FAULT-TOLERANT DISTRIBUTED ALGORITHMS FOR AGREEMENT AND ELECTION

Hosame Hassan Abu-Amara

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.
This report consists of three parts. In the first part, we characterize completely the shared-memory requirements for achieving agreement in an asynchronous system of fail-stop processes that die undetectably. There is no agreement protocol that uses only read and write operations, even if at most one process dies. This result implies the impossibility of Byzantine agreement in asynchronous message-passing systems. Furthermore, there is no agreement protocol that uses test-and-set operations if memory cells have only two values and two or more processes may die. In contrast, there is an agreement protocol with test-and-set operations if either memory cells have at least three values or at most one process dies.

In the second part, we consider the election problem on asynchronous complete networks when the processors are reliable but some of the channels may be intermittently faulty. To be consistent with the standard model of distributed algorithms in which channel delays can be arbitrary but finite,
we assume that channel failures are undetectable. We give an algorithm that correctly solves the problem when the channels fail before or during the execution of the algorithm. Let $n$ be the number of processors in the network, $f$ be the maximum number of faulty channels, and $r$ be a design parameter. The algorithm uses no more than $O\left( nr + \frac{n}{(r-1)\log\left(\frac{n}{f}\right)} \right)$ messages in the worst case, runs in time $O\left( \frac{n}{(r-1)f} \right)$, and uses at most $O(\log |T|)$ bits per message, where $|T|$ is the cardinality of the set of processor identifiers. If $r$ is chosen to minimize the number of messages, our algorithm uses no more than $O(nf + n \log n)$ messages.

In the third part, we present the most efficient algorithm that we know of for election in synchronous square meshes. The algorithm uses $\frac{n^2}{229}$ messages, runs in time $\Theta\left( \sqrt{n} \right)$ time units, and requires $O(\log |T|)$ bits per message. Also, we prove that any comparison algorithm on meshes requires at least $\frac{n^2}{32}$ messages.
FAULT-TOLERANT DISTRIBUTED ALGORITHMS
FOR AGREEMENT AND ELECTION

BY

HOSAME HASSAN ABU-AMARA

B.S., University of California, Berkeley, 1983
M.S., University of Illinois, 1985

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1988

Urbana, Illinois
This thesis consists of three parts. In the first part, we characterize completely the shared-memory requirements for achieving agreement in an asynchronous system of fail-stop processes that die undetectably. There is no agreement protocol that uses only read and write operations, even if at most one process dies. This result implies the impossibility of Byzantine agreement in asynchronous message-passing systems. Furthermore, there is no agreement protocol that uses test-and-set operations if memory cells have only two values and two or more processes may die. In contrast, there is an agreement protocol with test-and-set operations if either memory cells have at least three values or at most one process dies.

In the second part, we consider the election problem on asynchronous complete networks when the processors are reliable but some of the channels may be intermittently faulty. To be consistent with the standard model of distributed algorithms in which channel delays can be arbitrary but finite, we assume that channel failures are undetectable. We give an algorithm that correctly solves the problem when the channels fail before or during the execution of the algorithm. Let \( n \) be the number of processors in the network, \( f \) be the maximum number of faulty channels, and \( r \) be a design parameter. The algorithm uses no more than \( O\left(\frac{nrf + nr}{(r-1)} \log \left(\frac{n}{(r-1)f}\right)\right) \) messages in the worst case, runs in time \( O\left(\frac{n}{(r-1)f}\right) \), and uses at most \( O(\log |T|) \) bits per message, where \(|T|\) is the cardinality of the set of processor identifiers. If \( r \) is chosen to minimize the number of messages, our algorithm uses no more than \( O(nf + n \log n) \) messages.

In the third part, we present the most efficient algorithm that we know of for election in synchronous square meshes. The algorithm uses \( \frac{229}{18} n \) messages, runs in time \( \Theta(\sqrt{n}) \) time units, and
requires $O(\log |T|)$ bits per message. Also, we prove that any comparison algorithm on meshes requires at least $\frac{57}{32} n$ messages.
ACKNOWLEDGMENTS

My sincere thanks and gratitude go to my advisor, Professor Michael C. Loui. His advice and teaching will guide me throughout my career. He is an example of an excellent researcher, a patient teacher, and a caring advisor. I hope that I enriched his life as much as he enriched mine. I also wish to thank the members of my committee: Professors Donna Brown, Kent Fuchs, Dave Liu, and Vijaya Ramachandran for their comments on my thesis.

Several people honored me with their comments about Chapter 2. James Burns noticed that two two-valued cells suffice to achieve agreement between two processes with test-and-set operations. Larry Stockmeyer clarified the relationship of Chapter 2 to Fischer, Lynch, and Paterson's paper [18]. A referee encouraged Professor Michael C. Loui and me to consider the impossibility of $k$-resilient agreement and conjectured the results of Section 2.5.

My thanks also go to my family (Hassan, Suad, Lina, and Marwan), colleagues (Marsha Prastin, Jerry Trahan, and Mike Wu), and friends for their emotional and financial support. Without their confidence in me, I would not have finished this thesis.

Finally, I gratefully acknowledge the financial support of the Office of Naval Research under contract N00014-85-K-0570.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION AND LITERATURE SURVEY</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Literature Survey</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Process agreement</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Election</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Contents of the Thesis</td>
<td>6</td>
</tr>
<tr>
<td>2. MEMORY REQUIREMENTS FOR AGREEMENT AMONG UNRELIABLE ASYNCHRONOUS PROCESSES</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Definitions</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Process systems</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Agreement protocols</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3 Process death</td>
<td>12</td>
</tr>
<tr>
<td>2.2.4 Resilience</td>
<td>12</td>
</tr>
<tr>
<td>2.2.5 Computation graphs</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Three Agreement Protocols</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Impossibility of Fully Resilient Agreement</td>
<td>16</td>
</tr>
<tr>
<td>2.4.1 Properties of fully resilient protocols</td>
<td>16</td>
</tr>
<tr>
<td>2.4.2 Read/write protocols</td>
<td>18</td>
</tr>
<tr>
<td>2.4.3 Test-and-set protocols</td>
<td>19</td>
</tr>
</tbody>
</table>
2.5 Impossibility of $k$-Resilient Agreement .................................................. 20
  2.5.1 Properties of $k$-resilient protocols .................................................... 20
  2.5.2 1-Resilient read/write protocols ....................................................... 22
  2.5.3 2-Resilient test-and-set protocols .................................................... 24
2.6 Summary .......................................................................................................... 27

3. FAULT-TOLERANT DISTRIBUTED ALGORITHM FOR ELECTION IN COMPLETE NETWORKS .......................................................... 29
  3.1 Introduction .................................................................................................... 29
  3.2 Model ............................................................................................................ 30
  3.3 Informal Description of the Algorithm ....................................................... 31
    3.3.1 Definitions .............................................................................................. 32
    3.3.2 The algorithm ......................................................................................... 33
  3.4 Proof of Correctness ..................................................................................... 35
  3.5 Message Complexity ..................................................................................... 41
  3.6 Time Complexity .......................................................................................... 43
  3.7 Formal Description of the Algorithm ........................................................... 43

4. UPPER AND LOWER BOUNDS FOR ELECTION IN SYNCHRONOUS SQUARE MESHES ......................................................... 48
  4.1 Introduction .................................................................................................... 48
  4.2 Model ............................................................................................................ 48
  4.3 Upper Bounds .............................................................................................. 50
    4.3.1 Overview of the algorithm ...................................................................... 51
    4.3.2 Details of the algorithm ......................................................................... 55
4.4 Proof of Correctness ................................................................. 59
4.5 Message Complexity ............................................................... 60
4.6 Time Complexity ...................................................................... 68
4.7 Lower Bounds .......................................................................... 69
  4.7.1 Assumptions ........................................................................ 69
  4.7.2 Executions ........................................................................... 70
  4.7.3 Chains ................................................................................ 70
  4.7.4 Replication symmetry .......................................................... 74
  4.7.5 Proof of the lower bound ....................................................... 76

5. FUTURE WORK ........................................................................... 82
  5.1 Problems Associated with Chapter 2 ........................................ 82
  5.2 Problems Associated with Chapter 3 ........................................ 82
  5.3 Problems Associated with Chapter 4 ........................................ 83
REFERENCES .................................................................................. 84
VITA .............................................................................................. 88
LIST OF FIGURES

Figure 2.1. The Transitions by $G$ and $H$ from $s$ ................................................................. 19
Figure 2.2. The Transitions by $G$ and $H$ from $u$ .................................................................... 23
Figure 2.3. The Transitions by $F, G$, and $H$ ........................................................................ 26
Figure 4.1. A 9-mesh .................................................................................................................... 49
Figure 4.2. The Definitions of $NW(u,q), NE(u,q), SE(u,q),$ and $SW(u,q)$ .......................... 52
Figure 4.3. Processor $w$ in $SW(u,a_1)$ .................................................................................... 53
Figure 4.4. Processor $r_0$ in $SE(r,a_1)$ .................................................................................... 61
Figure 4.5. The Definitions of $NW(u,q,q'), NE(u,q,q'), SE(u,q,q'),$ and $SW(u,q,q')$ ........ 64
Figure 4.6. Processors $w_0, u_0,$ and $v_0$ .................................................................................. 67
Figure 4.7. Processor $p$'s 3-diamond ........................................................................................... 71
Figure 4.8. A $k$-moddiamond ...................................................................................................... 75
Figure 4.9. Processor Identifiers in a $1$-moddiamond ................................................................. 77
Figure 4.10. Processor Identifiers in a 64-mesh .......................................................................... 78
CHAPTER 1
INTRODUCTION AND LITERATURE SURVEY

1.1 Introduction

Distributed systems are often used to access remote shared resources. These systems offer high availability and reliability in addition to speed. A component failure in a distributed system leads only to performance degradation, while a component failure in a single processor system is often catastrophic. For example, the files in a distributed system usually have several replicas at various sites. If a site crashes, then copies of the files are available at other sites. Furthermore, the nonfaulty sites can ignore the failed site until it is repaired. Thus, distributed systems are particularly suited to applications requiring availability in adverse, stressful environments.

Distributed systems are either synchronous or asynchronous. The clocks of the processors in a synchronous system are synchronized so that all processors execute each instruction within some fixed time interval. On the other hand, the clocks of the processors in an asynchronous system are not guaranteed to be synchronized. To solve a given problem, the processors in a distributed system must be able to communicate with one another. In shared-memory systems, the processors communicate through shared-memory cells. These cells may reside at a central processor that offers virtual shared-memory services. Shared-memory systems are analogous to PRAMs in parallel systems. In message-passing systems, the processors communicate by passing messages to one another. Such systems are analogous to fixed interconnection networks in parallel systems.

Link and node failures complicate the design of distributed algorithms. Failures may occur before or during the execution of the algorithms, and hence data and messages may be lost. There are several types of failure. For example, failures can be fail-stop, Byzantine, or intermittent. The fail-stop failure is the most benign of the failure types; a failed node stops sending messages, and a failed link stops transmitting messages. In the fail-stop failure, once a node or a link stops sending messages, it never sends another message. On the other hand, the Byzantine failure is one of the most malicious of the
failure types; nodes or links fail by altering messages, sending false information, and so on. The intermittent failure is more malicious than fail-stop failure but less malicious than Byzantine failure: nodes or links fail by losing messages at will. Schlichting and Schneider [46] show how to design a network in which processors fail only by stopping.

To deal with faults, researchers either design algorithms that try to detect faults, or assume that the faults are undetectable and design enough message-redundancy in the algorithms to tolerate the failures. Preparata, Metze, and Chien [44] formulate necessary and sufficient conditions for automatic fault detection in systems with multiple faults. Unfortunately, fault detection is not practical in asynchronous systems since it forces the nodes to distinguish between very slow links and faulty links. Furthermore, fault detection wastes valuable time in synchronous systems. Therefore, it is desirable to design fault-tolerant distributed algorithms that do not rely on fault detection.

In this thesis, we solve some important problems often encountered in the design of fault-tolerant distributed algorithms: Process agreement and election.

A fundamental problem of distributed computation is the construction of protocols to reach agreement among the processes in a distributed computer system. Agreement appears in the design of mutual exclusion algorithms for processes that compete for a shared resource. By communicating among themselves, the processes agree on which process gains access to the resource. Agreement also appears in transaction commitment protocols for distributed databases. A transaction may invoke several processes -- sometimes called the agents of the transaction -- to access data records. To commit the transaction, all processes must agree to write the new values of the records.

Election is the problem of choosing a unique processor as the leader of a network of processors. The processors are identical except that each processor has a unique identifier chosen from a totally ordered set. Initially, no processor knows the identifier of any other processor. Hence, the processors cannot elect a predetermined leader. Two adjacent processors communicate by sending messages to each other on the communication link connecting them. Two nonadjacent processors communicate through
processors adjacent to both of them. The election problem occurs, for instance, when a processor must be
selected to replace a malfunctioning central lock coordinator in a distributed database system, or to
replace a primary site in a replicated distributed file system [5], [23], [37]. Election also occurs in
token-passing systems. The processors in these systems pass a unique token among themselves. The
processor that receives the token executes the processor’s algorithm and then sends the token to some
other processor. If the token is lost, then the processors elect a leader to issue a new token.

1.2 Literature Survey

1.2.1 Process agreement

Protocols for agreement depend on the characteristics of the distributed system. The system may be
synchronous or asynchronous. The processes may communicate by passing messages or by accessing
cells in a shared memory. Process failures may be Byzantine, in which a faulty process may
communicate maliciously, or fail-stop, in which a process dies without communicating further.

Pease, Shostak, and Lamport [40] showed how to achieve agreement in synchronous message-
passing systems with Byzantine failures. In this situation the problem of reaching agreement is called the
Byzantine Generals Problem. Dolev et al. [15] and Dolev and Strong [17] devised more efficient
solutions for the Byzantine Generals Problem. The most efficient solution to the problem was presented
by Coan [13]. His paper also includes an extensive bibliography on the problem. Dolev, Dwork, Lynch,
and Stockmeyer [14], [18] gave conditions under which limited asynchronism can be tolerated in the
Byzantine Generals Problem.

Now consider asynchronous systems with fail-stop processes. For detectable process death, a
mutual exclusion algorithm of Burns [9] can be modified to achieve agreement with two-valued shared
memory cells, and the protocols of Schneider [47] achieve agreement on message-passing systems. For
undetectable process death, the decentralized simulation of resource managers of Jaffe [31] achieves
agreement with test-and-set operations on four-valued shared memory cells. (Actually, three-valued cells
suffice [30].) Indeed, Jaffe’s simulation uses $O(n^2)$ cells to achieve agreement an unlimited number of
times. On the other hand, Fischer, Lynch, Merrit, and Paterson [19], [20] proved that agreement cannot be achieved in asynchronous message-passing systems with the undetectable death of even one process; their result holds a fortiori for Byzantine failures.

All papers that we have cited treat only deterministic protocols. Ben-Or [8] designed a simple probabilistic agreement protocol for completely asynchronous message-passing systems with Byzantine failures.

1.2.2 Election

There are three common measures of the efficiency of a distributed algorithm: the maximum number of messages sent during any execution of the algorithm, the maximum running time of the algorithm, and the size of the messages required in the algorithm. Election algorithms should minimize all three measures.

Several synchronous and asynchronous election algorithms have been proposed on a variety of network topologies. Gafni [23], and Gallager, Humblet, and Spira [24] prove that election algorithms that work for all networks require $\Theta(m+n\log n)$ messages, where $m$ is the number of links in the network. Now consider election algorithms that work for specific networks. Burns [9], Frederickson and Lynch [22], and Pachl, Korach, and Rotem [39] show that election in synchronous rings requires $\Omega(n\log n)$ messages. Since synchronous rings are a special case of asynchronous rings, election algorithms for asynchronous rings also require $\Omega(n\log n)$ messages. A series of increasingly efficient election algorithms for rings appeared in the literature [11], [28], [42]. The recent algorithm by van Leeuwen and Tan [49] works for synchronous and asynchronous rings and uses $1.44 n\log n$ messages in the worst case. Similarly, Afek and Gafni [4] and Peterson [41] prove that election in synchronous and asynchronous complete networks requires $\Theta(n\log n)$ messages. Afek and Gafni show that election algorithms for synchronous complete networks require $\Theta(n\log n)$ messages and run in time $\Theta(\log n)$ time units. Next, Afek and Gafni [4], and Peterson [41], develop several asynchronous algorithms for complete networks that have a time complexity of $O(n)$ time units and a message complexity of
E(n\log n) messages.

Election algorithms for some networks can be improved if the processors have additional information besides the topology. For example, Loui, Matsushita, and West [35] show that \( O(n) \) messages suffice for election in asynchronous complete networks if at each processor the label on each link gives the distance, along a fixed Hamiltonian cycle, to the processor at the other end of the link.

Also, Peterson [41] proves that election in asynchronous \textit{square meshes} requires only \( O(n) \) messages if the processors have a \textit{sense of direction} [45], i.e., the processors have a consistent view of the north, east, south, and west direction in the mesh. When each processor can distinguish between its left link and its right link in a \textit{ring}, Dolev, Klawe, and Rodeh [16] present an election algorithm that works for synchronous and asynchronous rings and uses \( 1.356 n \log n \) messages in the worst case.

The references we cited so far assume that the election algorithms are \textit{comparison} algorithms, i.e., the election algorithms are restricted to use only comparisons of processor identifiers. Frederickson and Lynch [22] prove that both the departure from the comparison model and the possibility of using a large number of rounds are necessary in order to obtain an election algorithm of \( O(n) \) message complexity for synchronous rings. Gafni [23] presents a synchronous noncomparison election algorithm for rings. The algorithm is of message complexity \( \Theta(n) \) and of time complexity \( \Theta(n 2^n + |T|^2) \), where \(|T|\) is the cardinality of the set of the node identifiers.

Processor and communication link failures complicate the election problem. The impossibility result of Fischer, Lynch, and Paterson [20] implies that if a \textit{node} may fail by stopping, then no election algorithm exists for \textit{asynchronous} networks even if all the links are reliable. On the other hand, the synchronous algorithms of Pease, Shostak, and Lamport [40], Dolev et al. [15], Dolev and Strong [17], and Coan [13] can be modified to obtain election algorithms for \textit{synchronous} complete networks with Byzantine node failures.

Only recently have algorithms been designed for networks with faulty \textit{links}. Goldreich and Shrira [25] study election in asynchronous \textit{rings} with one \textit{intermittently} faulty link. If \( n \) is known to all
the nodes, then they present an algorithm that uses $\Theta(n \log n)$ messages; otherwise, they develop an algorithm that uses $\Theta(n^2)$ messages. Cimet and Kumar [12] discuss an algorithm that elects a leader when links fail detectably. The type of failure they consider is fail-stop.

1.3 Contents of the Thesis

Chapters 2, 3, and 4 are self-contained papers, except that all the references are grouped together at the end of the thesis. The reader may choose to read any chapter without referring to other chapters.

In Chapter 2, we characterize completely the shared-memory requirements for achieving agreement in an asynchronous system of fail-stop processes that die undetectably. This work, co-authored with Michael C. Loui, appeared in Advances in Computing Research [34]. Section 2.4 appeared in Hosame Hassan Abu-Amara's M.S. thesis [3].

In Chapter 3, we present an algorithm for election in asynchronous complete networks when the links may fail intermittently and undetectably. The algorithm appeared in the IEEE Transactions on Computers [2].

In Chapter 4, we describe an efficient algorithm for election in synchronous square meshes. Also, we prove a lower bound on the number of messages any algorithm uses in synchronous meshes.

In Chapter 5, we discuss some open problems.
CHAPTER 2
MEMORY REQUIREMENTS FOR AGREEMENT
AMONG UNRELIABLE ASYNCHRONOUS PROCESSES

2.1 Introduction

We study an asynchronous system of processes that communicate via a shared memory and die undetectably. Each process has a binary input, and the processes together must agree on a decision that is one of their inputs. We prove that there is no agreement protocol that uses only read and write operations, even if the protocol may assume that at most one process dies. Furthermore, there is no agreement protocol that uses test-and-set operations if the memory cells have only two values and two or more processes may die, but there are test-and-set protocols if either memory cells have three values or at most one process dies. A table in Section 2.6 summarizes our results. Our results imply that Jaffe’s simulation [31] cannot be modified to use two-valued memory cells.

Fischer, Lynch, and Paterson [20] proved that agreement cannot be achieved in asynchronous message-passing systems with the undetectable death of even one process; their result holds a fortiori for Byzantine failures. There are three major differences between their paper and ours, however. First, Fischer, Lynch, and Paterson assumed only one kind of “atomic step” similar to our test-and-set operation; in one step a process receives a message, changes state, and broadcasts a message to all processes. Besides test-and-set protocols, we study protocols with read and write operations. Second, Fischer, Lynch, and Paterson did not consider the message contents, whereas we show that the size of the set of values that cells may store affects whether agreement is possible. Third, Fischer, Lynch, and Paterson assumed a weak communication mechanism: there is no bound on the delay between sending and receiving a message. In contrast, in our shared memory the new value of a cell becomes available for reading immediately after a process writes the cell. Dolev, Dwork, and Stockmeyer [14] called the communication mechanism of Fischer, Lynch, and Paterson asynchronous and our communication mechanism synchronous. Recently, Abrahamson [1] and Herlihy [27] generalized our results.
Section 2.2 defines process systems, agreement protocols, resilience, and computation graphs. Section 2.3 presents three simple agreement protocols. Section 2.4 establishes the impossibility of fully resilient agreement, Section 2.5 the impossibility of k-resilient agreement. Although the results of Section 2.5 supersede the results of Section 2.4, their proofs are significantly more difficult. Since Sections 2.4 and 2.5 are each self-contained, the reader may choose to read only the section of interest.

2.2 Definitions

2.2.1 Process systems

Informally, a process system is a set of asynchronously executing processes in a computer system. The processes communicate via a shared memory. Any memory local to a process is incorporated into its state.

Formally, a process system $S$ consists of the following:

1. A finite set of memory cells, denoted $\text{MEM}(S)$. Let $m = |\text{MEM}(S)|$, and let $\text{VAL}(S)$ be the finite set of possible cell values. For nontriviality, $|\text{VAL}(S)| \geq 2$.

