedure *BuildSeg* for the first stage, procedure *Compete* for the second stage, and procedure *PostWrapAround* for the last stage. There are $\sqrt{n}$ concurrent and independent executions (one for each vertical ring) of procedure BuildSeg, while the other two procedures are executed only once during an execution of algorithm $M1$.

Spontaneous start-up of procedure BuildSeg (the first stage) by all processors is assumed. If some processors started independently, each could send "start up" messages in all four directions. Upon receiving a "start up" message, a processor

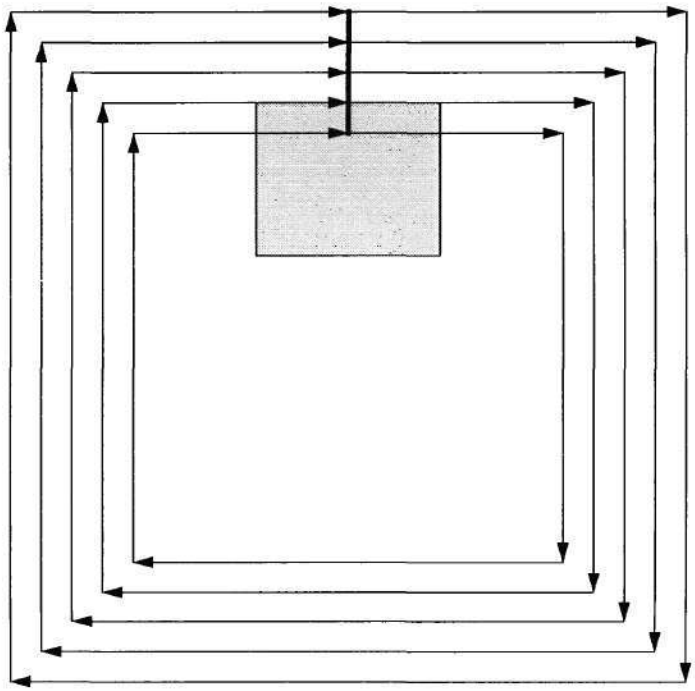

Figure 16: A Trying Segment

relays the message to all four directions except the direction from which it received the message (if it has not done so) and starts the algorithm. This start-up procedure requires $O(n)$ messages, since every link transmits the "start up" message at most twice.

Any processor that finishes execution of a procedure enters the next stage and starts to execute the appropriate procedure. If a processor receives a message used in lower stage than that which it is currently executing, it ignores and discards the message. If a processor receives a message used in higher stage than it is currently executing, the processor terminates the execution of the current procedure and starts the appropriate procedure by responding to the message received.

Algorithm $M1$
/* Code for processor p_{ij} */
Stage 1: /* Construct "trying segments" */
Execute procedure BuildSeg on vertical ring v_j;
Stage 2: /* Reduce the number of "trying segments" to some constant. */
Execute procedure Compete;
Stage 3: /* Reduce the number of "trying segments" to one. */
Execute procedure PostWrapAround;

Figure 17: Algorithm M1

The following describes the main task of each stage of algorithm $M1$ (see Figure 17). Mechanisms to accomplish these tasks are explained in the following section. The goal of first stage (procedure BuildSeg) is to build *trying segments*. At the end of the first stage, there are at most $\sqrt{n}/(t+1)^2$ active processors (since a trying segment consists of $t+1$ processors) and at least one active processor in a vertical ring without faulty links. Once a trying segment is built, every processor in the trying segment starts to execute procedure Compete (the second stage).

Upon entering the second stage, all processors in a trying segment try to mark off the boundaries of squares of side distances from d to $d+t$ (initially $d = t+1$). As the algorithm proceeds, the number of *active trying segments* that are trying to mark off squares decreases while squares get larger. If distance d for a processor becomes greater than or equal to $\sqrt{n}$, all processors in the trying segment detect this and enter the third stage (procedure PostWrapAround). As will be shown later, only constant number of trying segments start the procedure PostWrapAround.

In the third stage, every processor in a trying segment tries to mark off a cross of distance $\sqrt{n}$ as shown in Figure 18. In the figure, a trying segment is shown by the

dark line and the shaded squared is a square marked off before the start of procedure PostWrapAround. *Wrap-around* is the condition that occurs when a stage 2 message sent by a processor in the same direction passes through all processors in the ring and returns to its originator without changing direction. Processors that see wrap-around conditions on both horizontal and vertical rings are called *wrap-around processors*. Note that wrap around processors are not in a trying segment; their function is to provide link-disjoint crosses. (In the figure, wrap around processors are marked by circles.)

The number of active segments is reduced in procedure PostWrapAround in exactly same way as in procedure Compete. Since procedure PostWrapAround starts with a constant number of active segments, the number of active segments is reduced to one in a constant number of phases. After a constant number of phases, processors in the only trying segment declare their common *tid* as leader's *id* by sending the leader's *id* to all processors in the mesh and the algorithm terminates.

6.3.2 Detailed Description of Algorithm $M1$

In describing algorithm $M1$, the following conventions are used. The four links of a processor are referred with names *right*, *down*, *left*, and *up* and assumed to be numbered 0, 1, 2, and 3, respectively. Note that this is possible since the availability of a global sense of direction is assumed. The statement "send $\langle s; var_1, \cdots, var_k; dir_1, \cdots, dir_r \rangle$" is to be interpreted as "send a message for the stage s whose content is $var_1, \cdots, var_k$ to directions $dir_1, \cdots, dir_r$ $(1 \leq r \leq 4)$". The statement "receive $\langle var_1, \cdots, var_k; dir \rangle$" is to be interpreted as "receive a message and store the content of the message to variables $var_1, \cdots, var_k$, also store the direction from which the message is received into