2. A set of processes, denoted $\text{PROC}(S)$. Let $n = |\text{PROC}(S)|$. Let $\text{STATES}(j,S)$ be the finite set of states of process $j$ for $j = 1, \ldots, n$. $\text{STATES}(i,S)$ and $\text{STATES}(j,S)$ are disjoint if $i \neq j$.

3. A cell assignment function

$$cell: \bigcup_{j=1}^{n} \text{STATES}(j,S) \rightarrow \text{MEM}(S)$$

that associates with each process state $q$ in each $\text{STATES}(j,S)$ the cell $cell(q)$ in $\text{MEM}(S)$ that process $j$ accesses when it is in state $q$.

4. Partial process transition functions $\delta_1, \ldots, \delta_n$, with each

$$\delta_j: \text{STATES}(j,S) \times \text{VAL}(S) \rightarrow \text{STATES}(j,S) \times \text{VAL}(S).$$

$\delta_j(q,v)$ may not be defined for some process states $q$.

The set of system states of $S$ is
The state of process $j$ in system state $s$, denoted state $(j,s)$, is the $j$th coordinate. The value of cell $c$ in system state $s$ is denoted value $(c,s)$. In an initial system state all cells have the same value. The state of a process in an initial system state is its initial state.

The process transition functions induce a system transition function

$$\Delta: \{1,2,\ldots,n\} \times \text{SYS}(S) \rightarrow \text{SYS}(S)$$

defined as follows. For $j$ in $\{1,2,\ldots,n\}$ and $s$ in SYS $(S)$ let $q = \text{state} (j,s)$, $c = \text{cell} (q)$, $v = \text{value} (c,s)$, and $\delta_j(q,v) = (q',v')$. Then

$$\text{state} (j,\Delta(j,s)) = q', \text{ and state} (i,\Delta(j,s)) = \text{state} (i,s) \text{ for all } i \neq j;$$

$$\text{value} (c,\Delta(j,s)) = v', \text{ and value} (d,\Delta(j,s)) = \text{value} (d,s) \text{ for all } d \neq c.$$ 

We say that $\Delta(j,s)$ is the new system state after a transition from $s$ by process $j$. The definition of $\Delta(j,s)$ implies that only process $j$ may change state and only cell $c$ may change value. Moreover, the new state of process $j$ and the new value of $c$ depend on no other process states or cell values.

A computation is a sequence of system states

$$s_0, s_1, s_2, \ldots$$

such that $s_0$ is an initial system state, and for every $i$ there is some process $j$ such that $s_{i+1} = \Delta(j,s_i)$. A computation terminates at system state $s_f$ if there is no process $j$ for which $\Delta(j,s_f)$ is defined. Call $s_f$ the final system state in the computation.

2.2.2 Agreement protocols

After we define agreement protocols informally, we give precise definitions.

Every process in $S$ has an input in $\{0,1\}$. An agreement protocol $\Pi$ for $S$ is a protocol such that at the end of every computation induced by $\Pi$ the processes must agree on a common decision that is the input of one of the processes. Furthermore, $\Pi$ has these properties:

(1) every computation induced by $\Pi$ terminates;
(2) Π tolerates undetectable process death.

(3) the processes communicate only through the shared memory.

All processes must eventually agree on either 0 or 1. The processes cannot agree on a fixed value such as 0. For example, if all inputs are 1, then all decisions must be 1.

Formally, for every process j, STATES(j,S) has five special states \( B_j^0, B_j^1, E_j^0, E_j^1, \) and \( D_j \). The initial state of process j is either \( B_j^0 \) or \( B_j^1 \). The input of process j is 0 (1) if its initial state is \( B_j^0 \) (\( B_j^1 \)).

The decision of process j is 0 (1) if \( \text{state}(j,s_f) = E_j^0 \) (\( E_j^1 \)). Process j is dead in system state s if \( \text{state}(j,s) = D_j \). If process j is not dead in system state s, then it is live in s. Section 2.2.3 will discuss dead processes further. Once process j enters \( E_j^0 \) or \( E_j^1 \) or \( D_j \), it makes no further transitions: \( \delta_j(E_j^0,v) \), \( \delta_j(E_j^1,v) \), and \( \delta_j(D_j,v) \) are undefined for all cell values v.

An agreement protocol Π for S is a specification of the process states and process transition functions of S such that

(1) every computation induced by Π terminates;

(2) for every computation induced by Π the decisions of all live processes are the same;

(3) if some process in the final system state \( s_f \) is live, then its decision is the input of one of the processes.

Note that Π must necessarily tolerate undetectable process death because for every j, when process j makes a transition, it does not inspect the state of any other process.

Now we define two kinds of agreement protocols: read/write protocols and test-and-set protocols. To simplify the notation assume that \( \text{VAL}(S) = \{0,1\} \). It is straightforward to modify the following definitions for \( |\text{VAL}(S)| > 2 \).

For read/write protocols a process may atomically read or atomically write a cell. Formally, each state in STATES(j,S) other than \( E_j^0, E_j^1 \), or \( D_j \) can have two forms:
(i) $q = (j, \text{READ}, c, r_0, r_1)$, where $c \in \text{MEM}(S)$ and $r_0, r_1 \in \text{STATES}(j, S)$. Here $\text{cell}(q) = c$.

Furthermore,

$$\delta_j(q, 0) = (r_0, 0) \quad \text{and} \quad \delta_j(q, 1) = (r_1, 1).$$

That is, process $j$ reads cell $c$. If the value of $c$ is 0 (1), then the next state of process $j$ is $r_0$ ($r_1$), and the value of $c$ remains the same. Call $q$ a \textit{READ state}.

(ii) $q = (j, \text{WRITE}, c, r, v)$, where $c \in \text{MEM}(S)$, $r \in \text{STATES}(j, S)$, and $v \in \text{VAL}(S)$. Here $\text{cell}(q) = c$.

Furthermore,

$$\delta_j(q, 0) = \delta_j(q, 1) = (r, v).$$

That is, without inspecting the current value of $c$, process $j$ writes the value $v$ into $c$ and enters state $r$. Call $q$ a \textit{WRITE state}.

For \textit{test-and-set} protocols a process may read (test) a cell, and depending on its value, write (set) a new value into the cell and change state in one atomic step. Formally, each state in $\text{STATES}(j, S)$ other than $E_j^0$, $E_j^1$, or $D_j$ has the form

$$q = (j, c, r_0, r_1, v_0, v_1),$$

where $c \in \text{MEM}(S)$, $r_0, r_1 \in \text{STATES}(j, S)$ and $v_0, v_1 \in \text{VAL}(S)$. Here $\text{cell}(q) = c$. Furthermore,

$$\delta_j(q, 0) = (r_0, v_0) \quad \text{and} \quad \delta_j(q, 1) = (r_1, v_1).$$

That is, if the value of $c$ is 0 (1), then the next state of process $j$ is $r_0$ ($r_1$), and the new value of $c$ is $v_0$ ($v_1$). With a test-and-set operation one can implement both a read operation (by setting $v_0=0$ and $v_1=1$) and a write operation (by setting $r_0=r_1$ and $v_0=v_1$).

This test-and-set operation was popularized by Lynch and Fischer [36]. In the traditional, weaker, test-and-set operation (Peterson and Silberschatz [43]) the process must set the cell to the same value, regardless of the cell's old value. Our test-and-set operation can implement the weak test-and-set operation by setting $v_0=v_1$. 
2.2.3 Process death

Section 2.2.1 defined a deterministic system transition function $\Delta$. We modify the definition to model spontaneous process death—a process may die at any time. Define $\Delta$ to be a nondeterministic partial function induced by the process transition functions as follows. For $j$ in $\{1, 2, \ldots, n\}$ and $s$ in $SYS(S)$ let $q = state(j,s)$, $c = cell(q)$, $v = value(c,s)$, and $\delta_j(q,v) = (q',v')$. Then

\[
state(j, \Delta(j,s)) \in \{q', D_j\};
\]
\[
state(i, \Delta(j,s)) = state(i,s) \text{ for all } i \neq j;
\]
\[
value(c, \Delta(j,s)) = v' \text{ if } state(j, \Delta(j,s)) = q', \text{ but}
\]
\[
value(c, \Delta(j,s)) = value(c,s) \text{ if } state(j, \Delta(j,s)) = D_j; \text{ and}
\]
\[
value(d, \Delta(j,s)) = value(d,s) \text{ for all } d \neq c.
\]

In other words, $\Delta(j,s)$ is one of two new system states. If process $j$ dies by entering state $D_j$, then no cell value changes. In particular, a process may die on its first transition without changing the value of a memory cell. After a process dies, it makes no further transitions.

2.2.4 Resilience

Protocols that allow processes to wait differ subtly from protocols that always force processes to make progress. A protocol is $k$-resilient if it achieves agreement in all computations with at most $k$ process deaths, provided that every live process eventually makes transitions. Call an $(n-1)$-resilient protocol fully resilient. By definition, every $k$-resilient protocol is $k'$-resilient for all $k' < k$. A $k$-resilient protocol could allow a process to wait for other processes to take transitions. For example, standard mutual exclusion protocols are 0-resilient since they assume a perfectly reliable system: in these protocols fast processes wait for slow processes that eventually make transitions. A 1-resilient protocol assumes that at most one process will die; thus all processes can wait for one of two distinguished processes to change the value of a cell. In contrast, in fully resilient protocols live processes must not wait for other processes to make transitions. The simulation of Jaffe [31] requires fully resilient protocols. This simulation assumes that a dead process is indistinguishable from a slow process.
2.2.5 Computation graphs

The following definitions are used in Section 2.4 and Section 2.5.

Let $\Pi$ be an agreement protocol, and let $s_0$ be an initial system state. Define a directed computation graph $\Gamma = (V,A)$ for $\Pi$ as follows. The node set $V$ is the set of system states $s$ such that $s$ has no dead processes and $s$ is in some computation of $\Pi$ starting from some initial system state. There is an arc $(s,t)$ in $A$ if and only if $\Delta(j,s) = r$ for some process $j$; that is, $t$ follows from $s$ after a transition by process $j$. Label arc $(s,t)$ with $j$. Call $t$ a child of $s$. Node $u$ is reachable from node $s$ if there is a directed path from $s$ to $u$.

In the sequel we shall not distinguish between a node $s$ in $V$ and the system state corresponding to $s$. Also, we shall not distinguish between an arc and the process transition corresponding to the arc.

A leaf of $\Gamma$ is a node with no children. By construction, the leaves of $\Gamma$ are system states in which the state of every process $j$ is in $\{E_0^j,E_1^j\}$. Since $\Pi$ is an agreement protocol, the processes at a leaf $t$ must have the same decision. Call $t$ a 0-leaf (1-leaf) if the decision of the processes in system state $t$ is 0 (1). Call a node $s$ bivalent if there is a 0-leaf reachable from $s$ and a 1-leaf reachable from $s$. Call a node $s$ 0-valent if every leaf reachable from $s$ is a 0-leaf. By definition, every node reachable from a 0-valent node is 0-valent. Also, if $state(j,s) = E_0^j$ for some process $j$, then $s$ is 0-valent because $j$ will make no further transitions, and all processes will agree with the decision of $j$. Similarly, call a node $s$ 1-valent if every leaf reachable from $s$ is a 1-leaf. Call node $s$ univalent if it is either 0-valent or 1-valent. Every node is either bivalent or univalent.

2.3 Three Agreement Protocols

This section describes three simple test-and-set protocols that motivate the main results of Sections 4 and 5. We present the protocols informally; it is straightforward to express the protocols in terms of process states and process transition functions.

First, let $PROC(S) = \{1,2, \ldots, n\}$. We can guarantee that all processes achieve the same decision by using one three-valued cell $c$. That is, $VAL(S) = \{0,1,\#\}$, and $MEM(S) = \{c\}$. Assume that the initial
value of $c$ is # (blank). In the agreement protocol for this system the first process to take a transition sets $c$ to its input, and all later live processes adopt this value as their decisions.

Protocol 1. Every process executes the following:

1. Test-and-set $c$. If the value of $c$ is #, then set $c$ to 0 (1) if the input is 0 (1) and keep the input as the decision. If the value of $c$ is not #, then set the decision to the value of $c$.

Theorem 3.1: For any number of processes, there is a fully resilient agreement protocol that uses test-and-set operations on one three-valued cell.

Second, consider a system $S$ with two processes, $PROC(S) = \{1, 2\}$, and two-valued cells, $VAL(S) = \{0, 1\}$. Assume that the initial value of every cell is 0. It is easy to design an agreement protocol with three cells: each process uses one cell to announce its input, and then the processes test-and-set the third cell to 1; the decision of both processes is the input of the process that changed this cell from 0 to 1. We show that just two cells suffice. Let $MEM(S) = (c, d)$. Both processes execute the following protocol.

Protocol 2.

1. Test-and-set $c$. If the value of $c$ is 1, then leave $c$ unchanged, set the decision to 1, and halt. If the value of $c$ is 0, then set the value of $c$ to the input, and continue.
2. Test-and-set $d$. If the value of $d$ is 0, then set $d$ to 1, keep the input as the decision, and halt. If the value of $d$ is 1, then leave $d$ unchanged, and continue.
3. Read $c$. If the value of $c$ is 0, then set the decision to 0. If the value of $c$ is 1, then set the decision to the complement of the input; that is, if the input is 0 (1), then the decision is 1 (0).

Theorem 3.2: For two processes there is an agreement protocol that uses test-and-set operations on two two-valued cells.

Proof: We establish the correctness of Protocol 2. Call the processes George and Hannah, and assume that George performs the test-and-set on $c$ first.
Case 1: The input of George is 1. Since George sets \( c \) to 1, Hannah determines that the value of \( c \) is 1, and Hannah halts with decision 1 without accessing \( d \). Thus, George must find that \( d \) is 0. After George sets \( d \) to 1, it halts with decision 1.

Case 2: The input of George is 0, and George sets \( d \) to 1 first. In this case George retains 0 as its decision. After Hannah finds that \( d \) is 1, it reads \( c \) again. The value of \( c \) at this time is Hannah's input. If \( c \) is 0, then Hannah halts with decision 0, agreeing with George. If \( c \) is 1, then Hannah sets its decision to 0, the complement of its input, again agreeing with George.

Case 3: The input of George is 0, and Hannah sets \( d \) to 1 first. In this case Hannah retains its input as its decision. After George finds that \( d \) is 1, it reads \( c \) again. The value of \( c \) at this time is Hannah's input. If \( c \) is 0, then George halts with decision 0, agreeing with Hannah. If \( c \) is 1, then George sets its decision to 1, the complement of its input, again agreeing with Hannah.

Section 2.4.3 shows that there is no fully resilient test-and-set protocol with two-valued cells if there are more than two processes.

Third, we present a 1-resilient test-and-set protocol for \( n \) processes with four two-valued cells. Consider a system \( S \) with \( PROC(S) = \{1,2,\ldots,n\}, \ MEM(S) = \{c,d,e,f\}, \) and \( VAL(S) = \{0,1\} \). Assume that the initial value of every cell is 0. In the agreement protocol for this system process 1 and process 2 execute Protocol 2 using cells \( c \) and \( d \). Then both write the decision into cell \( e \) and set cell \( f \) ("finished") to 1 to signify the completion of the protocol. Processes 3, \ldots, \( n \) wait until either process 1 or process 2 sets \( f \) to 1.


A1. Test-and-set \( c \). If the value of \( c \) is 1, then leave \( c \) unchanged, set the decision to 1, and go to step A4. If the value of \( c \) is 0, then set the value of \( c \) to the input, and continue.

A2. Test-and-set \( d \). If the value of \( d \) is 0, then set \( d \) to 1, keep the input as the decision, and go to step A4. If the value of \( d \) is 1, then leave \( d \) unchanged, and continue.
A3. Read $c$. If the value of $c$ is 0, then set the decision to 0. If the value of $c$ is 1, then set the decision to the complement of the input.

A4. Write the decision into $e$.

A5. Write 1 into $f$.

Processes $3, \ldots, n$ execute B1 and B2 below.

B1. Repeatedly read $f$ until its value is 1.

B2. Read $e$ and set the decision to the value of $e$.

Theorem 3.3: For any number of processes, there is a 1-resilient agreement protocol that uses test-and-set operations on four two-valued cells.

Proof: We establish the correctness of Protocol 3 for computations in which at most one process dies.

By the correctness of Protocol 2, both process 1 and process 2 choose the same decision $v$. Since at most one process dies, either process 1 or process 2 sets cell $e$ to $v$ and cell $f$ to 1. Once $f$ is set to 1, the value of $e$ will not change. Therefore, processes $3, \ldots, n$ will eventually read $e$ and set their decisions to $v$. □

Section 2.5.3 shows that there is no 2-resilient test-and-set protocol with two-valued cells.

2.4 Impossibility of Fully Resilient Agreement

2.4.1 Properties of fully resilient protocols

Let $\Pi$ be a fully resilient agreement protocol on the process system with $n$ processes. Let $\Gamma$ be the computation graph for $\Pi$.

Lemma 4.1: $\Gamma$ is acyclic.

Proof: Suppose, on the contrary, $\Gamma$ has a directed cycle

$$s_1, s_2, \ldots, s_k, s_1.$$ 

The arcs in this cycle define an infinite sequence of process transitions because the participating processes do not have decisions. Because this computation does not terminate, this contradicts the hypothesis that $\Pi$ is a fully resilient agreement protocol. □
For a node \( s \) and a process \( j \), consider the directed path in \( \Gamma \) that starts at \( s \) and follows only arcs labeled \( j \). Since \( \Pi \) is fully resilient, it achieves agreement even if all processes but \( j \) die; hence, this path must end at a system state \( r \) in which process \( j \) has a decision. That is, \( \text{state}(j, r) \in \{E_j^0, E_j^1\} \). Define \( \text{endpath}(j, s) \) to be this system state \( r \).

**Lemma 4.2:** There is a bivalent initial system state.

**Proof:** Let \( G \) and \( H \) be processes. Let \( s_0 \) be an initial system state in which the input of process \( G \) is 0 and the input of \( H \) is 1. Let \( s_G = \text{endpath}(G, s_0) \) and \( s_H = \text{endpath}(H, s_0) \). When \( G \) reaches its state in \( s_G \) it must have a decision that should be valid even if all other processes died. Thus the decision of \( G \) in \( s_G \) is 0. It follows that the decisions of all processes in all leaves that are successors of \( s_G \) must be 0. Similarly the decisions of all processes in all leaves that are successors of \( s_H \) must be 1. \( \square \)

**Lemma 4.3:** Every bivalent node has \( n \) children.

**Proof:** Let \( s \) be a node with fewer than \( n \) children. There is some process \( j \) with no transition from \( s \). Then \( \text{state}(j, s) \) is either \( E_j^0 \) or \( E_j^1 \). In the first case the decision of every process in every leaf reachable from \( s \) must be 0, in the second case 1. Consequently, \( s \) is univalent. \( \square \)

**Lemma 4.4:** There is a bivalent node \( s \) whose children are all univalent.

**Proof:** Let \( W \) be the bivalent nodes of \( \Gamma \). By Lemma 4.2, \( W \) is not empty. By Lemma 4.3, each node in \( W \) has \( n \) children. If every node in \( W \) had a child in \( W \), then since \( \Gamma \) is acyclic, \( \Gamma \) would be infinite. Because \( \Gamma \) is finite, there is a node \( s \) in \( W \) such that all children of \( s \) are not in \( W \). \( \square \)

Lemma 4.4 asserts the existence of a crucial node \( s \) such that every transition from \( s \) compels an irrevocable decision. We show that each of these transitions must change the value of a cell.

**Lemma 4.5:** Let \( j \) be a process and \( s \) and \( t \) be system states such that \( \text{state}(j, s) = \text{state}(j, t) \) and \( \text{value}(c, s) = \text{value}(c, t) \) for all memory cells \( c \). Then the decisions of process \( j \) in \( \text{endpath}(j, s) \) and in \( \text{endpath}(j, t) \) are the same.
Proof: Let $s' = \Delta(j,s)$ and $t' = \Delta(j,t)$. The hypotheses on $s$ and $t$ guarantee that $\text{state}(j,s') = \text{state}(j,t')$ and $\text{value}(c,s') = \text{value}(c,t')$ for all memory cells $c$. By induction, the states of process $j$ in $\text{endpath}(j,s)$ and in $\text{endpath}(j,t)$ are the same. □

Lemma 4.6: Let $s$ be a bivalent node whose children are all univalent. For every process $j$, the transition by $j$ from $s$ changes the value of the cell that $j$ accesses in $s$.

Proof: Let George ($G$) and Hannah ($H$) be any processes such that $w = \Delta(G,s)$ is a 0-valent child of $s$ and $x = \Delta(H,s)$ is a 1-valent child of $s$. Let $c_G = \text{cell}(\text{state}(G,s))$ and $c_H = \text{cell}(\text{state}(H,s))$. If, to the contrary, the transition by George from $s$ to $w$ does not change the value of $c_G$, then for every cell $d$, $\text{value}(d,s) = \text{value}(d,w)$. Consequently, by Lemma 4.5, since $\text{state}(H,w) = \text{state}(H,s)$, the decisions of Hannah in $\text{endpath}(H,w)$ and in $\text{endpath}(H,s)$ would be the same. But this is impossible because $w$ is 0-valent and $x$ is 1-valent. Similarly, the transition by Hannah from $s$ to $x$ changes the value of $c_H$. □

2.4.2 Read/write protocols

Theorem 4.1: There is no fully resilient read/write agreement protocol for $n \geq 2$.

Proof: Suppose $\Pi$ is a fully resilient read/write agreement protocol. In the computation graph for $\Pi$ Lemma 4.4 guarantees the existence of a bivalent node $s$ whose children are all univalent. Choose a 0-valent child $w$ of $s$ and a 1-valent child $x$ of $s$. Let George ($G$) and Hannah ($H$) be the processes such that $w = \Delta(G,s)$ and $x = \Delta(H,s)$. Let $y = \Delta(H,w)$ and $z = \Delta(G,x)$. Since $w$ is 0-valent, $y$ is 0-valent. Since $x$ is 1-valent, $z$ is 1-valent. See Figure 2.1.

Let $q_G = \text{state}(G,s)$ and $q_H = \text{state}(H,s)$. Lemma 4.6 guarantees that both $q_G$ and $q_H$ are WRITE states. Let $c_G = \text{cell}(q_G)$ and $c_H = \text{cell}(q_H)$. We examine two cases, both of which lead to contradictions.

Case 1: $c_G \neq c_H$. Since the transitions by George and by Hannah affect different memory cells, system states $y$ and $z$ are the same. But $y = z$ cannot be both 0-valent and 1-valent.

Case 2: $c_G = c_H$. Since the transition by George from $x$ to $z$ obliterates the value written by Hannah in the transition from $s$ to $x$, $\text{value}(c_G,w) = \text{value}(c_G,z)$. Indeed, $\text{value}(d,w) = \text{value}(d,z)$ for all cells $d$. 


Furthermore, George's state is the same in both w and z. By Lemma 4.5, George's decision in endpath(G,w) is the same as in endpath(G,z). But this is impossible because w is 0-valent and z is 1-valent. 

Theorem 4.1 is similar in spirit to Theorem 1 of Johnson and Schneider [32].

2.4.3 Test-and-set protocols

Theorem 4.2: There is no fully resilient test-and-set agreement protocol if $n \geq 3$ and memory cells have only two values.

Proof: Suppose $\Pi$ is a fully resilient test-and-set agreement protocol for a system with $n \geq 3$ processes and two-valued memory cells. In the computation graph for $\Pi$ Lemma 4.4 guarantees the existence of a bivalent node $s$ whose children are all univalent. Choose a 0-valent child $w$ of $s$ and a 1-valent child $x$ of $s$. 

Figure 2.1. The Transitions by $G$ and $H$ from $s$
Let George \((G)\), and Hannah \((H)\) be the processes such that \(w = \Delta(G,s)\) and \(x = \Delta(H,s)\). Let \(y = \Delta(H,w)\) and \(z = \Delta(G,x)\). Since \(w\) is 0-valent, \(y\) is 0-valent; since \(x\) is 1-valent, \(z\) is 1-valent. See Figure 2.1.

Let \(c_G = \text{cell}(\text{state}(G,s))\) and \(c_H = \text{cell}(\text{state}(H,s))\). We examine two cases, both of which lead to contradictions.

**Case 1:** \(c_G \neq c_H\). Since George and Hannah access different cells, system states \(y\) and \(z\) are the same. But \(y = z\) cannot be both 0-valent and 1-valent.

**Case 2:** \(c_G = c_H\). Let \(c = c_G = c_H\). Because \(c\) can attain only two different values and because, by Lemma 4.6, both the transition by George to \(w\) and the transition by Hannah to \(x\) change the value of \(c\), it follows that \(\text{value}(c,w) = \text{value}(c,x)\). Since \(n \geq 3\), there is a process Frances \((F)\) different from both George and Hannah. By Lemma 4.5, since \(\text{state}(F,w) = \text{state}(F,x) = \text{state}(F,s)\), the decisions of Frances in \(\text{endpath}(F,w)\) and in \(\text{endpath}(F,x)\) are the same. This is a contradiction because \(w\) is 0-valent and \(x\) is 1-valent.\(\square\)

### 2.5 Impossibility of \(k\)-Resilient Agreement

We establish that there is no 1-resilient read/write agreement protocol and no 2-resilient test-and-set agreement protocol. These results imply Theorems 4.1 and 4.2, but their proofs are more subtle. The proofs resemble the arguments of Dolev, Dwork, and Stockmeyer [14], who considered only message-passing systems.

#### 2.5.1 Properties of \(k\)-resilient protocols

Let \(S\) be a system of \(n\) processes; we allow processes to have an infinite number of states. Let \(\Pi\) be a \(k\)-resilient agreement protocol for \(S\) with \(k \geq 1\). Let \(\Gamma\) be the computation graph for \(\Pi\). If \(k < n - 1\), then since \(\Pi\) may allow processes to wait, \(\Gamma\) may have directed cycles. For example, in Protocol 3 in Section 2.3, process 3 repeatedly reads \(f\) until process 1 or process 2 writes 1 into \(f\). Furthermore, if the number of process states is infinite, then \(\Gamma\) would be infinite. The proofs in Section 2.4 require that \(\Gamma\) be acyclic and finite.
In $\Gamma$ let $R$ be a directed path, possibly with repeated nodes, from node $s$ to node $t$. Let $\text{nodes}(R)$ be the sequence of nodes visited by $R$, including $s$ and $t$. For a set of processes $P$ call $R$ $P$-free if $R$ has no transitions by processes in $P$. If $P$ has at most $k$ processes, then since $\Pi$ is $k$-resilient, from every node there is a $P$-free path to a node in which all processes not in $P$ have reached the same decision.

Lemma 5.1: Let $s$ and $t$ be univalent nodes such that $\text{value}(c,s) = \text{value}(c,t)$ for all memory cells $c$. Let $P = \{ p : \text{state}(p,s) \neq \text{state}(p,t) \}$. If $|P| \leq k$, then $s$ and $t$ are either both 0-valent or both 1-valent.

Proof: Suppose $s$ is 0-valent; the case of a 1-valent $s$ is similar. Since $\Pi$ is $k$-resilient, there is a $P$-free path $R$ from $s$ to a node in which all processes not in $P$ have reached 0 decisions. Let $j_1, \ldots, j_r$ be the labels of arcs of $R$, in sequence. Since $R$ is $P$-free, none of $j_1, \ldots, j_r$ is in $P$. Thus from the viewpoint of processes $j_1, \ldots, j_r$, system states $s$ and $t$ look the same. Define $s_0 = s$ and $t_0 = t$, and for $i = 1, \ldots, r$, $s_i = \Delta(j_i, s_{i-1})$ and $t_i = \Delta(j_i, t_{i-1})$. A straightforward inductive argument yields for $i = 0, \ldots, r$, \( \text{state}(j,s_i) = \text{state}(j,t_i) \) for all $j \notin P$, and $\text{value}(c,s_i) = \text{value}(c,t_i)$ for all memory cells $c$. By construction of $R$, in $s$, all processes not in $P$ have reached 0 decisions. Consequently, since $\text{state}(j,s_i) = \text{state}(j,t_i)$ for all $j \notin P$, in $t$, all processes not in $P$ have reached 0 decisions. Therefore, since $t$ is univalent, $t$ must be 0-valent. \( \Box \)

Lemma 5.2: There is a bivalent initial system state.

Proof: Suppose, to the contrary, every initial system state is univalent. Since $\Pi$ is an agreement protocol, the initial system state in which all processes have input 0 is 0-valent. Similarly, the initial system state in which all processes have input 1 is 1-valent. By changing each input from 0 to 1 one at a time, we obtain initial system states $s_0$ and $s_1$ such that $s_0$ is 0-valent, $s_1$ is 1-valent, and there is exactly one process $h$ such that $\text{state}(h,s_0) \neq \text{state}(h,s_1)$. Since $\Pi$ is $k$-resilient, it is at least 1-resilient, and we have obtained a contradiction of Lemma 5.1. \( \Box \)
2.5.2 1-Resilient read/write protocols

Throughout this section assume that \( \Pi \) is a 1-resilient read/write agreement protocol and that \( \Gamma \) is the computation graph for \( \Pi \). We shall construct a path in \( \Gamma \) in which every process takes infinitely many transitions, but no process reaches a decision.

**Lemma 5.3:** For every bivalent node \( s \) and every process \( j \) there is a node \( t \) reachable from \( s \) such that \( \Delta(j,t) \) is bivalent.

**Proof:** Let George \((G)\) be a process. Suppose, to the contrary, for every \( t \) reachable from \( s \), \( \Delta(G,t) \) is univalent. In particular, \( \Delta(G,s) \) is univalent. Without loss of generality, assume \( \Delta(G,s) \) is 0-valent.

Since \( s \) is bivalent, there is a path from \( s \) to a 1-leaf. Let \( s' \) be the node on this path such that \( \text{state}(G,\Delta(G,s')) = E_G^1 \). Node \( \Delta(G,s') \) is 1-valent.

Let \( R \) be a path from \( s \) to \( s' \). By assumption, \( \Delta(G,t) \) is univalent for all \( t \) in \( \text{nodes}(R) \). Since \( \Delta(G,s) \) is 0-valent and \( \Delta(G,s') \) is 1-valent, it is straightforward to prove by induction on the length of \( R \) that there exist consecutive nodes \( u \) and \( x \) in \( \text{nodes}(R) \) such that \( \Delta(G,u) \) is 0-valent and \( \Delta(G,x) \) is 1-valent. Let Hannah \((H)\) be the process such that \( x = \Delta(H,u) \). Let \( w = \Delta(G,u) \), \( y = \Delta(H,w) \), and \( z = \Delta(G,x) \). Since \( w \) is 0-valent, \( y \) is 0-valent. See Figure 2.2.

Let \( q_G = \text{state}(G,u) = \text{state}(G,x) \) and \( q_H = \text{state}(H,u) \). First, we claim that \( q_G \) and \( q_H \) must be **WRITE** states. Suppose, to the contrary, \( q_G \) is a **READ** state. Since the values of all cells are the same in \( u \) and in \( w \), Hannah undergoes the same transition from both \( u \) and \( w \). Consequently, the values of all cells are the same in \( y \) and in \( x \); hence, the values of all cells are the same in \( y \) and in \( z \). The states of all processes other than George are the same in \( y \) and in \( z \). This is a contradiction of Lemma 5.1 because \( y \) is 0-valent and \( z \) is 1-valent. Thus \( q_G \) must be a **WRITE** state. Similarly, \( q_H \) must be a **WRITE** state.

Let \( c_G = \text{cell}(q_G) \) and \( c_H = \text{cell}(q_H) \). We examine two cases, both of which lead to contradictions.

**Case 1:** \( c_G \neq c_H \). Since the transitions by George and by Hannah affect different memory cells, system states \( y \) and \( z \) are the same. But \( y = z \) cannot be both 0-valent and 1-valent.
Case 2: $c_G = c_H$. Since the transition by George from $x$ to $z$ obliterates the value written by Hannah in the transition from $u$ to $x$, all cells have the same values in $w$ and in $z$. Furthermore, all processes other than Hannah are in the same state in $w$ and in $z$. This contradicts Lemma 5.1 because $w$ is 0-valent and $z$ is 1-valent. □

Theorem 5.1: There is no 1-resilient read/write agreement protocol.

Proof: Suppose there is a 1-resilient read/write agreement protocol $\Pi$ for a system with $n$ processes numbered $0, \ldots, n-1$. Let $s_0$ be a bivalent initial system state guaranteed by Lemma 5.2. Define infinite sequences of paths $R_0, R_1, \ldots$ and nodes $s_1, s_2, \ldots$ as follows: for $i = 0, 1, \ldots$, use Lemma 5.3 to find a path $R_i$ starting from $s_i$ to a node $t_i$ such that $\Delta((i \mod n), t_i)$ is bivalent, and define

$$s_{i+1} = \Delta((i \mod n), t_i).$$

The concatenation of nodes $(R_0), (R_1), \ldots$ is a computation of $\Pi$ in which every process makes
infinitely many transitions, but no process reaches a decision. Thus Π may not terminate.

Contradiction.

2.5.3 2-Resilient test-and-set protocols

Consider a system with \( n \geq 3 \) processes in which cells have only two values. Suppose \( \Pi \) is a 2-resilient test-and-set agreement protocol. First we establish two Lemmas about the computation graph of \( \Pi \).

Lemma 5.4: No bivalent node has both a 0-valent child and a 1-valent child.

Proof: Suppose, to the contrary, bivalent node \( s \) has a 0-valent child \( w \) and a 1-valent child \( x \). Let George (G), and Hannah (H) be the processes such that \( w = \Delta(G,s) \) and \( x = \Delta(H,s) \). Let \( y = \Delta(H,w) \) and \( z = \Delta(G,x) \). Since \( w \) is 0-valent, \( y \) is 0-valent; since \( x \) is 1-valent, \( z \) is 1-valent. See Figure 2.1.

Let \( c_G = \text{cell} (\text{state} (G,s)) \) and \( c_H = \text{cell} (\text{state} (H,s)) \). First, we claim that \( c_G = c_H \). If \( c_G \neq c_H \), then the transitions by George and by Hannah affect different cells; hence, \( y \) and \( z \) are the same. But \( y = z \) cannot be both 0-valent and 1-valent.

Let \( c = c_G = c_H \). There are three cases, all of which lead to contradictions.

Case 1: \( \text{value} (c,w) = \text{value} (c,s) \). The transition by Hannah from \( s \) to \( x \) is the same as from \( w \) to \( y \).

All cells have the same values in \( x \) and in \( y \). All processes other than George are in the same states in \( x \) and in \( y \). Apply Lemma 5.1 to \( x \) and \( y \) to obtain a contradiction.

Case 2: \( \text{value} (c,x) = \text{value} (c,s) \). Similar to Case 1.

Case 3: \( \text{value} (c,w) \neq \text{value} (c,s) \) and \( \text{value} (c,x) \neq \text{value} (c,s) \). Since \( c \) can attain only two values, \( \text{value} (c,w) = \text{value} (c,x) \). Thus all cells have the same values in \( w \) and in \( x \). All processes other than George and Hannah are in the same states in \( w \) and in \( x \). Since \( \Pi \) is 2-resilient and there are at least 3 processes, apply Lemma 5.1 to \( w \) and \( x \) to obtain a contradiction.

Lemma 5.5: For every bivalent node \( s \) and every two distinct processes \( i \) and \( j \), there is a path from \( s \) to a node \( t \) such that either \( \Delta(i,t) \) or \( \Delta(j,t) \) is bivalent.
Proof: Let George (G) and Hannah (H) be two distinct processes. Suppose, to the contrary, for every t reachable from s, \(A(G,t)\) and \(A(H,t)\) are univalent. Without loss of generality assume \(A(G,s)\) is 0-valent. By Lemma 5.4, \(A(H,s)\) is 0-valent.

Since \(s\) is bivalent, there is a path from \(s\) to a 1-leaf. Without loss of generality, assume that George reaches a 1 decision before Hannah does on this path. Let \(s'\) be the node on this path such that \(\text{state}(G,A(G,s')) = E_1^G\). Node \(A(G,s')\) is 1-valent. By Lemma 5.4, \(A(H,s')\) is either bivalent or 1-valent. By assumption, since \(s\) is reachable from \(s\), \(A(H,s)\) is 1-valent.

Let \(R\) be a path from \(s\) to \(s'\). By assumption, \(A(G,t)\) and \(A(H,t)\) are both univalent for all \(t\) in nodes \((R)\). Therefore, there exist consecutive nodes \(t_1\) and \(t_2\) in nodes \((R)\) such that both \(A(G,t_1)\) and \(A(H,t_1)\) are 0-valent, and both \(A(G,t_2)\) and \(A(H,t_2)\) are 1-valent. Let Frances (F) be the process such that \(t_2 = \Delta(F,t_1)\).

Let \(t_3 = \Delta(G,t_1), t_4 = \Delta(G,t_2), t_5 = \Delta(H,t_2), t_6 = \Delta(F,t_3), t_7 = \Delta(H,t_3),\) and \(t_8 = \Delta(H,t_6)\). Nodes \(t_6, t_7,\) and \(t_8\) are 0-valent because \(t_3\) is 0-valent. See Figure 2.3. Let \(c_F = \text{cell}((\text{state}(F,t_1))), c_G = \text{cell}((\text{state}(G,t_1))),\) and \(c_H = \text{cell}((\text{state}(H,t_1)))\).

First we claim that \(c_F = c_G\). If \(c_F \neq c_G\), then system states \(t_4\) and \(t_6\) would be the same. But \(t_4\) is 1-valent and \(t_6\) is 0-valent.

Let \(c = c_F = c_G\). There are three cases, all of which lead to contradictions.

Case 1: \(\text{value}(c,t_2) = \text{value}(c,t_1)\). Then George undergoes the same transition from \(t_1\) to \(t_3\) as from \(t_2\) to \(t_4\). All cells have the same values in \(t_3\) and in \(t_4\). All processes other than Frances are in the same states in \(t_3\) and in \(t_4\). Apply Lemma 5.1 to \(t_3\) and \(t_4\) to obtain a contradiction.

Case 2: \(\text{value}(c,t_3) = \text{value}(c,t_1)\). The transitions of Frances and Hannah are the same from \(t_1\) to \(t_5\) as from \(t_3\) to \(t_8\). All cells have the same values in \(t_5\) and in \(t_8\). All processes other than George are in the same states in \(t_5\) and in \(t_8\). Apply Lemma 5.1 to \(t_5\) and \(t_8\) to obtain a contradiction.

Case 3: \(\text{value}(c,t_2) \neq \text{value}(c,t_1)\) and \(\text{value}(c,t_3) \neq \text{value}(c,t_1)\). Since \(c\) can have only two different values, \(\text{value}(c,t_2) = \text{value}(c,t_5)\). Hannah undergoes the same transition from \(t_2\) to \(t_5\) as from
Figure 2.3. The Transitions by F, G, and H

$t_3$ to $t_7$. All cells have the same values in $t_3$ and in $t_7$. All processes other than George and Frances are in the same states in $t_3$ and in $t_7$. Apply Lemma 5.1 to $t_3$ and $t_7$ to obtain a contradiction.

Theorem 5.2: There is no 2-resilient test-and-set agreement protocol if $n \geq 3$ and memory cells have only two values.

Proof: Suppose $\Pi$ is a 2-resilient test-and-set agreement protocol. To obtain a contradiction, we construct a computation of $\Pi$ in which at least $n-1$ processes make infinitely many transitions, but no process reaches a decision. Let $s_0$ be a bivalent initial system state guaranteed by Lemma 5.2. Let $P_0 = PROC(S)$. Define infinite sequences of paths $R_0, R_1, \ldots$ and nodes $s_1, s_2, \ldots$ as follows:
for $i = 0, 1, \ldots$ do
if $|P_i| = 1$ then $P_i := \text{PROC}(S)$ end if;
Choose any $G_i$ and $H_i$ in $P_i$;
Use Lemma 5.5 to find a path $R_i$ starting from $s_i$ to a node $t_i$ such that either $\Delta(G_i, t_i)$ or $\Delta(H_i, t_i)$ is bivalent;
if $\Delta(G_i, t_i)$ is bivalent
then $s_{i+1} := \Delta(G_i, t_i)$; $P_{i+1} := P_i \setminus \{G_i\}$
else if $\Delta(H_i, t_i)$ is bivalent
then $s_{i+1} := \Delta(H_i, t_i)$; $P_{i+1} := P_i \setminus \{H_i\}$
end if
end for

By construction, $s_1, \ldots, s_{n-1}$ are defined by transitions of $n-1$ different processes; $s_n, \ldots, s_{2n-2}$ are defined by transitions of $n-1$ different processes; and so on. Ergo, the concatenation of $\text{nodes}(R_0)$, $\text{nodes}(R_1), \ldots$ is a nonterminating computation of $\Pi$ in which at most one process makes only finitely many transitions. □

### 2.6 Summary

Table 2.1 summarizes our results on the existence of agreement protocols for asynchronous shared memory systems in which processes die undetectably.

Theorem 5.1 implies the impossibility of 1-resilient Byzantine agreement in asynchronous message-passing systems because a shared memory can simulate message-passing. More precisely, if there were a 1-resilient agreement protocol for an asynchronous message-passing system with undetectable process death, then we could transform it into a 1-resilient read/write protocol on a shared memory. For every pair of processes $(i, j)$ with $i \neq j$, allocate a group of cells to hold all the messages that

### Table 2.1. Summary of the Results of Chapter 2

<table>
<thead>
<tr>
<th>Type of Protocol</th>
<th>1-resilient</th>
<th>$k$-resilient for $k \geq 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/write</td>
<td>No (Theorem 5.1)</td>
<td>No (Theorem 5.1)</td>
</tr>
<tr>
<td>Test-and-set three-valued cells</td>
<td>Yes (Theorem 3.1)</td>
<td>Yes (Theorem 3.1)</td>
</tr>
<tr>
<td>Test-and-set two-valued cells</td>
<td>Yes (Theorem 3.3)</td>
<td>No (Theorem 5.2)</td>
</tr>
</tbody>
</table>
process \( i \) could send to process \( j \). To simulate the sending of a message, process \( i \) writes into the next unused cell in this group.

Recently Burns (personal communication) generalized our results. He determined relationships between the number of possible inputs and the number of memory cell values required to guarantee agreement. Furthermore, his protocols use the weak test-and-set operation.
3.1 Introduction

We consider the election problem on asynchronous complete networks when the processors are reliable but some of the links may be faulty. The failure type we consider is more malicious than fail-stop failure but less malicious than Byzantine failure. The faulty links fail by losing messages at will. Thus a faulty link may act as an adversary who deletes a message at the most inopportune time. Since the network is asynchronous, the link delays are arbitrary; hence, the processors cannot distinguish between slow links and faulty links. In other words, the faulty links are undetectable. We call the type of failure that we consider intermittent. Bar-Yehuda et al. [7] solve the same problem for fail-stop link failure. They also assume that if a link fails, then it fails before the execution of the algorithm. Their algorithm does not tolerate intermittent link failure, and there is no easy way to generalize their algorithm to handle intermittent failure. Our work is independent of their work.