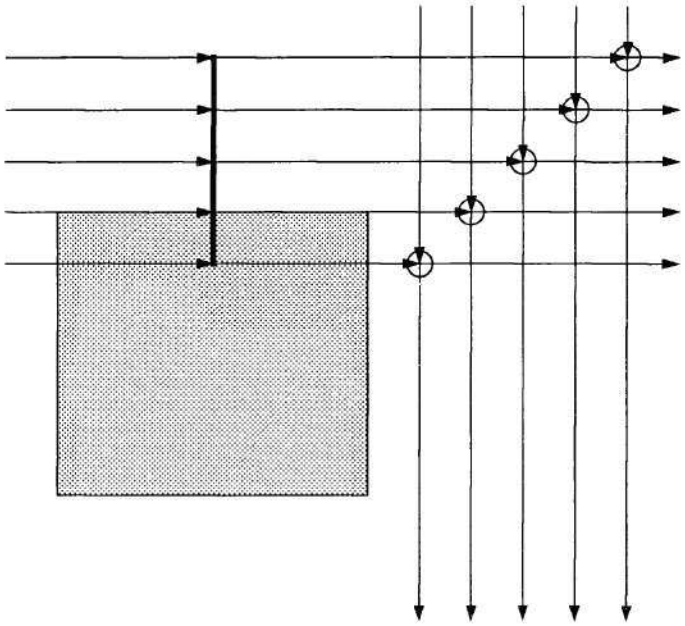

Figure 18: A Trying Segment after Wrap Around

dir". Upon receiving a message, the stage for which the message is sent is checked first. If the message is for the same stage as the stage the receive statement is executed, then the message is interpreted as stated above. If the stage is lower, then the message is ignored. If the stage is higher, then the appropriate procedure is invoked.

6.3.2.1 Description of Procedure BuildSeg

The main tasks of procedure BuildSeg are building trying segments and reducing the number of active processors. Recall that a trying segment is $t + 1$ consecutive processors within a vertical ring without faulty links between them. It is necessary to find such groups of processors in every vertical ring. It is desirable not to have too many trying segments initially, since too many trying segments could result in higher worst-case message complexity.

Procedure BuildSeg (Figure 19) is based on election algorithm DG (for unidirectional rings) presented in Chapter 4. Note that most of the efficient ring election algorithms can be used as a base of this procedure.

Procedure BuildSeg is executed concurrently and independently on every vertical ring. (Processor p_{ij} participates in an execution of procedure BuildSeg on vertical ring v_j.) A processor is in *active* or *passive* state during an execution of the procedure. The state of a processor is stored in the local variable *state*. Initially, all processors are active. Processors operate in phases. (A local variable *phase* stores a processor's phase number.) A local variable *tid* is used to store the temporary identifier of a processor. Initially, *tid* of a processor is set to its own identifier. To achieve $O(n \log n)$ worst-case message complexity, a variable *parity* is used. The *parity* is true for every other phase starting from the first phase. It is false for all other phases.

Upon entering a new phase i, an active processor sends its *tid* over its up link and waits for a message to be delivered over its down link. If a processor receives a message carrying *tid* that is greater (or less, respectively) than its current *tid* when *parity* is true (or false, respectively), then it sets its *tid* to the value of received *tid* and starts the next phase. Otherwise, it becomes passive. Passive processors always relay messages that are received. The procedure BuildSeg proceeds up to phase $\lceil \log_\phi t^2 \rceil$ (where $\phi = \frac{1+\sqrt{5}}{2}$) or until a leader of the ring is elected (the original election algorithm DG proceeds until a leader is elected). Note that some processors may not reach phase $\lceil \log_\phi t^2 \rceil$ because of faulty links.

Procedure BuildSeg

```
phase ← 0;
tid ← id;
state ← active;
parity ← true;
done ← false;
send⟨tid; up⟩;
while (¬done ∧ (phase ≤ ⌈log_φ t²⌉)) do
    receive⟨nid; OtherDir⟩;
    phase ← phase + 1;
    if (nid = id) then
        done ← true;
    case state of
    active:
        if ((nid > tid) ⊕ parity†) then
            tid ← nid;
            send⟨tid; up⟩;
        else
            state ← passive;
        parity ← ¬parity;
    passive:
        send⟨nid; up⟩;
if (state = active) then
    Build a "trying segment" by sending a special message over up link;
```

† ⊕ denotes *exclusive or*.

Figure 19: Procedure BuildSeg

6.3.2.2 Building a Trying Segment

An active processor that enters phase $\lceil \log_\phi t^2 \rceil$ or that becomes a leader starts to build a trying segment. Since there are at least t passive processors above an active processor and t is known to all processors, an active processor is able to build a trying segment. An active processor builds a trying segment as follows. Upon entering phase $\lceil \log_\phi t^2 \rceil$ or becoming a leader, an active processor sends a special message "build segment" with distance $d = t$ and its *tid* over its up link and waits for its return. Upon receiving a special message from its down link, a passive processor relays the message over its up link with distance $d - 1$ if $d \neq 0$. The passive processor that receives a special message "build segment" with $d = 0$ and passive processors that wait for the return of the special message perform the following actions:

- sets its *TryNum* to $t - d$,
- sets its *tid* with delivered *tid*,
- returns the message with $d + 1$ over its down link, and
- starts the next stage (procedure Compete).

The active processor that initiates the special message does the same except relay the message over their down link.

Thus, there there are t passive processors above the active processor, a trying segment is successfully built. All processors in the trying segment start the next stage. Note that any up links to a passive processor are non-faulty, since a processor becomes passive only by receiving a message and all faulty links fail before the start of an algorithm.

6.3.2.3 Description of Procedure Compete

The main task of procedure Complete is to decrease the number of active trying segments to some constant. Procedure Compete (Figures 20 and 21) also proceeds in phases. (Note that no active or passive states are used in procedure Compete.) Upon entering phase i (Compete starts it execution with $phase = 1$), a processor in a "trying segment" in phase i tries to mark off the boundary of a square with side $d = (t+1)c^{i-1} + 2\,TryNum$ (c is a constant whose value will be discussed later). This is done with two variables *Dist* (the distance to travel) and *Dir* (the direction that it is sent to). With those two variables, a message can be sent distance $\lfloor d/2 \rfloor$ to right then d down, d left, d up and $\lceil d/2 \rceil$ right again to the starting processor. (Refer to Figure 16.)

Before a processor starts to mark off a square, it checks whether a wrap-around condition occurs. Since the size of square mesh is not known to processors, a processor checks the wrap-around condition by sending a message. This is the task of function *IsWrapAround* . When procedure IsWrapAround is invoked with d, it sends out a special message of type "Checking" to right with distance d and *tid* of the processor. Upon receiving a message with type "Checking", a processor sends it to right with $d-1$ if $d \neq 1$. If $d = 1$, the processor returns the message back to left. When a message with same *tid* is received from the left link, the a wrap-around condition occurs and the procedure IsWrapAround returns true, otherwise it returns false.