Let \( n \) be the number of processors in the network. Let \( f \) be the maximum number of faulty links, where \( 1 \leq f \leq \lfloor \frac{n}{2} - 3 \rfloor \), and let \( r \) be a design parameter, where \( 1 + \frac{1}{f} \leq r \leq \frac{n-4}{2f} \). We develop an asynchronous algorithm that runs in time \( O(\frac{n}{r-1}) \), uses \( O(nrf + \frac{nr}{r-1} \log(\frac{n}{(r-1)f})) \) messages, and uses at most \( O(\log |T|) \) bits per message, where \(|T|\) is the cardinality of the set of node identifiers. The value of \( r \) that minimizes the total number of messages in our algorithm is \( r \)

\[
= \min(1 + C(\frac{\ln(n+2)}{f})^{1/2}, \frac{n-4}{2f}),
\]

where \( C = 1 + O(\frac{\log f}{\log n}) \). For every value of \( n \) and \( f \) subject to \( f \leq \lfloor \frac{n}{2} - 3 \rfloor \), \( \frac{7}{8} \leq C \leq 1 \). Thus the minimum number of messages that our algorithm uses is \( O(nf + n\log n) \). Since the complete network algorithm of Bar-Yehuda et al. [7] also uses \( \Theta(nf + n\log n) \) messages, our algorithm subsumes theirs.
3.2 Model

Our model follows Goldreich and Shira’s model [25]. Consider an asynchronous complete network of \( n \) processors. We model the network as a complete graph on \( n \) nodes, in which each node represents a processor, and each edge represents a bidirectional communication link. Henceforth, we will not distinguish between a node and the processor it represents, and we will not distinguish between an edge and the link it represents. Each node \( u \) has a unique identifier, \( ID(u) \), chosen from a totally ordered set. No node initially knows the identifier of any other node, but the nodes know that the network is complete. When a node \( u \) wishes to communicate with a node \( v \), then \( u \) sends a message to \( v \) on the link \( l(u,v) \) joining them.

A distributed algorithm on the network is a set of \( n \) deterministic local programs, each assigned to a node. Each local program consists of computation statements and communication statements. The computation statements control the internal computations of a node. The communication statements are of the form “send message \( M \) on link \( l \)” or “receive message \( M’ \) on link \( l \)” Each node \( u \) has a Send-Buffer\((u,l)\) and a Receive-Buffer\((u,l)\) associated with each link \( l \) incident on \( u \), where the buffers are not necessarily first-in first-out. Let \( l \) be \( l(u,v) \). When \( u \) wishes to send a message \( M \) on \( l \), \( u \) places \( M \) in Send-Buffer\((u,l)\). We call this event a send event. To capture the asynchronous nature of our network, messages may remain in the send-buffers for arbitrary lengths of time. A transmission event in \( l \) occurs when \( l \) places \( M \) in Receive-Buffer\((v,l)\). We assume that \( u \) can not inspect Send-Buffer\((u,l)\) to check whether \( M \) was removed from the buffer. Hence \( M \) is in transit from \( u \) to \( v \) if \( M \) is in Send-Buffer\((u,l)\). If \( u \) wishes to process a message \( M’ \) on \( l \), then \( u \) removes \( M’ \) from Receive-Buffer\((u,l)\). We call this event a receive event. For convenience, we assume that it takes one time unit to remove \( M’ \) from Receive-Buffer\((u,l)\) and to execute the computation statements on \( M’ \). If \( M’ \) is not in Receive-Buffer\((u,l)\), then \( u \) either waits for \( M’ \), or \( u \) receives some other message, depending on \( u \)'s local program. Note that when we say that node \( u \) receives a message, we mean that \( u \) removes the message from a Receive-Buffer and processes the message. A loss event in a link \( l \) is the event of \( l \) discarding a message.
Consider a particular execution $E$ of a distributed algorithm. Let $Events(E)$ be the multiset of the events in $E$. For convenience, we assume the existence of a global clock that gives the time at which each event in $E$ occurs. Although this clock is available to an observer of the network, the nodes do not know of its existence. We will assume that each event in $E$ occurs at some discrete unit of time starting from zero. Let $Events(u)$ be the multiset of $u$'s send and receive events in $E$. The local program in $u$ induces a total ordering on $Events(u)$. Unlike Goldreich and Shrir [25], two events, each in a distinct node, may occur at the same time. We say that a link $l$ is faulty in $E$ if $l$ experiences at least one loss event in $E$. In this chapter, we assume that at most $f$ links fail during any execution. Note that the faulty links may be different in different executions of the algorithm. If a link is not faulty, then it is reliable. We require delays on reliable links to be finite. In other words, messages sent on reliable links must eventually be delivered. Because of the asynchronous nature of the network, a node cannot distinguish between a slow link and a faulty link that loses messages. Therefore, the nodes cannot detect the faulty links.

In this chapter, we will construct a distributed election algorithm such that, at the end of every execution of the algorithm, a unique node is elected as a leader of the network. Also, all the nodes will know the identifier of the leader. We require that all the $n$ local programs that form the election algorithm be identical.

One of the efficiency measures that we will use is the maximum running time of an asynchronous algorithm. Although we assume that the link delays are arbitrary when we design the algorithm, we set the link delays to be at most one time unit when we compute the running time of the algorithm. For convenience, we also assume that all receive events take zero time units when we compute the running time. This assumption is reasonable in real systems where the nodes' processing time is negligible compared with link delays.

3.3 Informal Description of the Algorithm

We present the formal description of the algorithm in Section 3.7.
3.3.1 Definitions

Let \( n \) be the number of nodes, \( f \) be the maximum number of faulty links, and \( r \) be a design parameter. Consider a particular execution \( E \) of the algorithm. Each node tries to eliminate all the other nodes from the competition to be the leader. The unique node that survives all eliminations elects itself as the leader. If a node \( u \) is eliminated at some time \( t \), then we say that \( u \) is dead at \( t \). If \( u \) is dead at \( t \), then \( u \) is dead at all times \( t' \geq t \), and \( u \) does not try to eliminate any other node after \( t \). If \( u \) is not dead at \( t \), then \( u \) is live at \( t \). All nodes are live at the beginning of the algorithm. We assume that all the nodes start executing the algorithm simultaneously. (For the case when only a subset of the nodes starts executing the algorithm or when the nodes start executing the algorithm at different times, we can easily modify the algorithm by requiring sleeping nodes to be eliminated as soon as they receive any message.)

Each node \( u \) keeps an integer variable called \( \text{phase}(u) \). The value of \( \text{phase}(u) \) ranges from 0 to \( \left\lceil \frac{n+2}{2(r-1)f} \right\rceil +1 \). Intuitively, if \( u \) is live at some time \( t \), then the value of \( \text{phase}(u) \) at \( t \) is a lower bound on the number of nodes that \( u \) eliminated by time \( t \). A node, whether live or dead, cannot decrease its phase. A live node may increment its phase to reflect an increase in the number of nodes it has eliminated.

If \( u \) eliminates a node \( v \), then we say that \( u \) is a suppressor of \( v \) and that \( v \) is a victim of \( u \). Node \( v \) may have several suppressors during the execution \( E \). It is possible for a node to be its own suppressor. For every time \( t \), a live node at \( t \) does not have any suppressor at or before \( t \).

The algorithm maintains the following invariant, called the Algorithm Invariant: Node \( u \) becomes a suppressor of a node \( v \) at time \( t \) only if all previous suppressors of \( v \) are dead at \( t \). Thus each node \( v \) keeps a link pointer called \( \text{Suppressor-Link}(v) \) that points to the most recent suppressor of \( v \). Initially, \( \text{Suppressor-Link}(v) = \text{nil} \). A dead node \( v \) may increment \( \text{phase}(v) \) to give a lower bound on the phase of the most recent suppressor of \( v \).

When a node \( u \) decides that \( u \) is the leader, then \( u \) informs all the nodes. Each node \( v \) halts \( v \)'s execution of the algorithm when \( v \) learns of the existence of a leader.
3.3.2 The algorithm

Each node $u$ keeps a set called $Untrav(u)$ initially containing the names of the links incident on $u$. Before executing the algorithm, the phase of each node $u$ is 0. As soon as $u$ starts executing the algorithm, $u$ sets $\text{phase}(u)$ to 1. Each node $u$ repeats the following until $\text{phase}(u)$ becomes $\left\lceil \frac{n+2}{2(r-1)f} \right\rceil + 1$:

Suppose that, for some node $u$, $\text{phase}(u)$ becomes $i$ at some time $t$ and $u$ is live at $t$, where $1 \leq i \leq \left\lceil \frac{n+2}{2(r-1)f} \right\rceil$. If $i = 1$, then $u$ chooses $\lceil rf \rceil$ link names from $Untrav(u)$ and deletes them from $Untrav(u)$. If $i > 1$, then $u$ chooses $rf - f$ link names from $Untrav(u)$ and deletes them from $Untrav(u)$. Call the chosen links $\text{Chosen-Links}(u,i)$. Node $u$ sends the message "Eliminate-$(i,JD(u))$" on each link in $\text{Chosen-Links}(u,i)$. Call the multiset of these eliminate-messages $\text{New}(u,i)$. (For simplicity, we will use $rf$ instead of $\lceil rf \rceil$, and $(r-1)f$ instead of $\lceil rf \rceil - f$).

For each node $v \neq u$, if $v$ receives the message $M_1= \text{"Eliminate-} (i,JD(u)) \text{"}$ at some time $t_v$, then $v$ compares $(i,JD(u))$ with $(j,JD(v))$ lexicographically, where $j$ is the value of $\text{phase}(v)$ at $t_v$:

Case 1: $(i,JD(u)) < (j,JD(v))$: Node $v$ sends the message $M_2= \text{"Elimination-Unsuccessful-} (i,JD(u)) \text{"}$ on $l(v,u)$. If $u$ receives $M_2$ at some time $t_u$, then there are two cases:

Case 1.1: $u$ has a suppressor at $t_u$: Then $u$ discards $M_2$.

Case 1.2: $u$ has no suppressor at $t_u$: Since $u$ may have incremented $\text{phase}(u)$ after $u$ sent $M_1$, then $u$ compares $i$ with $k$, where $k$ is the value of $\text{phase}(u)$ at $t_u$. If $k = m$, then $u$ becomes eliminated at $t_u$, and $u$ sets $\text{Suppressor-Link}(u)$ to $l(u,u)$ (that is, to itself) to indicate that $u$ becomes its own suppressor. If $i < k$, then $M_2$ is an out-of-date message. Thus, $u$ sends the message "Eliminate-$(k,JD(u))$" on $l(u,v)$. We say that "Eliminate-$(k,JD(u))$" is a refresher for $M_1$.

Case 2: $(j,JD(v)) < (i,JD(u))$: Node $v$ sets $\text{phase}(v),JD(v))$ to $(i,JD(u))$. In addition, $v$ does one of the following:

Case 2.1: $v$ has no suppressor at $t_v$: Then $v$ becomes eliminated at $t_v$, and $u$ becomes $v$'s suppressor at $t_v$. Trivially, the Algorithm Invariant is preserved. Node $v$ sets $\text{Suppressor-Link}(v)$ to $l(v,u)$, and $v$ sends
the message "Elimination-Successful" on \( I(v,u) \).

**Case 2.2: v has a suppressor at \( t_w \):** Then \( \text{Suppressor-Link}(v) \neq \text{nil} \). Suppose that \( \text{Suppressor-Link}(v) = I(v,w) \), where \( w \) may be \( v \). Node \( v \) sends the message \( M_3 = \text{"Potential-Suppressor-}(\text{id}(u))\text{"} \) on \( I(v,w) \).

If \( w \) receives \( M_3 \) at some time \( t_w \), then \( w \) compares \( (\text{id}(u)) \) with \( (\text{id}(w)) \), where \( q \) is the value of \( \text{phase}(w) \) at \( t_w \):

**Case 2.2.1: \( (\text{id}(u)) < (\text{id}(w)) \):** Then \( w \) sends "Potential-Suppressor-Unsuccessful-(\text{id}(u))" on \( I(w,v) \). If \( v \) receives this message, then \( v \) sends "Elimination-Unsuccessful-(\text{id}(u))" on \( I(v,u) \).

**Case 2.2.2: \( (\text{id}(w)) \leq (\text{id}(u)) \):** Then \( w \) sends the message \( M_4 = \text{"Potential-Suppressor-Successful-(\text{id}(u))\text{"} \) on \( I(w,v) \). Also, \( w \) sets \( \text{Suppressor-Link}(w) \) to \( I(w,w) \) at time \( t_w \) if \( w \) has no suppressor at \( t_w \). Otherwise, if \( w \) has a suppressor at \( t_w \), then \( w \) leaves \( \text{Suppressor-Link}(w) \) unchanged. Node \( v \) may have received an eliminate-message from some node \( x \neq u \) after \( v \) sent \( M_3 \). Hence if \( v \) receives \( M_4 \), then \( v \) compares \( \text{id}(u) \) with \( \text{id}(v) \). If \( \text{id}(u) = \text{id}(v) \), then \( u \) becomes a suppressor of \( v \), and \( v \) sets \( \text{Suppressor-Link}(v) \) to \( I(v,u) \). Note that the Algorithm Invariant is maintained. Also, \( v \) sends "Elimination-Successful" on \( I(v,u) \). If \( \text{id}(u) \neq \text{id}(v) \), then \( v \) sends "Elimination-Unsuccessful-(\text{id}(u))" on \( I(v,u) \).

If \( u \) receives \((r-1)f\) "Elimination-Successful" messages, each on a distinct link, and if \( u \) is live, then \( u \) increments \( \text{phase}(u) \) by 1.

If \( \text{phase}(u) \) becomes \( \left\lceil \frac{n+2}{2(r-1)f} \right\rceil + 1 \) when \( u \) is live, then \( u \) elects itself as a leader. Node \( u \) then sends the message "ANNOUNCE-LEADER-(\text{id}(u))" to \( 2f+1 \) nodes and halts. All the nodes that receive this message send the message "LEADER-IS-(\text{id}(u))" on all the links incident on them and halt. When a node receives "LEADER-IS-(\text{f})", for some \( j \), then the node halts. Since there are at most \( f \) faulty links, at least one node \( v \) adjacent only to nonfaulty links receives "ANNOUNCE-LEADER-(\text{id}(u))." Thus, all the nodes in the network will know \( \text{id}(u) \) of the leader \( u \) and will halt.
3.4 Proof of Correctness

The strategy of our proof of correctness will be as follows. Lemma 4.5 shows that no two distinct nodes can be elected as leaders in the same execution. Lemma 4.9 shows that there exists at least one leader in each execution. Hence, Lemmas 4.5 and 4.9 imply that every execution produces one leader.

Consider any execution $E$. Let $t$ be a time, and $u$, $v$, and $w$ be distinct nodes. Node $u$ is a leader at $t$ if $u$ is live at $t$ and $\text{phase}(u)$ becomes $\left\lceil \frac{n+2}{2(r-1)f} \right\rceil +1$ at or before $t$. We say that $u$ eliminates $v$ at $t$ if $v$ sets $\text{Suppressor-Link}(v)$ to $l(v,u)$ at $t$. Define $\text{Suppressors}(v,t)$ as the empty sequence if $v$ is live at $t$. Otherwise, define $\text{Suppressors}(v,t)$ as the sequence of all nodes, other than $v$, that eliminate $v$ at time $0, 1, \ldots, t$: $u$ precedes $w$ in $\text{Suppressors}(v,t)$ if $u$ eliminates $v$ at some time $t_u$, $w$ eliminates $v$ at some time $t_w$, and $0 \leq t_u < t_w \leq t$. Note that, by the total ordering on $\text{Events}(v)$, $t_u \neq t_w$.

The following lemma shows that no node eliminates another node more than once in the same execution.

**Lemma 4.1:** All the nodes in $\text{Suppressors}(v,t)$ are distinct.

**Proof:** If $u$ is in $\text{Suppressors}(v,t)$, then, according to the algorithm, $v$ sends "Elimination-Successful" to $u$ on $l(v,u)$. Since $u$ sends a refresher message only if $u$ receives an out-of-date message, $u$ does not send any more eliminate-messages on $l(u,v)$.

We say that an eliminate-message $M$ that $v$ receives from $u$ is a fatal message if $M$ is the last eliminate-message that $u$ sends to $v$ before $u$ eliminates $v$. By Lemma 4.1, there is at most one fatal message for each pair of nodes.

**Lemma 4.2:** Suppose that $u$ precedes $w$ in $\text{Suppressors}(v,t)$. Suppose that $u$ eliminates $v$ at time $t_u \leq t$, and that $v$ receives the fatal message "Eliminate-$(iJD(w))$" from $w$ at some time $t_1$. Then, $t_u < t_1$.

**Proof:** Note that $t_u \neq t_1$ since $\text{Events}(v)$ is totally ordered. Suppose that, to the contrary, $t_1 < t_u$. If $\text{Suppressor-Link}(v) = \text{nil}$ at $t_1$, then according to the algorithm, $w$ is the first node in $\text{Suppressors}(v,t)$. This contradicts our hypothesis that $u$ precedes $w$ in $\text{Suppressors}(v,t)$.
If \( \text{Suppressor-Link}(v) \neq \text{nil} \) at \( t_1 \), then \( v \) sets \( \text{ID}(v) \) to \( \text{ID}(w) \) at \( t_1 \), and sends "Potential-Suppressor-(\( i JD(w) \))" on \( \text{Suppressor-Link}(v) \). Suppose that \( w \) eliminates \( v \) at some time \( t_2 \), where \( t_1 < t_2 \leq t \). Then, by the algorithm, \( v \) receives "Potential-Suppressor-Successful-(\( i JD(w) \))" at \( t_2 \). But since \( u \) precedes \( w \) in \( \text{Suppressors}(v,t) \), then \( t_u < t_2 \). Thus \( \text{ID}(v) \neq \text{ID}(w) \) at \( t_2 \) because \( t_1 < t_u \).

According to the algorithm, \( v \) cannot set \( \text{Suppressor-Link}(v) \) to \( l(v,w) \) at \( t_2 \), a contradiction.

Hence \( t_u < t_1 \). □

Lemma 4.3 shows that the Algorithm Invariant indeed holds.

Lemma 4.3: Suppose that \( u \) eliminates \( v \) at \( t \). Then, all the nodes except \( u \) in \( \text{Suppressors}(v,t) \) are dead at \( t \).

Proof: The proof proceeds by induction on the length of \( \text{Suppressors}(v,t) \), denoted \( |\text{Suppressors}(v,t)| \).

Basis: Suppose that \( |\text{Suppressors}(v,t)| = 1 \). Then \( \text{Suppressors}(v,t) \) consists only of \( u \), and the lemma is trivially true.

Inductive Step: Suppose that the lemma is true for \( |\text{Suppressors}(v,t)| \leq p-1 \) for some integer \( p \geq 2 \). Consider the case when \( |\text{Suppressors}(v,t)| = p \). Let \( \text{Suppressors}(v,t) = w_1, w_2, \ldots, w_{p-1}, u \). By Lemma 4.1, all the nodes in \( \text{Suppressors}(v,t) \) are distinct. Let the time when \( w_{p-1} \) eliminates \( v \) be \( t_w \). Since \( u \) eliminates \( v \) after \( w_{p-1} \) does, then \( t_w < t \). Node \( v \) receives the fatal message "Eliminate-(\( i JD(u) \))" from \( u \) at some time \( t_1 < t \), where \( i \) is some phase value. By Lemma 4.2, \( t_w < t_1 \). Thus \( v \) receives the fatal message from \( u \) after \( v \) sets \( \text{Suppressor-Link}(v) \) to \( l(v,w_{p-1}) \). Therefore, \( v \) receives \( M = "\text{Potential-Suppressor-Successful-(}i JD(u))" \) from \( w_{p-1} \) at \( t \). By the algorithm, \( w_{p-1} \) either eliminated itself or had a suppressor when \( w_{p-1} \) sent \( M \). In either case, \( w_{p-1} \) is dead at \( t \). By the inductive step, nodes \( w_1, \ldots, w_{p-2} \) are all dead at \( t_w \), and hence at \( t \). □

Suppose that \( u \) is live when \( u \) sets \( \text{phase}(u) \) to an integer \( i \) at \( t \). Then define \( \text{Victims}(u,i) \) as the set of all nodes that \( u \) eliminates before or at \( t \). If \( u \) never sets \( \text{phase}(u) \) to an integer \( j \), or if \( u \) is dead when \( u \) sets \( \text{phase}(u) \) to \( j \), then we define \( \text{Victims}(u,j) \) as \( \emptyset \).

Lemma 4.4: For every phase value \( i \) and nodes \( u \) and \( w \), \( \text{Victims}(u,i) \cap \text{Victims}(w,i) = \emptyset \).
Proof: Suppose that, contrary to the lemma, there exist some \( u, v, w, \) and \( i \), such that
\[ v \in \text{Victims}(u,i) \cap \text{Victims}(w,i). \]
Let \( t_u \) be the time at which \( u \) sets \( \text{phase}(u) \) to \( i \), \( t_w \) be the time at which \( w \) sets \( \text{phase}(w) \) to \( i \), \( t_{uw} \) be the time at which \( u \) eliminates \( v \), and \( t_{vw} \) be the time at which \( w \) eliminates \( v \).
Without loss of generality, let \( t_u \leq t_w \). Since \( v \in \text{Victims}(u,i) \) and \( v \in \text{Victims}(w,i) \), then \( t_{uw} \leq t_u \) and \( t_{vw} \leq t_w \).
By the total ordering on \( \text{Events}(v) \), \( t_{uw} \neq t_{vw} \). There are three cases, depending on the relationships among \( t_{uw}, t_{vw}, \) and \( t_u \):

**Case 1:** \( t_{uw} < t_{vw} \): By Lemma 4.3, \( w \) is dead at \( t_{vw} \). Since \( t_{uw} < t_{uw} \leq t_u \), then \( w \) is dead at \( t_u \). Thus \( \text{Victims}(w,i) = \emptyset \), a contradiction.

**Case 2:** \( t_u < t_{uw} \leq t_{vw} \): By Lemma 4.3, \( u \) is dead at \( t_u \). Hence, \( \text{Victims}(u,i) = \emptyset \), a contradiction.

**Case 3:** \( t_{uw} \leq t_u < t_{vw} \): By Lemma 4.3, \( u \) is dead at \( t_{uw} \). Since \( \text{Victims}(u,i) \neq \emptyset \), then \( u \) is live at \( t_u \).

Therefore, \( \text{Suppressor-Link}(v) \) at \( t_u \) is \( I(v,u) \). Suppose that \( t' \) is the time at which \( v \) received the fatal message \( M \) from \( w \). By the total ordering on \( \text{Events}(v) \) and Lemma 4.2, \( t' < t_{uw} \leq t_w \). Since \( v \) sets \( \text{Suppressor-Link}(v) \) to \( I(v,w) \) at \( t_{uw} > t_u \), and \( u \) is live at \( t_u \), then \( \text{phase}(w) \) in \( M \) must be at least equal to \( i \).

Thus \( \text{phase}(w) \) is at least \( i \) at \( t' \). This contradicts the fact that \( w \) sets \( \text{phase}(w) \) to \( i \) at \( t_w \).

Thus all three cases lead to contradictions. \( \square \)

**Lemma 4.5:** For every execution \( E \), there are no two leaders in \( E \).

Proof: Suppose that, to the contrary, there are two leaders \( u \) and \( w \) in some \( E \). Let \( i = \lceil \frac{n+2}{2(r-1)f} \rceil + 1 \).

Then, \( \text{phase}(u) \) and \( \text{phase}(w) \) become \( i \) in \( E \). Thus
\[ |\text{Victims}(u,i)|, |\text{Victims}(w,i)| \geq \left(\frac{n+2}{2(r-1)f}\right)(r-1)f = \frac{n+2}{2} \text{ nodes}. \]

Since there are \( n \) nodes, there is at least one node \( v \) in both \( \text{Victims}(u,i) \) and \( \text{Victims}(w,i) \). This contradicts Lemma 4.4. \( \square \)

Let \( M \) be an eliminate-message that \( u \) sends on \( I(u,v) \). We define \( \text{Path}(M) \) as follows: If \( M \) is lost on \( I(u,v) \), or if \( \text{Suppressor-Link}(v) = \text{nil} \) when \( v \) receives \( M \), then \( \text{Path}(M) = \{I(u,v)\} \). Otherwise,
\[ \text{Path}(M) = \{I(u,v), I(v,w)\}, \]
where \( I(v,w) = \text{Suppressor-Link}(v) \) when \( v \) receives \( M \). Recall that \( \text{New}(u,i) \) is the multiset of the eliminate-messages that a live node \( u \) sends on each link in \( \text{Chosen-Links}(u,i) \) when \( \text{phase}(u) \) becomes \( i \). We say that \( M \) is new if \( M \in \text{New}(u,i) \), for some \( i \).
The following lemma shows that the paths of any two distinct new messages sent by a node are disjoint.

**Lemma 4.6:** Let $M$ and $M'$ be two distinct new messages sent by $u$. Then, $\text{Path}(M) \cap \text{Path}(M') = \emptyset$.

**Proof:** When $u$ sends a new message on a link $l$, $u$ deletes $l$ from $\text{Untrav}(u)$. Hence, $u$ sends $M$ on some $l(u,v)$, while $u$ sends $M'$ on some $l(u,w)$, where $v \neq w$. Clearly, $\text{Suppressor-Link}(v) \neq \text{Suppressor-Link}(w)$ unless both suppressor-links are nil. □

If an event $e$ in $\text{Events}(u)$ occurs when $\text{phase}(u) = i$, then we say that $e$ occurs during phase $i$. Let $M_1$ be an eliminate-message sent by $u$ on some $l(u,v)$ during phase $i$. We say that a message $M_2$ is a successful reply for $M_1$ if $M_2 = "\text{Elimination-Successful}"$ and if $v$ sent $M_2$ on $l(v,u)$ in response to $M_1$. We say that $M_2$ is an unsuccessful reply for $M_1$ if $M_2 = "\text{Elimination-Unsuccessful-(iID(u))}"$ and if $v$ sent $M_2$ on $l(v,u)$ in response to $M_1$. We say that $M_2$ is a reply for $M_1$ if $M_2$ is a successful or unsuccessful reply for $M_1$.

According to the algorithm, an eliminate-message can be either new or a refresher for another eliminate-message. By Lemma 4.6, if $u$ sends two eliminate-messages on the same link, then at least one of the messages is a refresher.

**Lemma 4.7:** Suppose that $u$ sends $m \geq f$ eliminate-messages in the execution. If no leader is elected in $E$, then $u$ receives at least $m - f$ replies for these messages.

**Proof:** Because there is no leader in $E$, every node continues to process messages. Thus a node $u$ receives no reply for an eliminate-message $M$ only if $\text{Path}(M)$ contains a faulty link. Let $S(u)$ be the multiset of the $m$ eliminate-messages sent by $u$ in the entire execution. Some of the eliminate-messages in $S(u)$ may be refresher messages. Construct the multiset $S'(u)$ from $S(u)$ as follows: An eliminate-message $M \in S(u)$ sent to a node $v$ is in $S'(u)$ if and only if $M$ is the last eliminate-message in $S(u)$ that was sent to $v$. Note that if $M$ is in $S(u) - S'(u)$, then $u$ must have sent a refresher message for $M$, and thus $u$ received a reply for $M$. 

Let $|S'(u)| = m'$. If $m' \leq f$, then $u$ received replies for at least $m - m' \geq m - f$ eliminate-messages, and the lemma is true. If $f < m'$, then let $M_1$ and $M_2$ be any two distinct messages in $S'(u)$. Suppose that $u$ sent $M_1$ to some node $v$. By the definition of $S'(u)$, $u$ must have sent $M_2$ to some node $w \neq v$. Thus, by the definition of $\text{Path}(M_1)$ and $\text{Path}(M_2)$, $\text{Path}(M_1) \cap \text{Path}(M_2) = \emptyset$. Hence at most $f$ messages $M$ in $S'(u)$ have a faulty link in $\text{Path}(M)$, and at least $m' - f$ replies are received for messages in $S'(u)$. The total number of replies that $u$ receives is at least $(m - m') + (m' - f) = m - f$. \hfill \square

Suppose that $u$ is live when $u$ sets $\text{phase}(u)$ to $i$ at some time $t$, for some $2 \leq i \leq \left\lceil \frac{n+2}{2(r-1)f} \right\rceil$. Then define $\text{Old}(u,i)$ as the multiset of the eliminate-messages that $u$ sent and for which $u$ did not receive a reply by $t$. Define $\text{Old}(u,1)$ as $\emptyset$. Recall that, if $u$ is live, then $u$ sends a refresher message as soon as $u$ receives an out-of-date message.

Lemma 4.8: Let $u$ be live when $u$ sets $\text{phase}(u)$ to $i \geq 2$. Then $|\text{Old}(u,i)| = f$. Hence $u$ receives at most $f$ out-of-date messages during phase $i$.

Proof: The proof proceeds by induction on $i$.

Basis: Suppose that $i = 2$. Node $u$ sent $rf$ new messages during phase 1 and received $(r-1)f$ successful replies for them. Hence the lemma is true.

Inductive Step: Assume that the lemma holds for $i = p-1$, for some $3 \leq p \leq \left\lceil \frac{n+2}{2(r-1)f} \right\rceil$. Suppose that $i = p$. During phase $p-1$, $u$ sent $(r-1)f$ new messages and received replies for $k$ messages in $\text{Old}(u,p-1)$, where $0 \leq k \leq f$. Suppose that, during phase $p-1$, $u$ received successful replies for $k_1$ messages in $\text{Old}(u,p-1)$, where $0 \leq k_1 \leq k$. Then $u$ received unsuccessful replies for $k-k_1$ messages in $\text{Old}(u,p-1)$ during phase $p-1$. Since $u$ is live in phase $p$, all these unsuccessful replies were of the form "Elimination-Unsuccessful-(j,ID(u))", where $j < p-1$. Thus all of the unsuccessful replies that $u$ received in phase $p-1$ were out-of-date messages. Hence $u$ sent $k-k_1$ refresher messages during phase $p-1$. Thus $u$ sent $(r-1)f+k-k_1$ eliminate-messages during phase $p-1$. Since $u$ advanced to phase $p$, $u$ must have received successful replies for $(r-1)f-k_1$ eliminate-messages sent during phase $p-1$. Hence $\text{Old}(u,p)$ is equal to the union of (the multiset of eliminate-messages in $\text{Old}(u,p-1)$) for which $u$ did not
receive replies during phase $p-1$ and [the multiset of the eliminate-messages sent during phase $p-1$ for which $u$ did not receive replies during phase $p-1$]. Hence,

$$|\text{Old}(u,p)| = [f-k] + [(r-1)f-k - ((r-1)f-k)] = f.$$ \hfill \Box

**Lemma 4.9:** There exists at least one leader in the execution $E$.

**Proof:** Suppose that, to the contrary, no leader exists. Thus, for every node $v$, $\text{phase}(v) < \left\lfloor \frac{n+2}{2(r-1)f} \right\rfloor + 1$. Since no node can decrease its phase, there exists some time $t_f$ such that no node changes its phase after $t_f$. At $t_f$, let $(i,j)$ be the highest (phase,ID)-pair among the nodes. Let $u$ be the unique node whose identifier was $j$ at the start of the algorithm. Then $\text{phase}(u) = i$ at $t_f$. To see this, let $v$ be any node whose (phase,ID)-pair is $(i,j)$ at $t_f$. If $v = u$, then $\text{phase}(u) = i$ at $t_f$. Suppose $v \neq u$. Node $v$ is dead at $t_f$ since $\text{ID}(v) = j$ at $t_f$. Thus, $v$ must have received "Eliminate-$(i,j)$" at or before $t_f$. By the algorithm, only $u$ can send "Eliminate-$(i,j)$." Hence, by the choice of $(i,j)$, and since no node can decrease its phase, $\text{phase}(u)$ is $i$ at $t_f$. Note also that $u$ is live at $t_f$. To see this, let $t_u$ be the time when $u$ sets $\text{phase}(u)$ to $i$ for the first time, where $t_u \leq t_f$. At $t_u$, $u$ is either dead or live. Since $u$ changes $\text{phase}(u)$ to $i$ at $t_u$, then if $u$ is dead at $t_u$, then $u$ must change $\text{ID}(u)$ according to the algorithm. Thus $\text{ID}(u) \neq j$ at $t_u$. By the algorithm, $u$ never sends eliminate-messages after $u$ is dead. Thus $u$ never sends "Eliminate-$(i,j)$." But this means that $(i,j)$ is not a (phase,ID)-pair of any node, a contradiction. Therefore, $u$ is live at $t_u$. By the choice of $(i,j)$, and since the (phase,ID)-pair of any node can not decrease, then no node can eliminate $u$ after $t_u$. Thus $u$ is live at $t_f$.

To prove the lemma, there are two cases, depending on the value of $i$:

**Case 1:** $i \geq 2$. Suppose that $u$ sets $\text{phase}(u)$ to $i$ at some time $t_u \leq t_f$. After $t_u$, $u$ sends $(r-1)f$ new messages. Because there is no leader in $E$, $u$ must receive replies for all the messages $M$ in $\text{Old}(u,i)$ and in $\text{New}(u,i)$, where $\text{Path}(M)$ contains no faulty links. If $M$ is in $\text{New}(u,i)$, then the reply for $M$ is a successful reply since $u$ is live at $t_f$. If $M$ is in $\text{Old}(u,i)$, then the reply for $M$ can be an out-of-date message. Since $E$ has no leader, $u$ has not halted, and $u$ must send a refresher message for $M$. By Lemma 4.7, at most $f$ messages in $\text{New}(u,i) \cup \text{Old}(u,i)$ are lost. Hence, $u$ receives at least
(r-1)f+ | Old(u,i)|-f "Elimination-Successful" messages during phase i, all with nonfaulty paths.

Hence, by Lemma 4.8, u receives at least (r-1)f successful reply messages during phase i. Thus u must increment phase(u) by 1 some time after t_u. This contradicts our assumption that phase(u) = i at t_f.

Case 2: i = 1. This case is very similar to the previous case, except that u sends rf new messages during phase 1, and that |Old(u,1)|=0. This case also leads to a contradiction.

Theorem 4.1: For every execution E of the algorithm, there exists a unique leader. Furthermore, every node in the network knows the ID of the leader.

Proof: By Lemmas 4.5 and 4.9, there exists a unique leader u in E. Furthermore, u sends

"ANNOUNCE-LEADER-(ID(u))" to 2f+1 nodes. Since there are at most f ≤ \[ \frac{n-3}{2} \] faulty links, there exists a node v that is not adjacent to any faulty links and that receives "ANNOUNCE-LEADER-(ID(u))." According to the algorithm, v sends ID(u) to all adjacent nodes. Thus all the nodes will know the ID of the leader.

3.5 Message Complexity

The following lemma specifies the maximum number of live nodes that reach phase i.

Lemma 5.1: Let i be an integer such that 2 ≤ i ≤ \[ \frac{n+2}{2(r-1)f} \] + 1. For every i, there are at most \[ \frac{n}{(i-1)(r-1)f} \] nodes u such that u is live when u sets phase(u) to i.

Proof: Suppose that there are at most k nodes u such that u is live when u sets phase(u) to i. Trivially, k ≤ n. Let u_j denote the j-th live node to set its phase to i, for every 1 ≤ j ≤ k. By induction on i, we can show that u_j must have eliminated at least (i-1)(r-1)f nodes to reach phase i. By Lemma 4.4,

Victims(u_j,i)\Victims(u_p,i)=\emptyset for every 1 ≤ j \neq p ≤ k. Hence k ≤ \[ \frac{n}{(i-1)(r-1)f} \].

Theorem 5.1: The algorithm uses \( O( nrf^+ \frac{nr}{(r-1)} \log(\frac{n}{(r-1)f}) ) \) messages in the worst case, where n is the number of nodes in the network, and f is the maximum number of faulty links.

Proof: Let \( i < \frac{n+2}{2(r-1)f} \) + 1 be an integer. If a node u is live when it sets phase(u) to i at some time t.
then \( u \) sends \((r-1)f\) new messages during phase \( i \). By Lemma 4.8, \( u \) sends at most \( f \) refresher messages during phase \( i \). Hence \( u \) sends at most \( rf \) eliminate-messages during phase \( i \). Each eliminate-message generates at most three additional messages as follows:

1. Suppose that \( u \) sends an eliminate-message to a node \( v \).
2. Node \( v \) sends "Potential-Suppressor-(i,ID(u))" on \( Suppressor-Link(v) \).
3. Node \( v \) receives "Potential-Suppressor-Successful-(i,ID(u))" or "Potential-Suppressor-Unsuccessful-(i,ID(u))" on \( Suppressor-Link(v) \).
4. Finally, \( v \) sends a reply to \( u \).

Thus \( u \) causes at most \( 4rf \) messages to be generated while \( phase(u) \) is \( i \). The total number of messages the algorithm uses is the sum of the number of messages generated to elect a leader plus the number of messages used to inform all the nodes of the ID of the leader. By Lemma 5.1, the number NUM of the messages generated to elect a leader is as follows:

\[
NUM \leq \sum_{i=1}^{\left\lfloor \frac{n+2}{2(r-1)f} \right\rfloor} (\text{num. live nodes } u \text{ that reach phase } i)(\text{num. messages generated by } u \in i)
\]

\[
= \sum_{i=1}^{\left\lfloor \frac{n+2}{2(r-1)f} \right\rfloor} \frac{n}{(i-1)(r-1)f} \cdot 4rf + 4nrf < \frac{4nr}{(r-1)} \sum_{i=1}^{n+2} \frac{1}{i} + 4nrf
\]

\[
NUM = O\left(\frac{nr}{(r-1)}\log\left(\frac{n}{(r-1)f}\right) + nrf\right)
\]

The algorithm uses \((2f+1)n = O(nf)\) messages to inform all the nodes of the ID of the leader.

A detailed analysis, omitted here, shows that the value of \( r \) that minimizes the total number of messages is \( r = \min(1 + C\left(\frac{\ln(n+2)}{f}\right)^{1/2}, \frac{n-4}{2f}) \), where \( C = 1 + O\left(\frac{\log f}{\log n}\right) \). For every value of \( n \) and \( f \) subject to \( f \leq \left\lfloor \frac{n}{2} \right\rfloor \), \( \frac{7}{8} \leq C \leq 1 \). Thus the minimum number of messages that our algorithm uses is \( O(nf + n\log n) \) messages.
3.6 Time Complexity

**Theorem 6.1:** The algorithm takes at most $O\left(\frac{n}{(r-1)f}\right)$ time units to complete, where $n$ is the number of nodes, and $f$ is the maximum number of faulty links.

**Proof:** The maximum running time of the algorithm is the maximum time spent to elect a leader in any execution. Consider any execution $E$, and let $u$ be the leader when $E$ terminates. Assume the delay on each reliable link to be at most one time unit. Let $M$ be any eliminate-message that $u$ sends. As in the proof of Theorem 5.1, $M$ generates at most three additional messages. Each message reaches its destination in at most one time unit. Hence $u$ receives a reply for $M$ in at most 4 time units. Thus, by induction on $i$, $u$ spends at most $4(i-1)$ time units to reach phase $i$. In particular, $u$ spends at most $4\left(\frac{n+2}{2(r-1)f}\right)$ time units to elect itself as the leader. By the algorithm, all the nodes will know the ID of the leader in at most two more time units. Since all the nodes start executing the algorithm simultaneously, the algorithm will terminate in $\leq 4\left(\frac{n+2}{2(r-1)f}\right) + 2 = O\left(\frac{n}{(r-1)f}\right)$ time units. $\square$

3.7 Formal Description of the Algorithm

We assume in what follows that the node ID's are integers. The algorithm for each node $u$ uses the following variables and data structures:

* $LINK(u)$ is the set of the names of all links incident on $u$.
* $UNTRAV(u)$ is a set of link names. Initially, $UNTRAV(u) = LINK(u)$. (Link $l \in UNTRAV(u)$ iff $l$ is incident on $u$ and $u$ has not sent any eliminate-message on $l$.)
* $phase(u)$ is an integer variable. Initially, $phase(u) = 0$.
* $dead(u)$ is a Boolean variable. Initially, $dead(u)$ is false.
* $num_of_victims(u)$ is an integer variable. Initially, $num_of_victims(u) = 0$. (This variable is a lower bound on the number of nodes that $u$ eliminated during the current phase.)
* \textit{SUPPRESSOR\_LINK}(u) is a pointer to a link. Initially, \textit{SUPPRESSOR\_LINK}(u) = l(u,u).

* \textit{POTENTIAL\_SUPPRESSOR\_ARRAY\_u} is an integer array indexed by the names of the links incident on \( u \). Initially, \textit{POTENTIAL\_SUPPRESSOR\_ARRAY\_u}[l] = \text{nil} for each \( l \) in \textit{LINK}(u). (Intuitively, \textit{POTENTIAL\_SUPPRESSOR\_ARRAY\_u} points to the nodes that are potential suppressors of \( u \).)

* \textit{leader}(u) is an integer variable. Initially, \textit{leader}(u) = \text{nil}.

The algorithm for each \( u \) is as follows:

(The comments refer to the cases in Section 3.3.2.)

\begin{verbatim}
begin
Set \text{phase}(u) to 1;
Choose \( rf \) links from \textit{UNTRAV}(u);
Call the chosen links \( e_1, e_2, \ldots, e_rf \);
\textit{UNTRAV}(u) := \textit{UNTRAV}(u) - \{ e_1, e_2, \ldots, e_rf \};
Send "\text{ELIMINATE}(1, ID(u))" on each \( e_i \), where \( i = 1, 2, \ldots, rf \);
while \text{phase}(u) < \left\lceil \frac{n+2}{2(r-1)f} \right\rceil + 1 or \text{dead}(u)
begin
Receive some message \( M \) on some link \( l \);
case \( M \) of:
(1) \( M \) is "\text{ELIMINATE}(k, j)" and \textit{SUPPRESSOR\_LINK}(u) = \text{nil}:
    /* Case 1 */
    if \((k, j) < (\text{phase}(u), ID(u))\) then Send "\text{ELIMINATION\_UNSUCCESSFUL}(k, j)" on \( l \);
\end{verbatim}
/* Case 2.1 */

if (k, j) > (phase(u), ID(u)) then

begin
    SUPPRESSOR_LINK(u) := l;
    dead(u) := true;
    (phase(u), ID(u)) := (k, j);
    Send "ELIMINATION_SUCCESSFUL" on l
end if;

(2) M is "ELIMINATION_UNSUCCESSFUL_(k,j)" and SUPPRESSOR_LINK(u) ≠ nil:

/* Case 1 */

if (k, j) < (phase(u), ID(u)) then Send "ELIMINATION_UNSUCCESSFUL_(k,j)" on l;

/* Case 2.2 */

if (k, j) > (phase(u), ID(u)) then

begin
    POTENTIAL_SUPPRESSOR ARRAY u[l] := j;
    (phase(u), ID(u)) := (k, j);
    Send "POTENTIAL_SUPPRESSOR_(k,j)" on SUPPRESSOR_LINK(u)
end if;

/* Case 1.2 */

(3) M is "ELIMINATION_UNSUCCESSFUL_(k,j)" and not dead(u):

if phase(u) = k then dead(u) := true;

if phase(u) ≠ k then Send "ELIMINATION_(phase(u),ID(u))" on l;

(4) M is "ELIMINATION_SUCCESSFUL" and not dead(u):

num_of_victims(u) := num_of_victims(u) + 1;

/* increment phase(u) */

if num_of_victims(u) = (r-1)f then

begin
    phase(u) := phase(u) + 1;
    if phase(u) < \( \left\lfloor \frac{n+2}{2(r-1)f} \right\rfloor + 1 \)
then begin
Set \textit{num\_of\_victims}(u) to 0;
Choose $(r-1)$ links from UNTRAV($u$);
Call these links $e_1, e_2, \ldots, e_{r-1}$;
UNTRAV($u$) := UNTRAV($u$) \setminus \{e_1, \ldots, e_{r-1}\};
Send "ELIMINATE(\text{phase}(u), \text{ID}(u))" on each $e_i$;
end if;

(5) $M$ is "\textit{POTENTIAL\_SUPPRESSOR}$_{(k,j)}$":

/\* Case 2.2.1 *\/

if $(k, j) < (\text{phase}(u), \text{ID}(u))$ then Send "\textit{POTENTIAL\_SUPPRESSOR\_UNSUCCESSFUL}$_{(k,j)}$" on $l$;

/\* Case 2.2.2 *\/

if $(k, j) \geq (\text{phase}(u), \text{ID}(u))$ then
  begin
  \textit{dead}(u) := true;
  Send "\textit{POTENTIAL\_SUPPRESSOR\_SUCCESSFUL}$_{(k,j)}$" on $l$
  end if;

/\* Case 2.2.1 *\/

(6) $M$ is "\textit{POTENTIAL\_SUPPRESSOR\_UNSUCCESSFUL}$_{(k,j)}$":

Find the link $l'$ such that \textit{POTENTIAL\_SUPPRESSOR\_ARRAY}$_u[l'] = j$;

Send "\textit{ELIMINATION\_UNSUCCESSFUL}$_{(k,j)}$" on $l'$;

/\* Case 2.2.2 *\/

(7) $M$ is "\textit{POTENTIAL\_SUPPRESSOR\_SUCCESSFUL}$_{(k,j)}$":

Find the link $l'$ such that \textit{POTENTIAL\_SUPPRESSOR\_ARRAY}$_u[l'] = j$;

if $j \neq \text{ID}(u)$ then Send "\textit{ELIMINATION\_UNSUCCESSFUL}$_{(k,j)}$" on $l'$;

if $j = \text{ID}(u)$ then
  begin
  Send "\textit{ELIMINATION\_SUCCESSFUL}" on $l'$;
SUPPRESSOR_LINK(u) := l' 
end if;

/* Case 1.1 */

(8) \( M \) is "ELIMINATION_SUCCESSFUL" or "ELIMINATION_UNSUCCESSFUL_{k,j}," 
and \( \text{dead}(u) \):

Discard \( M \);

(9) \( M \) is "ANNOUNCE_LEADER\(_{j}\)":

\( \text{dead}(u) := \text{true}; \)

Send "LEADER\(_{IS}_{j}\)" on each link in LINK\(_{u}\);

\( \text{leader}(u) := j \)

halt algorithm.