If a call to function IsWrapAround returns true, the processor informs other processors in the trying segment. All processors in the trying segment then enter the third stage.

Competition between trying segments in stages 2 and 3 is resolved as follows. (This

```
Procedure Compete /* Code for processor p_ij */
MyPhase ← 1;
MoveOn:
if (TryNum = 0) then
    send ⟨2; MoveOn, MyPhase; 3⟩ /* 3 = up */
else if (0 < TryNum < t) then
    send ⟨2; MoveOn, MyPhase; 1, 3⟩ /* 1 = down */
else
    send ⟨2; MoveOn, MyPhase; 1⟩;
repeat
    DidSeeSmaller ← false; WasSeenBySmaller ← false; SawNone ← false;
    Dist ← (t + 1)c^(MyPhase−1) + 2 · TryNum;
    if (IsWrapAround(Dist)) then goto Exit;
    send ⟨2; Looking, MyPhase, tid, TryNum, ⌈Dist/2⌉; 0⟩; /* 0 = right */
    repeat
        receive ⟨Stage; Type, OtherPhase, OtherTryNum, OtherId, OtherDist; OtherDir⟩;
        if ((Type = MoveOn) ∧ (OtherPhase > MyPhase)) then
            MyPhase ← OtherPhase; goto MoveOn;
        if ((tid ≠ OtherId) ∧ (OtherPhase > MyPhase)) then
            goto Relay;
        else if (OtherPhase = MyPhase) then
            case Type of
            Looking, SawSmaller :
                if (OtherId > tid) then
                    goto PreRelay;
                else if (tid = OtherId) then
                    if (Type = Looking) then
                        SawNone ← true;
                    else
                        DidSeeSmaller ← true;
                else
                    WasSeenBySmaller ← true;
            SeenbySmaller :
                    WasSeenBySmaller ← true
    until (SawNone ∨ (DidSeeSmaller ∧ WasSeenBySmaller))
    MyPhase ← MyPhase + 1; goto MoveOn;
until (false)
Exit:
Inform all processors in the trying segment to start Stage 3;
```

Figure 20: Procedure Compete

```
PRERELAY: Type ← SawSmaller;
RELAY:
SentSeenby ← false; MyPhase ← OtherPhase; SaveId ← OtherId;
if (OtherDist = 0) then
    SaveDist ← (t + 1)c^(OtherPhase−1) + 2 · OtherTryNum;
    if (Dir = 3) then
        SaveDist ← ⌊SaveDist/2⌋;
    SaveDir ← (Dir + 1) mod 4;
else
    SaveDist ← OtherDist − 1;
    SaveDir ← (Dir + 2) mod 4
send ⟨2; Type, MyPhase, SaveId, OtherTryNum, SaveDist; SaveDir⟩;
repeat
    receive ⟨Type, OtherPhase, OtherId, OtherTryNum, OtherDist; OtherDir⟩;
    if ((Type = MoveOn) ∧ (OtherPhase > MyPhase)) then
        MyPhase ← OtherPhase; goto MoveOn;
    if (OtherPhase > MyPhase) then
        goto Relay;
    if (OtherPhase = MyPhase) then
        if (OtherId > SaveId) then
            goto PreRelay;
        else if (¬SentSeenBy) then
            send ⟨2; SeenBySmaller, MyPhase, SaveId; SaveDist⟩;
            SentSeenBy ← true;
until (false)
```

Figure 21: Procedure Compete (continued)

is the same mechanism used in Peterson's algorithm.) If a message does not encounter the boundary of other active processor, it completes its boundary and enters the next phase. (Note that boundaries of processors in the same trying segments never cross each other on the same phase since all paths in a trying segment are link disjoint.) If it encounters the boundary of a processor with smaller *tid*, then it continues marking the boundary, but with message type *SawSmaller*. If it encounters the boundary of a processor with larger *tid*, then it sends a message of type *SeenBySmaller* along the boundary of the other processor. The boundary of the processor with the smaller *tid* will not be completed. A processor will go on to the next phase without changing its *tid* if the processor receives messages of types *SawSmaller* and *SeenBySmaller*.

To tolerate link failures, all processor in a trying segments enter the next phase if at least one of them completes its square. They enter the next phase only once, even if more than one of them completes its square. This task is accomplished as follows. Upon completing its square, a processor in a trying segment enters the next phase and sends messages of type *MoveOn* carrying its new phase to all processors in the trying segment. When a processor in the trying segment receives a *MoveOn* message, the processor compares its own phase with delivered one. If the delivered phase is greater than its own phase, the processor enters a new phase and relays the message to rest of processors in the trying segment. Otherwise, the message is discarded.

The algorithm continues in this way until d becomes greater than or equal to $\sqrt{n}$ for some processors; wrap-around occurs at this point. Note that no wrap-around occurs during any execution of procedure Compete, since the possibility of wrap-around is checked earlier.

6.3.2.4 Description of Procedure PostWrapAround

The main task of procedure PostWrapAround is to decrease the number of active trying segments to one. PostWrapAround operates similarly to Compete, but it executes only some constant q phases. (The value of q is given in the following section.) The main difference from Compete is the shape of paths that processors in a trying segment mark off. While a square is marked off in Compete, a cross, which consists of one vertical ring and one horizontal ring, is marked off in PostWrapAround. (Refer to Figure 18.) All other mechanisms remain same (including the mechanism for resolving competition between active trying segments).

A cross is marked as follows. A processor in a trying segment sends a message to its right to distance $Dist = \sqrt{n} + (t+1) + TryNum$ to mark off a horizontal ring. (Since a wrap-around condition is detected in procedure Compete, the size of the square mesh is now available.) If a processor receives a message with $Dist > 1$, it relays the message in the same direction with distance $Dist - 1$. Eventually, a processor receives a message with $Dist = 1$. Note that this processor (called the wrap-around processor) is not the processor that initiated the message. (This is necessary to have all crosses be link disjoint.) If a processor receives a message with $Dist = 1$, the processor send a message to its down link with $Dist = \sqrt{n}$. The message eventually returns to its sender and a cross is marked if it does not see other boundaries with greater *tid*.