(10) \( M \) is "LEADER\(_{IS}_{j}\)":

\( \text{dead}(u) := \text{true}; \)

\( \text{leader}(u) := j \)

halt algorithm.

end while;

/* \( u \) elects itself as a leader */

Choose 2\( f+1 \) links from LINK\(_u\);

Send "ANNOUNCE_LEADER\((\text{ID}(u))\)" on each chosen link;

\( \text{leader}(u) := \text{ID}(u) \)

halt algorithm. \( \square \)
CHAPTER 4
UPPER AND LOWER BOUNDS
FOR ELECTION IN SYNCHRONOUS SQUARE MESHES

4.1 Introduction

Many election algorithms for various synchronous networks were proposed in the literature. Frederickson and Lynch [22] showed that $\Omega(n \log n)$ messages are necessary for election in synchronous rings when the election algorithm is required to being a comparison algorithm, i.e., when the election algorithm uses only comparisons of processor identifiers. On the other hand, the comparison algorithm of Loui, Matsushita, and West [35], originally designed for asynchronous complete networks, uses less than $4n$ messages in the worst case on synchronous complete networks. A natural question arises, therefore, about whether there exists a comparison algorithm on a bounded degree network using $O(n)$ messages. Peterson [41] answered this question in the affirmative when he designed a comparison algorithm for asynchronous square meshes that uses about $90n$ messages in the worst case. When run on synchronous square meshes, the message complexity of Peterson's algorithm becomes $32n$. Although Peterson's algorithm is of theoretical importance, it is not practical because of the large constants in the message complexity. In this chapter we present a comparison algorithm for square meshes that uses at most $\frac{229}{18} n$ messages, runs in time $O(\sqrt{n})$ time units, and requires $O(\log |T|)$ bits per message, where $n$ is the number of processors in the mesh, and $|T|$ is the cardinality of the set of processor identifiers. Also, we prove that any comparison algorithm on synchronous meshes requires at least $\frac{57}{32} n$ messages. The lower bound holds a fortiori for asynchronous meshes.

4.2 Model

We define an $n$-mesh as a square of $n$ processors, with $\sqrt{n}$ processors on each side, but with each column forming a ring and each row forming a ring. (Hwang and Briggs [29] call our mesh a near-neighbor mesh.) Figure 4.1 shows a 9-mesh. Two processors in an $n$-mesh communicate by sending messages to each other either on the link joining them or via other processors. We assume a synchronous
Figure 4.1. A 9-mesh

mode of communication. This implies that there is an upper limit on link delays before messages are delivered. We take the delay on each link to be at most one time unit.

We assume that all the processors in the $n$-mesh are identical except that each processor $p$ has a unique identifier $ID(p)$ chosen from a totally ordered set. Initially, no processor knows the identifier of any other processor. We assume that the processors know that the network is a square mesh, but they do not know the value of $n$. We also assume that the processors have a sense of direction [45]. Informally, sense of direction means that the processors have a uniform notion of which of their four links is the east link, the south link, the west link, and the north link. Thus, for example, if processor $p$ sends a message $M$ on $p$'s east link, then processor $q$ that receives $M$ knows that $q$ received $M$ from the west.

A distributed algorithm for an $n$-mesh is a set of $n$ identical programs; each program is assigned to a processor. We model each processor $p$ as an automaton. An election algorithm specifies the following
for $p$:

(1) the set of states $S_p$. The initial state of $p$ is $p$'s identifier $ID(p)$.

(2) a subset $L_p$ of $S_p$ called the set of elected states. If $p$ is in a state in $L_p$, then $p$ is elected as a leader.

(3) a message generation function that maps each state $s$ of $p$ to a 4-tuple $(M_E, M_S, M_W, M_N)$, where each element of the 4-tuple is either a message or the atom nil. Let $p_E$ (respectively $p_S$, $p_W$, $p_N$) denote $p$'s east (respectively south, west, north) neighbor. If $M_E$ (respectively $M_S$, $M_W$, $M_N$) is not nil, then $M_E$ (respectively $M_S$, $M_W$, $M_N$) represents the message that $p$ sends to $p_E$ (respectively $p_S$, $p_W$, $p_N$) when $p$ is in state $s$. If $M_E$ (respectively $M_S$, $M_W$, $M_N$) is nil, then $p$ does not send a message to $p_E$ (respectively $p_S$, $p_W$, $p_N$) when $p$ is in state $s$.

(4) a transition function that maps the 4-tuple $(M'_E, M'_S, M'_W, M'_N)$ and $p$'s current state $s$ to $p$'s next state, where each element of the 4-tuple is either a message or the atom nil. If $M'_E$ (respectively $M'_S$, $M'_W$, $M'_N$) is nil, then $p$ did not receive a message from $p_E$ (respectively $p_S$, $p_W$, $p_N$) when $p$ is in state $s$.

We assume that if $p$'s current state is in $L_p$, then $p$'s next state is in $L_p$. In other words, $L_p$ is a closed set.

In this chapter, the $n$-mesh is synchronous. Thus election algorithms on the mesh proceed in rounds. In each round, each processor $p$ sends some messages according to the message generation function and $p$'s current state, receives the messages sent to $p$ in the current round, does some internal computations, and changes state according to the transition function.

We wish the processors to execute an election algorithm so that a unique processor is chosen as the leader of the network when the algorithm terminates.

4.3 Upper Bounds

We now present an algorithm for election in synchronous $n$-meshes. All the processors start the algorithm simultaneously.
4.3.1 Overview of the algorithm

The algorithm proceeds in phases. Each processor tries to eliminate all other processors from the competition to be the leader. The processor with the largest identifier will be elected as the leader. We say that processor $u$ is live at the start of phase $i$ if $u$ has not been eliminated before the start of phase $i$. If $u$ is not live, then $u$ is dead. A dead processor does not initiate any messages, although it may relay messages. Each processor $u$ keeps a variable called $\text{largest\text{-}\text{seen}}(u)$ that contains the largest identifier that $u$ is aware of. Initially, $\text{largest\text{-}\text{seen}}(u) = \text{ID}(u)$.

Before we give the details of the algorithm, we present the intuition behind it. Define a $q$-square of processors to be a square mesh of processors with $q+1$ processors on each side but without wrap-around connections. As in Figure 4.2, let $SE(u,q)$ denote the $q$-square with processor $u$ at the northwest corner. When there is no confusion, we let $SE(u,q)$ denote also the set of all processors that are contained in or on the boundary of square $SE(u,q)$. Let $SW(u,q)$ denote the $q$-square with processor $u$ at the northeast corner. Define $NW(u,q)$ and $NE(u,q)$ similarly. The algorithm uses four kinds of messages. We will explain their use as we explain the algorithm. The messages are

(1) Eliminate messages of the form "Eliminate_$(ID_1,k,ID_2)$," where $ID_1$ is the identifier of the processor that initiates the message, $k$ is the number of links that the message will traverse in the current direction, and $ID_2$ is an identifier of another processor;

(2) Kill messages of the form "Kill_$(ID)$," where $ID$ is an identifier of a processor;

(3) Mark messages of the form "Mark_$(ID,k)$," where $ID$ is an identifier of a processor, and $k$ is the number of links that the message will traverse in the current direction;

(4) Final messages of the form "Final_$(ID)$," where $ID$ is the identifier of the processor that initiates the message.

For the rest of the chapter, we use the letter $E$ to denote Eliminate messages, $K$ to denote Kill messages, $M$ to denote Mark messages, and $F$ to denote Final messages. We let $E_u$ denote any message of the form "Eliminate_$(ID(u),k,ID')$," where $ID(u)$ is the identifier of the processor that initiates the message. We
Figure 4.2. The Definitions of NW(u,q), NE(u,q), SE(u,q), and SW(u,q)

also let $M_u$ denote any message of the form "Mark_[$ID(u),k]$," and $F_u$ denote any message of the form "Final_[$ID(u)$]." The algorithm has four main ideas.

Let $\alpha > 1$ be a parameter to be optimized later. The first idea is that, at the start of each phase $i$, each live processor $u$ sends an Eliminate message $E_u$ to traverse clockwise the boundary of $SE(u, \alpha^i)$. If $E_u$ completes the traversal of the boundary of $SE(u, \alpha^i)$, and if $u$ is live when $u$ receives $E_u$, then $u$ advances to phase $i+1$. Suppose that $v$ is a processor on the boundary of $SE(u, \alpha^i)$, and that $v$ receives $E_u$ which contains $ID(u)$. If $\text{largest-seen}(v) \leq ID(u)$, then $v$ is eliminated, $\text{largest-seen}(v)$ becomes $ID(u)$, and $E_u$ continues the traversal of the boundary of $SE(u, \alpha^i)$. If $\text{largest-seen}(v) > ID(u)$, then $v$
discards $E_u$. Since the network is synchronous, if $u$ does not receive $E_u$ after $4\alpha'$ time units, then $u$ knows that $E_u$ was discarded, and $u$ becomes eliminated.

We can show that the first idea guarantees that if $u$ advances to phase $i+1$, then all the processors in $SE(u, \alpha') \cup NW(u, \alpha') - \{u\}$ are eliminated by the time phase $i+1$ starts. Our second and third ideas will guarantee that all processors in $SW(u, \alpha') \cup NE(u, \alpha') - \{u\}$ are also eliminated by the time phase $i+1$ starts. Before we present our second idea, observe that if $w$ is a processor in $SW(u, \alpha')$, then $SE(w, \alpha') \cap SE(u, \alpha')$ contains a processor $v_w$ on the west boundary of $SE(u, \alpha')$ and a processor $s_w$ on the south boundary of $SE(u, \alpha')$, where $v_w$ may be the same as $s_w$ (i.e., $v_w$ may be the southwest corner of $SE(u, \alpha')$). See Figure 4.3. If $w$ is live at the start of phase $i$, then $w$'s Eliminate message reaches $s_w$ before $u$'s Eliminate message does. Our second idea is that, when traversing the south boundary of

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure4.3.png}
\caption{Processor $w$ in $SW(u, \alpha')$}
\end{figure}
$SE(u, a')$, $u$'s Eliminate message $E_u$ can determine the processor $w_0$ with the largest identifier in $SW(u, a')$ whose $v_{w_0}$ reaches the south boundary of $SE(u, a')$. Thus when $E_u$ reaches the processor $v_{w_0}$ on the west boundary of $SE(u, a')$, then $v_{w_0}$ sends a Kill message to $w_0$, and $E_u$ continues the traversal of the boundary of $SE(u, a')$. Processor $w_0$ is eliminated as soon as $w_0$ receives a Kill message. Note that $E_u$ spawns at most one Kill message in phase $i$.

Our third idea is as follows. Suppose that an Eliminate message $E_u$ successfully traverses the north boundary of $SE(u, a')$ in phase $i$, but that a processor $v$ on the east or south boundary of $SE(u, a')$ discards $E_u$ because $ID(u) < \text{largest-seen}(v)$. Then $v$ sends a Mark message $M_u$ that contains $ID(u)$ to traverse the remainder of the east and south boundary of $SE(u, a')$. $M_u$ makes certain that $\text{largest-seen}(v')$ is at least $ID(u)$ for each processor $v'$ on the east and south boundary of $SE(u, a')$. Thus when $v'$ receives $M_u$, processor $v'$ compares $ID(u)$ with $\text{largest-seen}(v')$. If $ID(u) \leq \text{largest-seen}(v')$, then $M_u$ continues its traversal. If $\text{largest-seen}(v') < ID(u)$, then $v'$ is eliminated, $v'$ sets $\text{largest-seen}(v')$ to $ID(u)$, and $v'$ relays $M_u$. The processor on the southwest corner of $SE(u, a')$ discards $M_u$. Similarly, if a processor $v''$ on the north boundary of $SE(u, a')$ discards $E_u$, then $v''$ sends a Mark message $M_u$ that contains $ID(u)$ to traverse the remainder of the north boundary of $SE(u, a')$. The processor on the northeast corner of $SE(u, a')$ discards $M_u$. (The necessity for distinguishing between the $v$ and $v''$ cases will become clear in Lemma 5.11.)

We will show that our second and third ideas are sufficient to guarantee that all processors in $SW(u, a') \cup NE(u, a') \setminus \{u\}$ are eliminated when phase $i+1$ starts.

For our fourth idea, let $E_u$ be an Eliminate message that is not discarded in phase $i^* = \lceil \log_a \sqrt{n} \rceil$. Then $E_u$ loops around the row containing processor $u$ and reaches $u$ from the west. We will prove that in phase $i^*$, there is at most a constant number of processors that receive their Eliminate messages from the west. Thus for phase $i^*+1$, the live processors execute a simple algorithm that guarantees the uniqueness of the leader. At the start of phase $i^*+1$, $u$ sends to the north a Final message $F_u$ that contains $ID(u)$. Let $z$ be the processor with the largest identifier in the network. In the proof of correctness, we will prove
that $z$ executes phase $i^* + 1$. Thus if $F_u$ reaches a processor $z'$ in the row $\text{Row}(z)$ containing $z$, then

$ID(u) < ID(z) = \text{largest}-\text{seen}(z')$, and $z'$ discards $F_u$. On the other hand, $F_z$ traverses the entire column containing $z$. When $z$ receives $F_z$ from the south, then $z$ knows that $z$ has the largest identifier in the network, and that all Final messages $F_u$ with $u \neq z$ will be discarded when they encounter $\text{Row}(z)$. Thus $z$ elects itself as the unique leader.

4.3.2 Details of the algorithm

Each processor $u$ keeps a variable called $\text{largest}-\text{seen}(u,i)$. Intuitively, $\text{largest}-\text{seen}(u,i)$ is the largest identifier in an Eliminate message that $u$ receives in phase $i$ from the west or from the north. At the start of phase $i$, $\text{largest}-\text{seen}(u,i) = \text{nil}$, where nil is smaller than the identifier of every processor. For every time $t$, $\text{largest}-\text{seen}(u,i)$ at $t$ is no more than $\text{largest}-\text{seen}(u)$ at $t$. Also, $\text{largest}-\text{seen}(u,i)$ can never be equal to $ID(u)$. Processor $u$ also maintains a variable called $\text{eliminated}-\text{from}(u)$. Intuitively, $\text{eliminated}-\text{from}(u)$ specifies the direction from which the message that eliminated $u$ came. Initially, $\text{eliminated}-\text{from}(u)$ is undefined. If $u$ is live at the start of phase $i$, then $\text{largest}-\text{seen}(u) = ID(u)$.

At the start of each phase $i \geq 0$, each live processor $y$ sends the message $E_y =$ "Eliminate$_{[ID(y),c',\text{nil}]}$" to the east. Next, each processor $u$, whether $u$ is live or dead, processes each message that $u$ receives. Suppose that $u$ receives the message $E_y =$ "Eliminate$_{[ID(v),k,ID']}"$ from the direction $\text{dir}$, where $v$ is some processor, $1 \leq k \leq c'$, and $\text{dir} \in \{\text{west}, \text{north}, \text{east}, \text{south}\}$. An invariant of the algorithm is that $ID' = \text{nil}$ if $\text{dir}$ is west or north; otherwise, $ID'$ is $\text{largest}-\text{seen}(w,i)$ for some $w$ on the south boundary of $SE(v,c')$. Depending on the value of $\text{dir}$, $u$ executes one of the following four cases:

Case 1: $\text{dir} = \text{west}$:

There are two subcases:
Case 1.1: \( ID(v) \neq ID(u) \):

Then \( E_v \) is a message from some processor \( v \) different from \( u \).

If \( \text{largest-seen}(u) > ID(v) \), then \( u \) discards \( E_v \). Processor \( u \) sets \( \text{largest-seen}(u,i) \) to \( \max(\text{largest-seen}(u,i), ID(v)) \). Also, if \( k \neq 1 \), then \( u \) sends "Mark_{[ID(v),k-1]}" to the east.

If \( \text{largest-seen}(u) \leq ID(v) \), then \( u \) sets \( \text{largest-seen}(u) \) to \( ID(v) \), \( u \) sets \( \text{eliminated-from}(u) \) to \( \text{west} \), and \( u \) is eliminated. Processor \( u \) sets \( \text{largest-seen}(u,i) \) to \( ID(v) \). Also, if \( k \neq 1 \), then \( u \) sends "Eliminate_{[ID(v),k-1,nil]}" to the east. If \( k = 1 \), then \( u \) is at the northwest comer of \( SE(v, \alpha') \), and \( u \) sends "Eliminate_{[ID(v),\alpha',nil]}" to the south.

Case 1.2: \( ID(v) = ID(u) \):

Then \( E_v \) is an Eliminate message from \( u \). Thus \( E_v \) originated at \( u \), traversed the entire row \( \text{Row}(u) \) containing \( u \), and returned to \( u \). No processor other than \( u \) is currently live in \( \text{Row}(u) \) since \( E_v \) was not discarded. Thus \( u \) executes the procedure \( \text{FINAL} \) defined as follows. Processor \( u \) sends the message \( F_u = \"\text{Final}_{[ID(u)]}\" \) to the north. Each processor \( v \neq u \) with \( \text{largest-seen}(v) \leq ID(u) \) passes \( F_u \) to the north. If \( F_u \) arrives at some \( v \) with \( ID(u) < \text{largest-seen}(v) \), then \( v \) discards \( F_u \). If \( F_u \) arrives at \( u \), then \( u \) elects itself as the leader.

Case 2: \( dir = \text{north} \):

Then \( ID(v) \neq ID(u) \).

If \( \text{largest-seen}(u) > ID(v) \), then \( u \) sets \( \text{largest-seen}(u,i) \) to \( \max(\text{largest-seen}(u,i), ID(v)) \), and \( u \) discards \( E_v \). Also, if \( k \neq 1 \), then \( u \) sends "Mark_{[ID(v),k-1]}" to the south. If \( k = 1 \), then \( u \) sends "Mark_{[ID(v),\alpha']}" to the west.

If \( \text{largest-seen}(u) \leq ID(v) \), then \( u \) sets \( \text{largest-seen}(u) \) to \( ID(v) \), \( u \) sets \( \text{eliminated-from}(u) \) to \( \text{north} \), and \( u \) is eliminated. Also, if \( k \neq 1 \), then \( u \) sends "Eliminate_{[ID(v),k-1,nil]}" to the south. If \( k = 1 \), then \( u \) sends "Eliminate_{[ID(v),\alpha',\text{largest-seen}(u,i)]}" to the west. Finally, \( u \) sets \( \text{largest-seen}(u,i) \) to \( ID(v) \).
Case 3: \( \text{dir} = \text{east} \):

Then \( ID(v) \neq ID(u) \).

If \( \text{largest-seen}(u) > ID(v) \), then \( u \) discards \( E_v \). Processor \( u \) sets \( \text{largest-seen}(u,i) \) to \( \max\{\text{largest-seen}(u,i), ID(v)\} \). Also, if \( k \neq 1 \), then \( u \) sends "Mark\_[ID(v),k-1"]" to the west. If \( k = 1 \), then \( u \) does not send a Mark message.

If \( \text{largest-seen}(u) \leq ID(v) \), then \( u \) sets \( \text{largest-seen}(u) \) to \( ID(v) \), and \( u \) is eliminated. Also, \( u \) does one of the following:

Case 3.1: \( k \neq 1 \):

Processor \( u \) sets \( \text{eliminated-from}(u) \) to east. Also, if \( ID(u) = ID' \) and \( \text{largest-seen}(u,i) < ID' \), then \( u \) sends "Eliminate\_[ID(v),k-1,nil]" to the west. Otherwise, \( u \) sends "Eliminate\_[ID(v),k-1, \( \max\{ID', \text{largest-seen}(u,i)\}]" to the west. Finally, \( u \) sets \( \text{largest-seen}(u,i) \) to \( ID(v) \).

Case 3.2: \( k = 1 \):

Processor \( u \) is at the southwest corner of \( SE(v, \alpha') \). If \( ID(u) = ID' \), then \( u \) sends "Eliminate\_[ID(v),\alpha',\text{nil}]" to the north, and \( u \) sets \( \text{eliminated-from}(u) \) to east. Processor \( u \) sets \( \text{largest-seen}(u,i) \) to \( ID(v) \).