Note that no two wrap-around processors are in same horizontal ring or in same vertical ring. Thus, all crosses are link disjoint. This ensures that at least one cross for a segment can be completed.

Upon entering PostWrapAround, processors in an active trying segment send $t+1$

messages to their right to mark off crosses. If one of the crosses is completed, the wrap-around processor sends a message back to a processor in the trying segment. Upon receiving such a message, all processors in a trying segment enter the next phase. (Note this is done exactly the same way as in Compete.) Since there are only a constant number of active segments that start PostWrapAround, only a constant number of phases are necessary to reduce the number of active trying segments to one. Eventually one active trying segment enters the last phase, and processors in the trying segment declare their *tid* as the leader's *id*.

6.3.3 Correctness of Algorithm $M1$

This section proves the correctness of algorithm $M1$. First, the existence of a trying segment is shown in the following lemma.

Lemma 6.3.1 *There is at least one trying segment at the end of an execution of BuildSeg of algorithm* $M1$.

Proof. Since there are at most t faulty links and $t < \sqrt{n}$, there is at least one vertical ring without faulty links. Thus, at least one execution of BuildSeg proceeds to phase $\lceil \log_\phi t^2 \rceil$ or terminates with an elected leader. If a leader is elected in a ring, there should be at least t passive processors above the leader. The number of passive processors between any two active processors is at least t. Therefore, at least one trying segment is built at the end of the first stage (BuildSeg) of algorithm $M1$. □

The following lemma shows that at least one message originated by a trying segment in stages 2 and 3 does not see any faulty links.

Lemma 6.3.2 *There is at least one message (among messages that are originated by processors in a trying segment in each phase of stages 2 and 3) that does not encounter faulty links during an execution of algorithm $M1$.*

Proof. It is clear that all paths that messages from a trying segment follow are link-disjoint on a phase. There are $t+1$ processors in a trying segment while there are at most t link failures in a whole square mesh. Thus, at least one message does not encounter faulty links. □

As shown above, no trying segment is prevented from entering the next phase by faulty links. The following lemma shows that there is at least one trying segment that enters PostWrapAround.

Lemma 6.3.3 *Assume that there is at least one trying segment that enters phase u, during an execution of Compete of algorithm $M1$. Then, there is at least one segment that enters phase $u+1$.*

Proof. If no processor in an active trying segment sees a boundary of a processor that belongs to another active trying segment, the active trying segment enters the next phase. (A processor belongs to a trying segment's boundary if the variable *SavedId* is *tid* of the trying segment.)

Assume that there is more than one active trying segment that enters phase u of Compete. Lemma 6.3.2 implies that, for each such trying segment, at least one processor does not have faulty links on the boundary of its square. Assume that these processors see another active processor's boundary. Let $p_i, p_{i+1}, \cdots, p_{i+l}$ be processors from different active trying segments that see another processor's boundary. Then, it can be assumed that $tid(p_i) < tid(p_{i+1}) < \cdots < tid(p_{i+l})$ since all id's are different.

Let processor p_i see processor p_j $(i < j \leq i + l)$. Then p_j will enter the next phase, unless it saw p_k $(j < k \leq k + l)$. If no p_j $(i \leq j < i + l)$ enters the next phase, p_{i+l} should have seen by at least one p_j $(i \leq j < i + l)$ and it saw at least one p_k $(i \leq k < i + l)$. Thus, processor p_{i+l} enters the next phase. The lemma follows. □

The above lemma implies that there is at least one trying segment that starts the third stage (PostWrapAround).

Lemma 6.3.4 *Only a constant number of trying segments enter PostWrapAround during an execution of algorithm M1.*

Proof. Let A_i be the maximum number of active trying segments of phase i. Assume that the first wrap-around occurs in phase v. Then, $A_{i+1} \leq n/d_i^2 + (A_i - n/d_i^2)/2$, where d_i is the side distance of the smallest square that is marked off in phase i. The first term is the maximum number of active trying segments that can enter phase $i+1$ because they saw no other processors. The second term is the maximum number of active trying segments that can enter phase $i + 1$ because they have completed a square. These are at most half of active segments that see some other segments, since at least one other segments should become passive if boundaries of two segment across. Hence,

$$\begin{aligned}
A_{i+1} &\leq \frac{n}{d_i^2} + \frac{1}{2}\left(A_i - \frac{n}{d_i^2}\right) \\
&= \frac{1}{2}A_i + \frac{1}{2}\frac{n}{d_i^2} \\
&= \frac{1}{2}A_i + \frac{1}{2}\frac{n}{(t+1)^2 c^{2(i-1)}}, \text{ since } d_i = (t+1)c^{i-1} \\
&= \frac{1}{2^{i+1}}A_1 + \frac{n}{(t+1)^2}\left(\frac{1}{c^{2i}}\right)\left(1 - \left(\frac{c^2}{2}\right)^i\right)\left(\frac{c^2}{2-c^2}\right) \\
&\leq \frac{1}{2^{i+1}}A_1 + \frac{n}{(t+1)^2}\left(\frac{1}{c^{2i}}\right)\left(\frac{c^2}{2-c^2}\right), \text{ if } c^2 < 2
\end{aligned}$$

$$
\begin{aligned}
&\leq \frac{1}{2^{i+1}}\left(\frac{n}{(t+1)^2}\right) + \left(\frac{n}{(t+1)^2}\right)\left(\frac{1}{c^{2i}}\right)\left(\frac{c^2}{2-c^2}\right), \text{ since } A_1 \leq \frac{n}{(t+1)^2} \\
&= \frac{n}{(t+1)^2}\left(\frac{1}{2^{i+1}} + \frac{1}{c^{2i}}\left(\frac{c^2}{2-c^2}\right)\right) \\
&\leq \frac{n}{(t+1)^2}\left(\frac{1}{c^{2i}}\right)\left(1 + \frac{c^2}{2-c^2}\right), \text{ since } c^2 < 2 \\
&\leq \frac{n}{(t+1)^2}\left(\frac{1}{c^{2i}}\right)\left(\frac{2+c^2}{2(2-c^2)}\right).
\end{aligned}
$$

Since v is the first wrap around phase, $d_v = (t+1)c^{v-1} \geq \sqrt{n}$. Thus, $A_v = \frac{2+c^2}{2(2-c^2)}$ for some constant $1 < c < \sqrt{2}$. The lemma follows. □

It has been shown that at least one and at most some constant number of trying segments enter PostWrapAround.