If \( ID(u) \neq ID' \) and \( ID' \leq \text{largest-seen}(u,i) \), then \( u \) sends "Eliminate\_[ID(v),\alpha',\text{largest-seen}(u,i)]" to the north. Also, if \( \text{eliminated-from}(u) = \text{west} \), then \( u \) sends \( K_v = \text{"Kill\_[largest-seen(u,i)]"} \) to the west. Every processor \( x \) that receives \( K_v \) forwards the message to the west until \( K_v \) reaches a processor \( x \) such that \( ID(x) = \text{largest-seen}(u,i) \). Processor \( x \) becomes eliminated, but \( \text{eliminated-from}(x) \) is not changed. If \( \text{eliminated-from}(u) \neq \text{west} \), then \( u \) does not send a Kill message. Finally, \( u \) sets \( \text{eliminated-from}(u) \) to east so that \( u \) initiates at most one Kill message in phase \( i \). Also, \( u \) sets \( \text{largest-seen}(u,i) \) to \( ID(v) \).

If \( ID(u) \neq ID' \) and \( ID' > \text{largest-seen}(u,i) \), then \( u \) sets \( \text{eliminated-from}(u) \) to east and sends "Eliminate\_[ID(v),\alpha',ID']" to the north. Processor \( u \) sets \( \text{largest-seen}(u,i) \) to \( ID(v) \).
Case 4: \( \text{dir} = \text{south}: \)

If \( ID(v) = ID(u) \), then \( k = 1 \) and \( E_v \) is a message from \( u \). If \( u \) is live when \( u \) receives \( E_v \), then \( u \) continues to receive messages for an additional \( \alpha' \) time units, and \( u \) advances to phase \( i+1 \) if \( u \) does not receive a Kill message during these \( \alpha' \) time units. If \( u \) is dead when \( u \) receives \( E_v \), then \( u \) discards \( E_v \).

If \( ID(v) \neq ID(u) \) and \( \text{largest}-\text{seen}(u) > ID(v) \), then \( u \) discards \( E_v \). Processor \( u \) sets \( \text{largest}-\text{seen}(u,i) \) to \( \max(\text{largest}-\text{seen}(u,i), ID(v)) \).

If \( ID(v) \neq ID(u) \) and \( \text{largest}-\text{seen}(u) \leq ID(v) \), then \( k \neq 1 \), and \( u \) is eliminated. Also, \( u \) sends "\text{Eliminate}_{[ID(v),k-1,ID']}" to the north. In addition, if \( ID' = \text{largest}-\text{seen}(u) \) and \( \text{eliminated}-\text{from}(u) = \text{west} \), then \( u \) also sends "\text{Kill}[ID']" to the west. Otherwise, \( u \) does not send a Kill message. Finally, \( u \) sets \( \text{eliminated}-\text{from}(u) \) to south so that \( u \) initiates at most one Kill message in phase \( i \). Processor \( u \) sets \( \text{largest}-\text{seen}(u,i) \) to \( ID(v) \).

Suppose that \( u \) receives the message "\text{Mark}_{[ID(v),k]}" from the direction \( \text{dir} \), where \( 1 \leq k \leq \alpha' \), \( v \) is some processor, and \( \text{dir} \in \{ \text{west}, \text{north}, \text{east} \} \). First, \( u \) compares \( \text{largest}-\text{seen}(u) \) with \( ID(v) \). If \( \text{largest}-\text{seen}(u) < ID(v) \), then \( u \) sets \( \text{largest}-\text{seen}(u) \) to \( ID(v) \), and \( u \) is eliminated. Next, regardless of the relative sizes of \( \text{largest}-\text{seen}(u) \) and \( ID(v) \), \( u \) does one of the following depending on the value of \( \text{dir} \):

Case 1: \( \text{dir} = \text{west} \):

If \( k \neq 1 \), then \( u \) sends "\text{Mark}_{[ID(v),k-1]}" to the east.

If \( k = 1 \), then \( u \) does not send a Mark message.

Case 2: \( \text{dir} = \text{north} \):

If \( k \neq 1 \), then \( u \) sends "\text{Mark}_{[ID(v),k-1]}" to the south.

If \( k = 1 \), then \( u \) sends "\text{Mark}_{[ID(v),\alpha']}" to the west.
Case 3: \( \text{dir} = \text{east} \):

If \( k \neq 1 \), then \( u \) sends "Mark\(_{[ID(v),k-1]}\)" to the west.

If \( k = 1 \), then \( u \) does not send a Mark message.

Processor \( u \) remains live at the end of phase \( i \) if either \( u \) receives \( E_u \) from the south after \( 4\alpha_i \) time units and does not receive the message "Kill\(_{[ID(u)]}\)" or \( u \) receives \( E_u \) from the west.

4.4 Proof of Correctness

Let \( z \) be the processor with the largest identifier in the network.

Lemma 4.1: Processor \( z \) executes the procedure \( \text{FINAL(Case 1.2 of the algorithm)} \).

Proof: Processor \( z \) does not execute \( \text{FINAL} \) only if \( z \) is eliminated during some phase \( i \leq i^* \). Since \( z \) is the processor with the largest identifier, \( z \) can not be eliminated by receiving an \( \text{Eliminate} \) message.

Let \( E_z \) be an \( \text{Eliminate} \) message that \( z \) sends at the start of some phase \( i \). By our choice of \( z \), every processor \( v \) on the path that \( E_z \) traverses has \( \text{largest-seen}(v) \leq \text{ID}(z) \). Therefore, \( E_z \) is never discarded, and \( z \) can not be eliminated by a discard of \( E_z \).

The only other way that \( z \) may be eliminated is if \( z \) receives a \( \text{Kill} \) message \( K = "\text{Kill}_{[ID(z)]}" \). By the algorithm, \( K \) was spawned by some processor \( u \neq z \) in the same row \( \text{Row}(z) \) as \( z \), with \( \text{largest-seen}(u) = \text{ID}(z) \). Furthermore, \( u \) spawned \( K \) because \( u \) received a message \( E_u = "\text{Eliminate}_{[ID(v),k,ID(z)]}" \) from the direction \( \text{dir} \), where \( 1 \leq k \leq \alpha_i \), and \( \text{dir} = \{\text{east, south}\} \). By the algorithm, \( \text{ID}(v) \geq \text{largest-seen}(u) = \text{ID}(z) \). Thus, since \( z \) has the largest identifier, \( \text{ID}(v) = \text{ID}(z) \), and \( E_v \) was sent by \( z \). But an \( \text{Eliminate} \) message sent by \( z \) can not reach a processor \( u \) in \( \text{Row}(z) \) from the east or from the south, unless \( u = z \). Thus \( K \) does not exist. \( \square \)

Theorem 4.1: The algorithm elects \( z \) as the leader when the algorithm terminates.

Proof: Processor \( z \) executes the procedure \( \text{FINAL} \) because \( z \) receives its \( \text{Eliminate} \) message from the west in phase \( i^* \). Thus, at the start of \( \text{FINAL} \), all the processors \( v \) in \( \text{Row}(z) \) have \( \text{largest-seen}(v) = \text{ID}(z) \), and all the processors except \( z \) in \( \text{Row}(z) \) are eliminated. Suppose that another
processor \( u \) is also live at the start of FINAL. By the algorithm, \( z \) and \( u \) send final messages to the north. Since \( z \) has the largest identifier, \( z \)'s Final message will return to \( z \) from the south. Either \( u \)'s Final message \( F_u \) is discarded before reaching \( \text{Row}(z) \), or \( F_u \) reaches a processor \( v \) in \( \text{Row}(z) \). In the latter case, \( F_u \) is discarded since \( ID(u) \) is smaller than \( \text{largest-seen}(v) = ID(z) \). Thus \( u \) will be eliminated. Only \( z \) announces itself as the leader. □

4.5 Message Complexity

Lemma 5.1: Let \( r \) be a processor. Suppose that some processor receives an Eliminate message \( E_r \) from the north or from the east in phase \( i \). Then all the processors \( r_0 \) in \( SE(r, \alpha') \) with \( ID(r_0) < ID(r) \) are dead at the start of phase \( i+1 \).

Proof: If \( r_0 \) is dead at the start of phase \( i \), then \( r_0 \) is dead at the start of phase \( i+1 \). Hence, suppose that \( r_0 \) is live at the start of phase \( i \). Since there is a processor that receives \( E_r \) from the north or from the east in phase \( i \), every processor on the south boundary of \( SE(r, \alpha') \) receives \( E_r \) or \( M_r \). Either \( E_{r_0} \) is discarded before \( E_{r_0} \) reaches some processor on the south boundary of \( SE(r, \alpha') \) from the south, or \( E_{r_0} \) reaches a processor \( r_1 \) on the south boundary of \( SE(r, \alpha') \) from the south. In the later case, \( r_1 \) receives \( E_{r_0} \) after \( r_1 \) receives either \( E_r \) or \( M_r \). See Figure 4.4. Thus \( E_{r_0} \) reaches \( r_1 \) when \( \text{largest-seen}(r_1) \geq ID(r) > ID(r_0) \), and \( r_1 \) discards \( E_{r_0} \). Therefore, \( r_0 \) will be dead at the start of phase \( i+1 \). □

Lemma 5.2: Let \( r \) be a processor live at the start of phase \( i \). Then all the processors \( r_0 \) in \( NE(r, \alpha') \cup NW(r, \alpha') \) with \( ID(r_0) < ID(r) \) are dead at the start of phase \( i+1 \).

Proof: If \( r_0 \) is dead at the start of phase \( i \), then \( r_0 \) is dead at the start of phase \( i+1 \). Hence, suppose that \( r_0 \) is live at the start of phase \( i \). Since \( r \) is live at the start of phase \( i \), every processor on the north boundary of \( SE(r, \alpha') \) receives \( E_r \) or \( M_r \).

Suppose that \( r_0 \) is in \( NW(r, \alpha') \). Either \( E_{r_0} \) is discarded before \( E_{r_0} \) reaches some processor on the north boundary of \( SE(r, \alpha') \), or \( E_{r_0} \) reaches a processor \( r_1 \) on the north boundary of \( SE(r, \alpha') \). In the later case, \( r_1 \) receives \( E_{r_0} \) after \( r_1 \) receives either \( E_r \) or \( M_r \). Thus \( E_{r_0} \) reaches \( r_1 \) when \( \text{largest-seen}(r_1) \geq ID(r) > ID(r_0) \), and \( r_1 \) discards \( E_{r_0} \). Therefore, \( r_0 \) will be dead at the start of phase \( i+1 \).
Now suppose that $r_0$ is in $NE(r, \alpha')$. Either $E_{r_0}$ is discarded before $E_{r_0}$ reaches some processor on the north boundary of $SE(r, \alpha')$ from the south, or $E_{r_0}$ reaches a processor $r_1$ on the north boundary of $SE(r, \alpha')$ from the south. In the later case, $r_1$ receives $E_{r_0}$ after $r_1$ receives either $E_r$ or $M_r$. Thus $E_{r_0}$ reaches $r_1$ when $\text{largest\text{-}seen}(r_1) \geq ID(r) > ID(r_0)$, and $r_1$ discards $E_{r_0}$. Therefore, $r_0$ will be dead at the start of phase $i+1$. □

**Lemma 5.3:** Let $r$ be a processor live at the start of phase $i$. If there is a processor $q$ in $SE(r, \alpha') \cup SW(r, \alpha')$ live at the start of phase $i$ with $ID(q) > ID(r)$, then $r$ will be dead at the start of phase $i+1$.

**Proof:** Processor $r$ is in $NE(q, \alpha') \cup NW(q, \alpha')$. By Lemma 5.2, $r$ is dead at the start of phase $i+1$. □

**Lemma 5.4:** Let $r$ be a processor live at the start of phase $i$, and let $q$ be a processor in $NW(r, \alpha')$ with $ID(q) > ID(r)$. Suppose that a processor receives the Eliminate message $E_q$ from the north or from the east in phase $i$. Then $r$ will be dead at the start of phase $i+1$.

**Proof:** Processor $r$ is in $SE(q, \alpha')$. By Lemma 5.1, $r$ is dead at the start of phase $i+1$. □
Fix processor \( u \) to be live at the start of some phase \( i+1 \), where \( 0 \leq i \leq i^*-1 \). Thus, in phase \( i \), all processors on the east boundary of \( SE(u, \alpha^i) \) receive \( E_u \) from the north. Thus we can apply Lemmas 5.1 - 5.4 with \( r = u \). Lemmas 5.5 - 5.10 pertain to \( u \).

Lemma 5.5: All the processors in \( SE(u, \alpha^i) \cup NW(u, \alpha^i) - \{u\} \) are dead at the start of phase \( i+1 \).

Proof: Let \( u_0 \neq u \) be a processor in \( SE(u, \alpha^i) \) live at the start of phase \( i \). Since \( u \) is live at the start of phase \( i+1 \), \( ID(u_0) < ID(u) \) by Lemma 5.3. Thus \( u_0 \) is dead at the start of phase \( i+1 \) by Lemma 5.1. All processors in \( SE(u, \alpha^i) - \{u\} \) are, therefore, dead at the start of phase \( i+1 \).

Suppose, contrary to the lemma, a processor \( u_1 \neq u \) in \( NW(u, \alpha^i) \) is live at the start of phase \( i+1 \).

Then, by the argument we have just given, since \( u \) is in \( SE(u_1, \alpha^i) \), processor \( u \) would be dead at the start of phase \( i+1 \), contrary to the hypothesis. Hence all processors in \( NW(u, \alpha^i) - \{u\} \) are dead at the start of phase \( i+1 \).

We now show that all processors in \( SW(u, \alpha^i) - \{u\} \) are dead at the start of phase \( i+1 \). Let \( x \) be the processor at the southwest corner of \( SE(u, \alpha^i) \). Since \( u \) is live at the start of phase \( i+1 \), processor \( x \) sends \( E_{ux} = \text{"Eliminate}_{\{ID(u), \alpha^i, ID'\}} \) to the north in phase \( i \), where \( ID' \) is an identifier of a processor or is nil. By the algorithm, \( ID' \neq ID(u) \).

Lemma 5.6: If \( ID' = \text{nil} \) in \( E_{ux} \), then all processors in \( SW(u, \alpha^i) - \{u\} \) are dead at the start of phase \( i+1 \).

Proof: According to the algorithm, processor \( x \) sends \( E_{ux} \) with \( ID' = \text{nil} \) because \( x \) received the message \( E'_u = \text{"Eliminate}_{\{ID(u), 1, ID^"\}} \)," where \( ID^" \) is \( ID(x) \) or is nil. By the algorithm, \( x \) is dead when \( x \) sends \( E_{ux} \). Now consider all processors \( w \) in \( SW(u, \alpha^i) - \{u, x\} \) that are live at the start of phase \( i \). If \( E_w \) is discarded before \( E_w \) reaches the south boundary of \( SE(u, \alpha^i) \), then \( w \) is dead at the start of phase \( i+1 \).

Thus suppose that \( E_w \) reaches some processor \( u_0 \) on the south boundary of \( SE(u, \alpha^i) \), and that \( u_0 \) does not discard \( E_w \). Then \( \text{largest-seen}(u_0, i) \leq ID(w) \) when \( E_w \) reaches \( u_0 \). Thus \( ID^" \) in \( E_w \) is at least \( ID(w) \). Recall that \( ID^" \in \{\text{nil}, ID(x)\} \). Since \( ID^" \geq ID(w) > \text{nil} \), \( ID^" = ID(x) \). Since \( w \neq x \), \( ID(x) > ID(w) \).

By Lemma 5.2, \( w \) is dead at the start of phase \( i+1 \). Thus all processors in \( SW(u, \alpha^i) - \{u, x\} \) are also dead at the start of phase \( i+1 \). Hence all the processors in \( SW(u, \alpha^i) - \{u\} \) will be dead at the start of phase \( i+1 \).
Now suppose that \( ID' \neq \text{nil} \). Thus \( ID' = ID(v_0) \), where \( v_0 \) is some processor. Since \( E_w \) contains \( ID(v_0), v_0 \neq u \). Since \( ID' \neq \text{nil}, v_0 \neq x \). Note that, by the algorithm, \( v_0 \) must be live at the start of phase \( i \) for \( ID(v_0) \) to be in \( E_w \). Lemmas 5.7 - 5.9 pertain to \( u \) and \( v_0 \).

**Lemma 5.7**: Processor \( v_0 \) is in \( SW(u, \alpha') \).

**Proof**: Let \( u_0 \) be the western-most processor on the south boundary of \( SE(u, \alpha') \) with \( \text{largest-seen}(u_0,i) = ID(v_0) \) when \( u \)'s Eliminate message \( E_u \) reaches \( u_0 \). Since \( \text{largest-seen}(u_0,i) = ID(v_0) \), processor \( u_0 \) received an Eliminate message \( E_{v_0} \) that contained \( ID(v_0) \). Therefore, \( v_0 \) is on the boundary of \( NW(u_0, \alpha') \). Suppose that, contrary to the lemma, \( v_0 \) is not in \( SW(u, \alpha') \). Since \( v_0 \) is on the boundary of \( NW(u_0, \alpha') \), either \( v_0 \) is on the north boundary of \( SE(u, \alpha') \), or \( v_0 \) is on the south boundary of \( SE(u, \alpha') \). We will show that both cases lead to contradictions.

If \( v_0 \) were on the north boundary of \( SE(u, \alpha') \), then \( E_{v_0} \) would reach \( u_0 \) after \( E_w \) reaches \( u_0 \). Hence \( \text{largest-seen}(u_0,i) \neq ID(v_0) \) when \( E_w \) reaches \( u_0 \), a contradiction.

If \( v_0 \) were on the south boundary of \( SE(u, \alpha') \), then, by our choice of \( u_0 \), \( u_0 = v_0 \). By the algorithm, \( v_0 \) would send the message \"Eliminate_{[ID(u), \text{nil}]\} \" to the west when \( v_0 \) received \( E_u \).

Hence \( E_w \) would not contain \( ID(v_0) \), a contradiction. \( \square \)

As in Figure 4.5, let the rectangle \( SW(u,q,q') \) denote a rectangular mesh of processors with processor \( u \) at the northeast corner, \( q+1 \) processors on the east and west boundaries, and \( q'+1 \) processors on the north and south boundaries. Let the rectangle \( NE(u,q,q') \) denote a rectangular mesh of processors with processor \( u \) at the southwest corner, \( q+1 \) processors on the east and west boundaries, and \( q'+1 \) processors on the north and south boundaries. Recall that \( u \) is live at the start of phase \( i+1 \).

**Lemma 5.8**: Processor \( v_0 \) is not in the rectangle \( SW(u, \alpha'^{-1}, \alpha') \).

**Proof**: Processor \( u \) is live at the start of phase \( i \). Hence \( u \) received an Eliminate message \( E_u \) in phase \( i-1 \). Thus all processors \( b \) on the east boundary of \( SW(u, \alpha'^{-1}, \alpha') \) have \( \text{largest-seen}(b) \geq ID(u) \) at the start of phase \( i \). Since \( u \) is live at the start of phase \( i+1 \) and \( v_0 \in SW(u, \alpha'), ID(v_0) < ID(u) \) by Lemma 5.3. Thus if, contrary to the lemma, \( v_0 \) were in \( SW(u, \alpha'^{-1}, \alpha') \), then \( v_0 \)'s Eliminate message \( E_{v_0} \) in
Figure 4.5. The Definitions of $NW(u,q,q')$, $NE(u,q,q')$, $SE(u,q,q')$, and $SW(u,q,q')$

phase $i$ would be discarded when $E_{v_0}$ reached the east boundary of $SW(u, \alpha^{i-1}, \alpha')$, $E_{v_0}$ would not reach the south boundary of $SE(u, \alpha')$, and $ID(v_0)$ would not be in $E_{uw}$, a contradiction. Hence $v_0$ is not in $SW(u, \alpha^{i-1}, \alpha')$. □

We will need Lemma 5.9 in the proof of Lemma 5.10.

Lemma 5.9: Let $v_w$ be a processor on the west boundary of $SE(u, \alpha')$. Let $w_0$ be a processor in $SW(u, \alpha')$, and let $r \neq w_0$ be a processor distinct from $u$. Suppose that $\text{largest-seen}(v_w, i) = ID(w_0)$ at some time during phase $i$. If $\text{largest-seen}(v_w, i)$ becomes $ID(r)$, then $w_0$ is dead by the time phase $i+1$
starts.

Proof: Since \( v_w \) changes \( \text{largest-seen}(v_w, i) \) from \( ID(w_0) \) to \( ID(r) \), by the algorithm, \( v_w \) receives an Eliminate message \( E_r \) from \( r \) such that \( ID(w_0) < ID(r) \).

If \( v_w \) receives \( E_r \) from the north, then \( w_0 \) is in \( SE(r, \alpha') \) since \( v_w \) receives \( E_r \) after \( v_w \) receives \( w_0 \)'s Eliminate message \( E_{w_0} \). Since \( ID(w_0) < ID(r) \), then \( w_0 \) is dead at the start of phase \( i+1 \) by Lemma 5.1.

If \( v_w \) receives \( E_r \) from the west, then \( r \) is in the same row \( Row(v_w) \) as \( v_w \). Since \( v_w \) receives \( E_r \) after \( v_w \) receives \( E_{w_0} \), \( w_0 \) is in \( Row(v_w) \), and \( w_0 \) is to the east of \( r \) and to the west of \( v_w \). Thus \( E_r \) reaches \( w_0 \), and \( w_0 \) will be dead at the start of phase \( i+1 \).

Finally, we show that \( v_w \) could not have received \( E_r \) from the east or from the south. If, to the contrary, \( v_w \) received \( E_r \) from the east, then \( r \) would be in \( NW(u, \alpha') \). By Lemma 5.4, since \( u \) is live at the start of phase \( i+1 \), \( ID(r) < ID(u) \). Hence \( E_r \) would be discarded when \( E_r \) reached the north boundary of \( SE(u, \alpha') \) and before \( E_r \) reached \( v_w \). If \( v_w \) received \( E_r \) from the south, then \( r \) would be in the same column as \( v_w \). Further, \( r \) would be in \( SE(u, \alpha') \) or in \( NW(u, \alpha') \). In either case, \( ID(r) < ID(u) \) by Lemmas 5.3 and 5.4, since \( u \) is live at the start of phase \( i+1 \). If \( r \in SE(u, \alpha') \), then \( E_r \) would be discarded when \( E_r \) reached the south boundary of \( SE(u, \alpha') \) and before \( E_r \) reached \( v_w \). If \( r \in NW(u, \alpha') \), then \( E_r \) would be discarded when \( E_r \) reached the north boundary of \( NW(u, \alpha') \) and before \( E_r \) reached \( v_w \). Thus, \( v_w \) would not set \( \text{largest-seen}(v_w, i) \) to \( ID(r) \), a contradiction.\( \square \)

Lemma 5.10: If \( \alpha \leq 2 \), then all processors in \( SW(u, \alpha') \cup NE(u, \alpha')-{u} \) are dead at the start of phase \( i+1 \).