Lemma 6.3.5 *There is exactly one trying segment at the end of the execution of PostWrapAround of algorithm M1.*

Proof. A proof similar to the one for the Lemma 6.3.3 can be used to prove that there is at least one trying segment at the end of PostWrapAround. (Note that the PostWrapAround operates in the same way as Compete. The only difference is the path that a message follows in PostWrapAround is a cross while it is a square in Compete.)

Assume that there is more than one active trying segment that finishes the last phase p of PostWrapAround. Since all active trying segments during an execution of PostWrapAround mark off crosses that wrap-around, only one processor from one trying segment can see no other processor's boundary (if there is more than one, those should cross each other). Only half of remaining can enter the next phase since *tid* of one segment should be less than that of the other, so $A_{i+1} \leq \lceil A_i/2 \rceil$. Thus, the number of active trying segment decreases towards one. Let q be the number of

phases that procedure PostWrapAround executes. Then, by letting $q > \left\lceil \log \frac{2+c^2}{2(2-c^2)} \right\rceil$, only one processor can be elected. □

This section concludes with the following correctness theorem that immediately follows the lemma above.

Theorem 6.3.1 *Let N be an asynchronous bidirectional square mesh with at most $t < \sqrt{n}$ fail-stop link failures that occur before the start of an algorithm. Then algorithm $M1$ correctly solves the election problem on N.*

6.3.4 Message Complexity of Algorithm $M1$

This section gives an analysis of worst-case message complexity of algorithm $M1$. Analysis of each procedure in algorithm $M1$ is given in order.

Lemma 6.3.6 *The number of messages sent during an execution of procedure BuildSeg of algorithm M1 is $O(n \log t)$.*

Proof. There are $\sqrt{n}$ executions of procedure BuildSeg each of which needs $O(\sqrt{n} \log t)$ messages each in the worst case. Thus, the number of message sent is $O(n \log t)$. □

Lemma 6.3.7 *The number of messages sent during an execution of Compete of algorithm $M1$ is $O(n)$.*

Proof. The number of trying segments A_i active in phase i is bounded by

$$\left(\frac{n}{(t+1)^2(c^{2(i-1)})}\right)\left(\frac{2+c^2}{2(2-c^2)}\right)$$

by Lemma 6.3.4.

A square with side distance d causes $8d$ messages. Of these, $4d$ messages are needed to mark off its boundary with "Looking" or "SawSmaller" messages and another $4d$ message are needed for "SeenbySmaller" messages. Since there are $t+1$ processors for a trying segment, a trying segment causes $8d(t+1)+2(t+1)^2$ messages. The $2(t+1)^2$ messages are needed when processors that mark off larger squares make turns.

Let d_i be the smallest square marked off during phase i. Then, $d_i = (t+1)c^{i-1}$. Let m_i be the number of message sent on phase i. Then,

$$\begin{aligned} m_i &\leq A_i(8d_i(t+1)+2(t+1)^2) \\ &= \frac{n}{(t+1)^2(c^{2(i-1)})}\left(\frac{2+c^2}{2(2-c^2)}\right)(t+1)c^{i-1} \\ &= n\left(\frac{8}{c^{i-1}}+\frac{2}{c^{2(i-1)}}\right). \end{aligned}$$

Let p be the phase when a wrap-around first occurs. Then the total number of messages m_t sent during Compete is

$$\begin{aligned} m_t &= \sum_{i=1}^{p} m_i \\ &\leq \sum_{i=1}^{p} 8n\left(\frac{1}{c^{i-1}}+\frac{1}{c^{2(i-1)}}\right) \\ &= 8n\left[\frac{c}{c-1}\left(1-\frac{1}{c^p}\right)+\frac{c^2}{c^2-1}\left(1-\frac{1}{c^{2p}}\right)\right]. \end{aligned}$$

Since c is some constant greater than 1, n_t is $O(n)$. The lemma follows. □

Lemma 6.3.8 *The number of messages sent during an execution of PostWrapAround of algorithm M1 is $O(n)$.*

Proof. A trying segment that is marking off crosses requires $8\sqrt{n} + 2((t+1) + TryNum)$ messages. For the first term, half of these are needed to mark its boundary with "Looking" or "SawSmaller" messages and the other half are needed for

"SeenbySmaller" messages. The second term is due to the distance between the wrap around processors and the corresponding processor in the trying segment. There are $t+1$ processors, each of which tries to mark off a cross. Thus, the maximum number of messages needed for a trying segment becomes $O(n)$ by recalling that $t < \sqrt{n}$.

There are a constant number of trying segments active in each phase. Also, there are a constant number of phases in an execution of procedure PostWrapAround. Thus, the total number of messages required for procedure PostWrapAround is $O(n)$.

For the declaration (notifying the *id* of the elected leader to all processors that are connected to the leader), at most two messages are needed for a link since every processor relays the informed leader's *id* to all links except the one that the *id* is received from. Thus, the declaration also needs $O(n)$ messages. The lemma follows. □

The following theorem immediately follows.

Theorem 6.3.2 *The number of messages sent during an execution of algorithm M1 is* $O(n \log t)$.

An analysis of the number of bit required for the longest message follows. Since there are only a constant number of message types, only a constant number of bits are necessary to distinguish different types of messages. But distance information (that a message can travel) requires $O(\log n)$ bits. Thus, the size of longest message is $O(\log n + b)$, where b is the length of largest identifier.

6.4 An Algorithm for the Case of $t < 2\sqrt{n}$

This section considers the case in which t is less than $2\sqrt{n}$. Obviously, algorithm $M1$ does not work for this case, since algorithm $M1$ requires $t < \sqrt{n}$.

Since there are $\sqrt{n}$ horizontal rings and $\sqrt{n}$ vertical rings, at least one ring (vertical or horizontal) does not contains faulty links if $t < 2\sqrt{n}$. Note that the size of square meshes can be obtained by executing an election algorithm (such as algorithm DG, which does not work correctly if there are faulty link in a ring) on every vertical and horizontal ring, since at least one of them will successfully elect a leader.

Algorithm $M2$ elects a leader on such a square mesh based on the above fact. In the following an outline of algorithm $M2$ is described. (Since all procedures are based on algorithm described in previous chapters, a detailed description is omitted.)

Algorithm $M2$ operates in three stages. The main task of first stage is to find the size of a square mesh. This is done by executing election algorithm DG on every vertical and horizontal ring, independently and concurrently. Since there is at least one ring without faulty links, at least one execution correctly terminates. The elected leader of the ring sends a message to calculate the size of the ring (thus the size of the square mesh) and inform all connected processors the size of the ring. Upon learning the size of the ring, all connected processors in the ring relay the size to all processors in the square mesh. Note that there are some processors that are connected to the ring, since there are $2\sqrt{n}$ links that from a ring.

Algorithm $M2$ /* Code for processor p_{ij} */
Stage 1:
Execute algorithm DG on horizontal ring h_i and vertical ring v_j;
if (elected leader) then
    Inform all processors in the mesh of the size of the square mesh;
Stage 2:
Execute procedure HElection on horizontal ring h_i;
Stage 3:
Execute procedure VElection on vertical ring v_j;

Figure 22: Algorithm M2

6.4.1 Description of procedure HElection

Upon receiving a message that carries the size of the square mesh, a processor starts the second stage (procedure HElection) on its horizontal ring. Procedure HElection is based on algorithm $R1$ described in Chapter 5. Algorithm $R1$ elects a leader on an asynchronous bidirectional ring with at most one fail-stop link failure that occurs before the start of an algorithm. Since the size of ring is obtained in stage 1, HElection successfully elects a leader if there is only one link failure on the horizontal ring.

The only modification to the algorithm is that messages informing processors of the leader's *id* mark links as follows. A message marks the link that it just traversed as "found non-faulty" if the link is not already marked as "assumed faulty". Note that a link is marked by two processors that is connected by the link. Thus, it is possible that a message crosses the link to find that the link is marked as "assumed faulty" by the processor at the other end. In this case, the message stops its travel without changing the mark. The necessity of this will become clear. Upon receiving a message containing the leader's *id*, a processor enters the third stage (procedure

VElection).

Since there are $\sqrt{n}$ horizontal rings, $\sqrt{n}$ copies of HElection are executed. At least one copy of execution terminates on a horizontal ring, since there is at least one horizontal ring containing at most one faulty link. Horizontal rings on which the execution of HElection is terminated are called *candidate rings*. The leader's *id* of a candidate ring is called the *id* of the candidate ring.

6.4.2 Description of Procedure VElection

The goal of the third stage is to elect one of the candidate rings. This is done by executing algorithm $R1$ on every vertical ring. Since at least one vertical ring contains at most one faulty link, at least one execution terminates with the elected leader's *id* that is one of candidate ring's *id*. It must be ensured that all executions are performed on the same set of candidate rings, since there are $\sqrt{n}$ such executions. (Note that there could be some horizontal rings on which the execution of HElection never terminates. Also, not all processors in a horizontal ring are informed of the leader's *id* at the same time.)

The following ensures that that the same set of candidate rings is used for all executions of VElection as follows. Upon starting procedure VElection, processor p_{ij} sends two messages in both directions on horizontal ring h_i. These messages follow a link if the link is marked as "found non-faulty". It returns to its sender if a link is marked as "found non-faulty" after marking that link as "assumed faulty". When the message returns to its sender, it keeps track of the number of links marked as "found non-faulty". (If there are no links that are marked as "found non-faulty", the message eventually returns to p_{ij} after passing through all processors in the horizontal ring.)

Processor p_{ij} sets its state to *active* if at least $\sqrt{n}-1$ links are marked as "found non-faulty"; otherwise, it sets its state to *passive*. If there are at least $\sqrt{n}-1$ "found non-faulty" links, the election on h_i should be completed and the leader id should be available to all processors in the ring. An active processor waits for the leader's id of its horizontal ring if it is not available.

There are $\sqrt{n}$ executions of VElection. Thus, at least one of those executions terminates with a leader. Since all executions are performed on the same set of candidate rings, all terminated executions share the same leader's id. After a leader is elected, its id is sent to all processors that are connected to the leader in the square mesh.

6.4.3 Correctness of Algorithm $M2$

To show the correctness of algorithm M, it should be first proved that the same set of candidate rings is used for all executions of procedure VElection of algorithm $M2$.

Assume that there is a candidate ring h_i whose id is used for processor p_{iu}'s execution of VElection but not in processor p_{iv}'s execution. There should be at least one link in h_i that p_{iv} found marked as "assumed faulty" but p_{iv} did not.

Assume that p_{iu} checks the link before p_{iv}. When p_{iu} checks the link, it should be marked as "found non-faulty", otherwise p_{iu} mark is "assumed faulty". Since the link is marked as "found non-faulty", it can not be marked as "assumed faulty" later. Thus, p_{iu} should use h_i's id for election on its vertical ring. This is a contradiction.

Assume that p_{iv} checks the link before p_{iu}. After p_{iv} checks the link, it should be marked as "found non-faulty". Once a link is marked as "assumed faulty", it cannot be changed. Thus, p_{iu} cannot use h_i's id for election on its vertical ring. This is a

contradiction.

Since the algorithm $R1$ correctly elects the leader, the above lemma implies the correctness of VElection. By recalling the correctness of algorithm DG, the correctness of algorithm $M2$ follows immediately.

There are $2\sqrt{n}$ executions of algorithm DG that each requires $O(\sqrt{n}\log n)$ messages. $O(n)$ messages are needed to broadcast the size of square mesh to all connected processors. Thus, the first stage requires $O(n\log n)$ messages.

There are $\sqrt{n}$ executions of HElection that each requires $O(\sqrt{n}\log n)$ messages. Thus, the second stage also needs $O(n\log n)$ messages

At most $2\sqrt{n}$ message are required to determine the state of each processor. Since there are $\sqrt{n}$ processors in each vertical ring, this requires $O(n)$ messages. Thus, VElection needs $O(\sqrt{n}\log n)$ messages. Since there are $\sqrt{n}$ execution of VElection, the worst-case message complexity of the third stage is $O(n\log n)$.

Therefore, worst-case message complexity of algorithm $M2$ is $O(n\log n)$ The following theorem summarizes the results of this section.

Theorem 6.4.1 *Let $\mathcal{N}$ be a square mesh of n processors with at most $t < 2\sqrt{n}$ fail-stop link failures that occur before an execution of an algorithm. Then, algorithm M correctly solves election problem with worst-case message complexity $O(n\log n)$.*

6.5 An Impossibility Result

The following theorem shows a case in which election is impossible on square meshes.

Theorem 6.5.1 *Let N be an asynchronous square mesh with $t \geq 2\sqrt{n}$ fail-stop link failures. Assume that every processor know its own identifier, and t and its rlation*

to n. Then there is no distributed algorithm for electing a leader on N.

Proof. Assume the contrary, that there is an algorithm $\mathcal{A}$ that elects a leader in such networks of size n. Consider executions of algorithm $\mathcal{A}$ on four different square meshes (say M_1, M_2, M_3, and M_4) of size n such that no two *id*'s of all four square meshes are the same. Then, algorithm $\mathcal{A}$ should elect a leader correctly on each of four square meshes if $2\sqrt{n}$ are faulty links (per mesh).

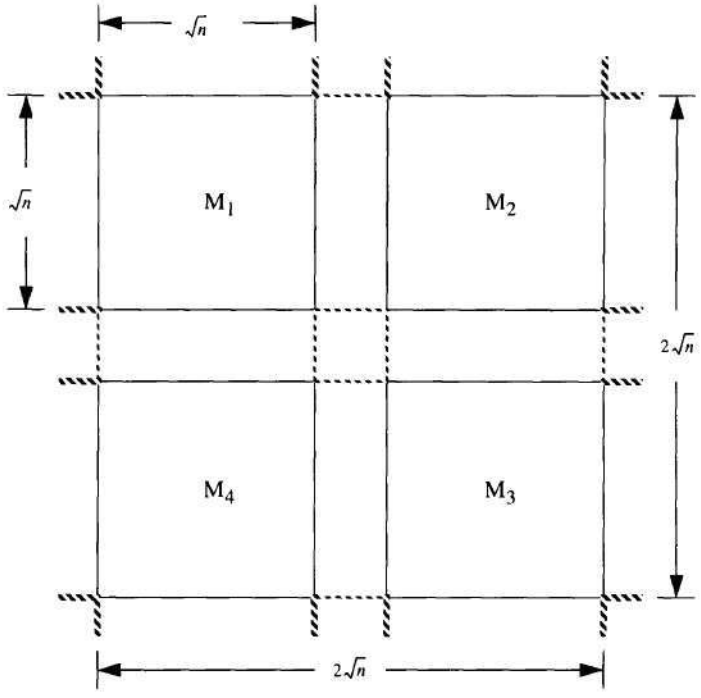

Figure 23: An Impossible Case

Now, consider a square mesh of size $4n$ in which all processors in the four square meshes of size n preserve the relative positions of the processors in each mesh of size n. (See Figure 23.) In the figure, the four squares are drawn with solid lines and dotted lines are links that connect them to make a square mesh of size $4n$. Assume that bottom $2\sqrt{n}$ and left vertical $2\sqrt{n}$ thick dotted links are faulty links. Also, assume

that thin dotted links that connect square meshes of size n are very slow links. Note that each square meshes of size n has $2\sqrt{n}$ are faulty links.

Consider an execution of algorithm $\mathcal{A}$ on the square mesh of size $4n$. Since faulty links and slow links are not distinguishable, processors in a square mesh of size n may act exactly the same as in the original execution. Thus, there could be four leaders elected. This is a contradiction and the theorem follows. □

6.6 Concluding Remarks

This chapter considered the election problem on asynchronous bidirectional square mesh networks with fail-stop link failures. Two algorithms and an impossibility result were obtained.

For the case $t < \sqrt{n}$ (t is maximum number of faulty link allowed), an algorithm with worst-case message complexity of $O(n \log t)$ is presented. An algorithm with worst-case message complexity of $O(n \log n)$ is obtained when $t < 2\sqrt{n}$. It is shown that the election is impossible if $t \geq 2\sqrt{n}$.

The lower bound of the election problem on asynchronous square meshes with $t < \sqrt{n}$ appears to be difficult but an interesting open problem. The existence of algorithms with better worst-case message complexity for cases $\sqrt{n} \leq t < 2\sqrt{n}$ is also an interesting open problem. It is conjectured that there is an algorithm with worst-case message complexity $O(n \log t)$ for square meshes with at most $t < \sqrt{n}$ intermittent (as opposed to fail-stop) link failures.

Chapter 7

Conclusions

This dissertation examined some issues concerning fault tolerance in distributed computing systems were examined. The first problem investigated was average-case behavior of algorithms for election on asynchronous rings of processors. An algorithm with good worst-case and good average-case message complexity was obtained. It was demonstrated by extensive simulations that average-case message complexity of the algorithm appears very close to the theoretical optimum. Theoretical analysis of average-case behavior of the algorithm is an interesting open problem. Also, the existence of similar algorithms on square meshes should be interesting since a square mesh can be viewed as a ring of rings.

The impact of inexact knowledge of processors was examined. Specifically, the election problem on asynchronous rings was considered with one possible link failure when a lower bound and/or an upper bound on ring size is known to all processors. It was shown that a good lower bound is most useful in designing algorithms with better worst-case message complexity. The availability of upper bound is useful only if the upper bound and the lower bound are sufficiently close. Even a very tight upper bound is not helpful if not combined with a good lower bound.

The impact of the additional knowledge of the identifiers of two neighbors was also examined. There are cases where the election problem is not solvable without this

knowledge. But this additional knowledge is not helpful in improving the worst-case message complexity if the problem is solvable without the knowledge. Investigating the impact of inexact knowledge of size on different topologies is an interesting open problem.

Tolerating link failures on square meshes of processors was also studied. While conceptually simpler algorithms were obtained using election algorithms on rings, a more sophisticated algorithm with better worst-case message complexity is also obtained for the case with smaller number of faulty links. The lower bound of the election problem in square mesh with link failures is still not solved. The existence of algorithms with better worst-case message complexity than the algorithm presented for the case $t \geq \sqrt{n}$ is also an interesting open problem.

Bibliography

[1] H. H. Abu-Amara. Fault-tolerant distributed algorithm ofr election in complete networks. *IEEE Trans. Comput.*, 37:449–453, April 1988.

[2] H. H. Abu-Amara. *Fault-Tolerant Distributed Algorithms for Agreement and Election.* PhD thesis, University of Illinois at Urbana-champaign, 1988.

[3] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete network. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, pages 186–195. ACM, 1985.

[4] Y. Afek and M. Saks. Detecting global termination conditions in the face of uncertainty. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 109–124, 1987.

[5] H. L. Bodlaender. A better lower bound for distributed leader finding in bidirectional asynchronous rings of processors. *Information Processing Letters*, 27:287–290, 1988.

[6] H.L. Bodlaender and J. van Leeuwen. New upperbounds for decentralized extrema-finding in a ring of processors. In *3rd Annual Symposium on Theoretical Aspects of Computer Science*, pages 119–129, 1986.

[7] J. E. Burns. A formal model for message passing systems. Technical Report Tech. Rep. 91, Computer Science Dept.,Indiana Univ.,Bloomington, May 1980.

[8] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extra-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, May 1979.

[9] B. A. Coan. A communication-efficient canonical form for fault-tolerant distributed protocols. In *Proc. 5th ACM Symp. PODC*, 1986.

[10] D. Dolev, M. J. Fischer, R. Fowler, N. A. Lynch, and H. R. Strong. An efficient algorithm for byzantine agreement without authentication. *Inf. Control*, 52:257–274, March 1982.

[11] D. Dolev and H. R. Strong. Athenticative algorithm for byzantine agreement. *SIAM J. Comput.*, 12:656–666, Nov. 1983.

[12] Danny Dolev, Maria Klawe, and Michael Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3:245–260, 1982.

[13] Paul Everhardt. Average case behavior of distributed extrema-finding algorithms. Technical Report ACT-49, Univ. Illinois Urbana-Champaign, 1984.

[14] M. J. Fisher, N.A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32:374–382, April 1985.

[15] G. N. Frederickson and N. A. Lynch. The impact of synchronous communication on the problem of electing a leader in a ring. In *Proceedings of the Sixteenth Annual Symposium on Theory of Computing*, pages 493–503. ACM, 1984.

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

[17] R. G. Gallager, P. A. Humbelt, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *Journal of the ACM*, 5(1):66–77, January 1983.

[18] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, c-31(1):48–59, January 1982.

[19] O. Goldreich and L. Shrira. The effect of link failures on computations in asynchronous rings. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 174–185. ACM, 1986.

[20] O. Goldreich and L. Shrira. Electing a leader in a ring with link failures. *Acta Informatica*, 24:79–91, 1987.

[21] Oded Goldreich and Liuba Shrira. On the complexity of computation in the presence of link failures: the case of a ring. *Distributed Computing*, pages 121–131, 1991.

[22] Lisa Higham. A simple efficient algorithm for maximum finding on rings. Research Report 92/494/32, The University of Calgary, 2500 University Dr. N.W., Calgary, Alberta, Canada T2N 1N4, 1992.

[23] Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. In *22st Annual Symposium on Foundations of Computer Science*, pages 150–158. IEEE, 1981.

[24] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. In *Proceedings of the Fourth Annual*

ACM Symposium on Principles of Distributed Computing, pages 163–174. ACM, 1985.

[25] E. Korach, S. Moran, and S. Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proc. 4th ACM Symp. PODC*, 1984.

[26] Christian Lavalut. Average number of message for distributed leader-finding in ring of processors. *Information Processing Letters*, 30:167–176, February 1989.

[27] G. LeLann. Distributed systems - towards a formal approach. In *Information Processing 77*, pages 155 – 160. Elsevier Science, 1977.

[28] M.C. Loui, T.A. Matsushita, and D.B. West. Election in complete networks with a sense of direction. *Information Processing Letters*, 22:185–187, April 1986.

[29] T. Masuzawa, N. Nishikawa, K. Haihara, and N. Tokura. Optimal fault-tolerant distributed algorithms for election in complete networks with a global sense of direction. In J.C. Bermond and M. Raynal, editors, *Distributed Algorithms. 3rd International Workshop*, pages 171–182. Springer-Verlag, 1989.

[30] F. Mattern. Message complexity of simple ring-based election algorithms - an empirical analysis. Technical Report SFB124-36/88, University of Kaiserslautern, Dept. of Computer Science, P.O.Box 3049, D 6750 Kaiserslautern, West-Germany, October 1988.

[31] Friedmann Mattern. Message complexity of simple ring-based election algorithms - an empirical analysis. In *9th Int. Conf. Dist. Computing Systems*, pages 94–100. IEEE, 1989.

[32] S. Moran, M. Shalom, and S. Zaks. An algorithm for distributed leader finding in bidirectional rings without common sense of direction. Technical report, Technion, Haifa, 1985.

[33] J. Pachl, E. Korach, and D. Rotem. Lower bounds for distributed maximum-finding algorithms. *J. ACM*, 31(4):905–918, Oct. 1984.

[34] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27:228–234, April 1980.

[35] G. L. Peterson. Efficient algorithms for elections in meshes and complete networks. Technical Report TR-140, University of Rochester, Detp. of Computer Science, July 1985.

[36] Gary L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, Oct. 1982.

[37] James L. Peterson and Abraham Siberschatz. *Operating System Concepts.* Addison Wesley, second edition, 1985.

[38] D. Rotem, E. Korach, and N. Santoro. Analysis of distributed algorithm for extrema finding in a ring. *J. Parallel and Distributed Computing*, 4:575–591, 1987.

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

[40] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[41] L. Shrira and O. Goldreich. Electing a leader in the presence of faults: a ring as a special case. Technical report #354, Technion, February 1985.

[42] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume 2. MIT Press, 1990.

Vita

Byungho Yi received his B.S. and M.S. degrees from Seoul National University, Seoul Korea in 1980 and 1984, respectively. He completed his Ph.D. work at Georgia Institute of Technology in 1994. His thesis research involves issues in fault tolerance for distributed computing systems. His other research interests include the design of distributed operating systems and parallel computation.