Proof: We first show that all processors in \( SW(u, \alpha')-{u} \) are dead at the start of phase \( i+1 \). Consider phase \( i \), and consider all processors \( w_0 \) in \( SW(u, \alpha') \) live at the start of phase \( i \) with \( ID(w_0) > ID(v_0) \).

We will show that \( w_0 \)'s Eliminate message \( E_{w_0} \) in phase \( i \) is discarded before or when \( E_{w_0} \) reaches the south boundary of \( SE(u, \alpha') \), and thus \( w_0 \) will be dead at the start of phase \( i+1 \). If \( E_{w_0} \) were not discarded before or when \( E_{w_0} \) reached the south boundary of \( SE(u, \alpha') \), then, since \( ID(w_0) > ID(v_0) \), \( E_{w_0} \) would not contain \( ID(v_0) \); instead, \( E_{w_0} \) would contain \( ID(w_0) \) or a larger identifier, a contradiction.
Now consider all processors \( w_0 \) in \( SW(u, \alpha') \) live at the start of phase \( i \) with \( ID(w_0) < ID(v_0) \). By Lemma 5.8, \( v_0 \) is not in the rectangle \( SW(u, \alpha'^{-1}, \alpha') \). Since \( \alpha \leq 2 \),

\[
SW(u, \alpha') \subseteq NE(v_0, \alpha') \cup NW(v_0, \alpha') \cup SE(v_0, \alpha'^{-1}, \alpha') \cup SW(v_0, \alpha'^{-1}, \alpha').
\]

If \( w_0 \in NE(v_0, \alpha') \cup NW(v_0, \alpha') \), then \( w_0 \) is dead at the start of phase \( i+1 \) by Lemma 5.2.

If \( w_0 \in SE(v_0, \alpha'^{-1}, \alpha') \), then \( w_0 \) is dead at the start of phase \( i+1 \) by Lemma 5.1.

Now suppose that \( w_0 \notin SW(v_0, \alpha'^{-1}, \alpha') \). Since \( v_0 \) is live at the start of phase \( i \), processor \( v_0 \) received the Eliminate message that \( v_0 \) sent in phase \( i-1 \). Thus all processors \( v_1 \) on the east boundary of \( SW(v_0, \alpha'^{-1}, \alpha') \) have \( \text{largest-seen}(v_1) \geq ID(v_0) > ID(w_0) \) at the start of phase \( i \). Hence, \( E_{w_0} \) will be discarded if \( E_{w_0} \) reaches the east boundary of \( SW(v_0, \alpha'^{-1}, \alpha') \). Consequently \( w_0 \) will be dead at the start of phase \( i+1 \).

We now show that \( v_0 \) is dead at the start of phase \( i+1 \). Let \( v_w \) be the first processor on the west boundary of \( SE(u, \alpha') \) to receive \( E_{v_0} \). If \( E_{v_0} \) is discarded when \( E_{v_0} \) reaches \( v_w \), then \( v_0 \) is dead at the start of phase \( i+1 \). Suppose now that \( v_w \) does not discard \( E_{v_0} \). When \( E_{v_0} \) reached \( v_w \), processor \( v_w \) set \( \text{largest-seen}(v_w, i) \) to \( ID(v_0) \) and \( \text{eliminated-from}(v_w) \) to west. By the algorithm, \( v_w \) changes \( \text{eliminated-from}(v_w) \) only if \( v_w \) changes \( \text{largest-seen}(v_w, i) \). By Lemma 5.9, if \( \text{largest-seen}(v_w, i) \) is not \( ID(v_0) \) when \( E_{w_0} \) reaches \( v_w \), then \( v_0 \) is dead at the start of phase \( i+1 \). Thus suppose that \( \text{eliminated-from}(v_w) \) is west and \( \text{largest-seen}(v_w, i) = ID(v_0) \). By the algorithm, \( v_w \) sends a Kill message to \( v_0 \) in phase \( i \). Thus \( v_0 \) will be dead at the start of phase \( i+1 \).

We have shown so far that all processors in \( SW(u, \alpha') - \{u\} \) are dead at the start of phase \( i+1 \). If a processor \( r \neq u \) in \( NE(u, \alpha') \) were live at the start of phase \( i+1 \), then, by the argument we have just given, since \( u \) is in \( SW(r, \alpha') \) \( u \) would be dead at the start of phase \( i+1 \), contrary to the hypothesis. Hence all processors in \( NE(u, \alpha') - \{u\} \) are dead at the start of phase \( i+1 \).

We say that \( u \) generates \( k \) messages in phase \( i \) if
(the number of links that \( u \)'s Eliminate message \( E_u \) traverses) 

+ (the number of links that a Mark message spawned by \( E_u \) traverses) 

+ (the number of links that a Kill message spawned by \( E_u \) traverses) = \( k \).

Lemma 5.11: The algorithm uses at most \( 3n \) messages in phase 0.

Proof: Consider a processor \( u_0 \). Suppose that \( v_0 \) is the processor immediately to the south of \( u_0 \), and \( w_0 \) is the processor immediately to the west of \( u_0 \). See Figure 4.6. Processor \( u_0 \) generates at most five messages in phase 0: four Eliminate messages and one Kill message. Suppose that \( u_0 \) generates four or five messages. Then \( u_0 \) receives \( u_0 \)'s Eliminate message \( E_{u_0} \) from the south. If \( ID(u) \) were less than \( ID(w_0) \), then \( w_0 \)'s Eliminate message \( E_{w_0} \) would reach \( v_0 \) before \( E_{u_0} \), and \( v_0 \) would discard \( E_{u_0} \). Thus \( ID(w_0) < ID(u) \). Hence \( u_0 \) discards \( E_{w_0} \), and \( w_0 \) generates only one message.

Since every processor \( u_0 \) that generates at least four messages has a processor \( w_0 \) to the west that generates only one message, the average number of messages per processor in phase 0 is at most three. \( \square \)

Theorem 5.1: The algorithm uses at most \( \frac{229n}{18} \) messages.

Proof: Suppose that \( u \) is live at the start of phase \( i \), where \( 1 \leq i \leq i^* \). By Lemmas 5.5 and 5.10 with \( \alpha \leq 2 \), all the processors in \( SE(u, \alpha^{-1}) \cup SW(u, \alpha^{-1}) \cup NE(u, \alpha^{-1}) \cup NW(u, \alpha^{-1}) -(u) \) are dead at

---

Figure 4.6. Processors \( w_0 \), \( u_0 \), and \( v_0 \)
the start of phase $i$. Hence there are at most $\left(\frac{\sqrt{n}}{\alpha^{i-1}+1}\right)^2$ processors $u$ live at the start of phase $i$. Processor $u$ generates at most $5\alpha^i$ messages in phase $i$, where $1 \leq i \leq i^* - 1$. In phase $i^*$, $u$ generates at most $\alpha^*$ messages. Thus the total number of messages used by the algorithm in phases 0 through $i^*$ is $3n + \sum_{i=1}^{i^*-1} 5\alpha^i \frac{n}{(\alpha^{i-1}+1)^2} + \alpha^* \frac{n}{(\alpha^{i^*-1}+1)^2}$. At most $\frac{n}{(\alpha^{i^*-1}+1)}$ processors $u$ execute FINAL, and $u$ generates at most $\sqrt{n}$ messages in FINAL. Hence the algorithm uses at most $\frac{n}{(\alpha^{i^*-1}+1)^2} \sqrt{n}$ messages in FINAL.

**FINAL.** The total number $NUM$ of messages that the algorithm uses satisfies

$$NUM \leq 3n + \sum_{i=1}^{i^*-1} 5\alpha^i \frac{n}{(\alpha^{i-1}+1)^2} + \alpha^* \frac{n}{(\alpha^{i^*-1}+1)^2} + \sqrt{n} \frac{n}{(\alpha^{i^*-1}+1)}$$  \hspace{1cm} (4.1)

Consider each term in inequality (4.1):

$$\alpha^* \frac{n}{(\alpha^{i^*-1}+1)} + \frac{n}{(\alpha^{i^*-1}+1)^2} \sqrt{n} < 2 \alpha^* \frac{n}{(\alpha^{i^*-1}+1)^2} + \frac{n}{(\alpha^{i^*-1}+1)}$$

$$\leq 2 \alpha^2 \sqrt{n} = O(\sqrt{n}) = o(n)$$

Also,

$$\sum_{i=1}^{i^*-1} 5\alpha^i \frac{n}{(\alpha^{i-1}+1)^2} = \frac{5n\alpha}{4} + \frac{5n\alpha^2}{(\alpha+1)^2} + \sum_{i=3}^{i^*-1} 5\alpha^i \frac{n}{(\alpha^{i-1}+1)^2}$$

$$\leq \frac{5n\alpha}{4} + \frac{5n\alpha^2}{(\alpha+1)^2} + \sum_{i=3}^{i^*-1} 5\alpha^i \frac{n}{(\alpha^{i-1})^2}$$

$$= \frac{5n\alpha}{4} + \frac{5n\alpha^2}{(\alpha+1)^2} + \frac{5n}{\alpha^{i-1}} + o(n)$$  \hspace{1cm} (4.2)

The value of $\alpha$ that minimizes expression (4.2), subject to $1 < \alpha \leq 2$, is $\alpha = 2$. Hence,

$$NUM \leq \frac{229n}{18} + o(n). \square$$

### 4.6 Time Complexity

**Theorem 6.1:** The algorithm runs in time $O(\sqrt{n})$.

**Proof:** Each phase $i$ lasts $5\alpha^i$ time units, where $0 \leq i \leq i^*-1$. Phases $i^*$ and $i^*+1$ last $O(\sqrt{n})$ time units.
Hence, the running time of the algorithm is

\[ O(\sqrt{n}) + \sum_{i=0}^{i^*} 5\alpha^i = O(\sqrt{n}). \]

4.7 Lower Bounds

For our lower bound proof, we generalize the techniques of Frederickson and Lynch [22].

4.7.1 Assumptions

We now present the assumptions we use to obtain our lower bounds. For convenience, we assume that each processor is indexed by a unique number chosen from the set \( \{0, 1, \ldots, n-1\} \). When we say “processor p” we mean the “processor with index p.” The index of a processor is not necessarily the same as the processor’s identifier. We assume that each message that processor p sends contains p’s entire state. The current state of p incorporates all the messages that p received so far as follows. Recall that \( p_N \) denotes p’s north neighbor, \( p_E \) denotes p’s east neighbor, \( p_S \) denotes p’s south neighbor, and \( p_W \) denotes p’s west neighbor. Let \( \text{state}(p, i) \) denote the state of processor p at the start of round i. Then \( \text{state}(p, 0) = (\text{ID}(p)); \text{state}(p, i) = (s, s_E, s_S, s_W, s_N), \) for each round \( i \geq 1 \), where:

\[
\begin{align*}
    s_1 &= \text{state}(p, i-1); \\
    s_E &= \text{state}(p_E, i-1) \text{ if } p \text{ received a message from } p_E \text{ in round } i-1, \text{ otherwise } s_E = \text{nil}; \\
    s_S &= \text{state}(p_S, i-1) \text{ if } p \text{ received a message from } p_S \text{ in round } i-1, \text{ otherwise } s_S = \text{nil}; \\
    s_W &= \text{state}(p_W, i-1) \text{ if } p \text{ received a message from } p_W \text{ in round } i-1, \text{ otherwise } s_W = \text{nil}; \\
    s_N &= \text{state}(p_N, i-1) \text{ if } p \text{ received a message from } p_N \text{ in round } i-1, \text{ otherwise } s_N = \text{nil};
\end{align*}
\]

We say that the two states \( s \) and \( s' \) are order-equivalent provided that \( s \) and \( s' \) are structurally equivalent, and that if two identifiers in \( s \) satisfy one of the order relations \(<, =, \text{ or } >\), then the corresponding identifiers in \( s' \) satisfy the same order relation. An election algorithm is a comparison algorithm provided that if \( s \) and \( s' \) are order-equivalent processors, then processors with states \( s \) and \( s' \) transmit messages in the same direction and have the same election status. The algorithm we presented in Section 4.3 is a comparison algorithm.
4.7.2 Executions

A configuration $C$ of size $n$ is a vector of size $n$ that specifies the state of each processor: the state of $p$ is in position $p$ in $C$. A message vector $M$ of size $n$ is a vector of $n$ 4-tuples that specifies the messages sent by each processor in one round: position $p$ in $C$ specifies the messages that $p$ sent to $p$'s east, south, west, and north neighbors. The execution of an election algorithm is a sequence of triples $(C_1, M, C_2)$, where $C_1$ and $C_2$ are configurations, and $M$ is a message vector, all of size $n$. We require all executions $e$ to satisfy several properties. First, the initial configuration in $e$ specifies the identifier of each processor. Second, the second configuration in each triple in $e$ must be the same as the first configuration in the next triple. Third, let state $(p, C)$ denote the state of processor $p$ in configuration $C$. The 4-tuple in position $p$ in $M$ must contain the messages that $p$ sends when $p$ is in state $(p, C_1)$. Finally, $C_2$ must be the configuration after $C_1$ when all the processors receive the messages in $M$. An execution fragment is any finite prefix of an execution.

4.7.3 Chains

Let $R(p,q)$ be a path that connects processors $p$ and $q$. Then the length of $R(p,q)$ is the number of processors in $R(p,q)$, including $p$ and $q$. We use $|R(p,q)|$ to denote the length of $R(p,q)$. Let $k \geq 1$ be an integer. We define $p$'s $k$-diamond to be the set of all processors $q$ such that there exists a path $R(p,q)$ of length $|R(p,q)| \leq k$. Figure 4.7 shows $p$'s $3$-diamond.

Let $P$ be $p$'s $k$-diamond, and $Q$ be $q$'s $k$-diamond. We call $p$ the center of $P$ and $q$ the center of $Q$. Recall that $p_E$ (respectively $p_S$, $p_W$, $p_N$) is $p$'s east (respectively south, west, north) neighbor. Suppose that $e$ is an execution or an execution fragment. Then an east-chain in $e$ for $(P, Q)$ is a subsequence $e_{i_1}, e_{i_2}, \ldots, e_{i_k}$ of $e$ such that the following three conditions are true:

1. There exists a processor $p'$ and a path $R(p,p')$ of length $k$. Path $R(p,p')$ must be of the form $p' p_2 \cdots p_{(k-2)} p_E p$, for some processors $p_2, p_3, \ldots, p_{(k-2)}$.
2. There exists a processor $q'$ and a path $R(q,q')$ of length $k$. Path $R(q,q')$ must be of the form $q' q_2 \cdots q_{(k-2)} q_E q$, for some processors $q_2, q_3, \ldots, q_{(k-2)}$. 

Figure 4.7. Processor p's 3-diamond

(3) Let $p'$ be $p_1$, $p_E$ be $p_{(k-1)}$, $p$ be $p_k$, $q'$ be $q_1$, $q_E$ be $q_{(k-1)}$, and $q$ be $q_k$. Let $p_0$ be some processor adjacent to $p'$, and $q_0$ be some processor adjacent to $q'$. Then for each step $e_{ij}$ in the chain, a message is sent either by processor $p_{(j-1)}$ to processor $p_j$ or by processor $q_{(j-1)}$ to processor $q_j$.

Thus an east-chain for $(P,Q)$ describes combined information flow to $p$ and $q$. We call the chain an east-chain because $p$ or $q$ receives its information from $p_E$ or $q_E$, respectively. We use similar definitions for south-chains, west-chains, or north-chains.

A chain is either an east-chain, a south-chain, a west-chain, or a north-chain. Two $k$-diamonds $P$ and $Q$ are order-equivalent provided that if the identifiers of two processors in $P$ satisfy one of the order
relations $<, =,$ or $>$, then the identifiers of the corresponding processors in $Q$ satisfy the same order relation. Two processors $p$ and $q$ are $k$-equivalent provided that the $k$-diamonds centered at $p$ and $q$ are order-equivalent. If $P$ and $Q$ are two $k$-diamonds, then the states $s$ and $t$ are congruent with respect to $(P, Q)$ provided that $s$ and $t$ are structurally equivalent, and corresponding positions in $s$ and $t$ contain the identifiers of processors in corresponding positions of $P$ and $Q$, respectively.

**Lemma 7.1:** Let $e$ be an execution fragment of a comparison algorithm. Suppose that $k$ is a positive integer. Let $p$ and $q$ be any pair of $k$-equivalent processors, and let $P$ and $Q$ be their respective $k$-diamonds. If there are no chains in $e$ for $(P, Q)$, then at the end of $e$, the states of $p$ and $q$ are congruent with respect to $(P, Q)$.

**Proof:** The proof is by induction on the length of $e$.

**Base:** $|e| = 0$. Neither $p$ nor $q$ has received any messages in $e$, so they will remain in states that are congruent with respect to $(P, Q)$.

**Inductive Step** $|e| > 0$. Assume as the induction hypothesis that the result holds for any execution fragment of length shorter than $|e|$ and all values of $k$. Let $e'$ denote $e$ except for $e$'s last step. Then by the inductive hypothesis, $p$ and $q$ remain in states that are congruent with respect to $(P, Q)$ up to the end of $e'$. Consider what happens at the last step.

**Case 1:** All of the following hold:

(a) Either $p_E$ and $q_E$ are in states that are congruent with respect to $(P, Q)$ just after $e'$, or else neither $p_E$ nor $q_E$ sends a message to the west at the last step of $e$.

(b) Either $p_S$ and $q_S$ are in states that are congruent with respect to $(P, Q)$ just after $e'$, or else neither $p_S$ nor $q_S$ sends a message to the north at the last step of $e$.

(c) Either $p_W$ and $q_W$ are in states that are congruent with respect to $(P, Q)$ just after $e'$, or else neither $p_W$ nor $q_W$ sends a message to the east at the last step of $e$.

(d) Either $p_N$ and $q_N$ are in states that are congruent with respect to $(P, Q)$ just after $e'$, or else neither $p_N$ nor $q_N$ sends a message to the south at the last step of $e$. 
In this case, it is easy to see that \( p \) and \( q \) remain in states that are congruent with respect to \((P,Q)\) after \( e \). For if \( p_E \) and \( q_E \) are in states that are congruent with respect to \((P,Q)\) just after \( e' \), then, since the algorithm is a comparison algorithm, they both make the same decision about whether to send a message to the west at the last step of \( e \). If they both send a message, then the messages they send contain their respective states, which are congruent with respect to \((P,Q)\). A similar argument applies to \( p_S \) and \( q_S \), to \( p_W \) and \( q_W \), and to \( p_N \) and \( q_N \). It follows that \( p \) and \( q \) remain in states that are congruent with respect to \((P,Q)\) after the last step of \( e \).

**Case 2**: Processors \( p_E \) and \( q_E \) are in states that are not congruent with respect to \((P,Q)\) just after \( e' \), and at least one of them sends a message to the west at the last step of \( e \). We will show that this case leads to contradictions.

If \( k = 1 \) (i.e., if \( P \) and \( Q \) consist only of \( p \) and \( q \), respectively), then an east-chain for \((P,Q)\) is produced by the message sent at the last step, a contradiction. So assume that \( k > 1 \). Since \( p \) and \( q \) are \( k \)-equivalent, it follows that \( p_E \) and \( q_E \) are \((k-1)\)-equivalent. Let \( P' \) and \( Q' \) denote their respective \((k-1)\)-diamonds. Since the states of \( p_E \) and \( q_E \) just after \( e' \) are not congruent with respect to \((P,Q)\), they are also not congruent with respect to \((P',Q')\). By the inductive hypothesis, there must be a chain in \( e' \) for \((P',Q')\). Since at least one of \( p_E \) and \( q_E \) sends a message to the west at the last step of \( e \), we obtain an east-chain in \( e \) for \((P,Q)\) by appending this step to \( e' \), a contradiction.

**Case 3**: Processors \( p_S \) and \( q_S \) are in states that are not congruent with respect to \((P,Q)\) just after \( e' \), and at least one of them sends a message to the north at the last step of \( e \). This case is similar to Case 2 and also leads to contradictions.

**Case 4**: Processors \( p_W \) and \( q_W \) are in states that are not congruent with respect to \((P,Q)\) just after \( e' \), and at least one of them sends a message to the east at the last step of \( e \). This case is similar to Case 2 and also leads to contradictions.

**Case 5**: Processors \( p_N \) and \( q_N \) are in states that are not congruent with respect to \((P,Q)\) just after \( e' \), and at least one of them sends a message to the south at the last step of \( e \). This case is similar to Case 2 and
also leads to contradictions. □

Let \( \text{max-east}(e) \) be the maximum \( k \) for which there are order-equivalent \( k \)-diamonds \( P \) and \( Q \) (possibly with \( P = Q \)) such that \( e \) contains an east-chain for \((P,Q)\). The quantities \( \text{max-south}(e) \), \( \text{max-west}(e) \), and \( \text{max-north}(e) \) are defined analogously. Let \( \text{sum}(e) = \text{max-east}(e) + \text{max-south}(e) + \text{max-west}(e) + \text{max-north}(e) \).

Lemma 7.2: Let \( k \) be a positive integer. Assume that mesh \( S \) is such that, initially, every \( k \)-diamond has at least \( i_k \) order-equivalent \( k \)-diamonds. Let \( e \) be any execution fragment of a comparison algorithm in \( S \), and let \( e' \) be another fragment consisting of all but the last step of \( e \). Assume that \( \text{sum}(e') < k \). If some processor \( p \) sends a message in the direction \( \text{dir} \) at the last step of \( e \), then there are at least \( i_k \) processors that send a message in the direction \( \text{dir} \), where \( \text{dir} \in \{\text{east}, \text{south}, \text{west}, \text{north}\} \).

Proof: Processor \( p \) has at least \( i_k \) \( k \)-equivalent processors (including \( p \) itself). Let \( q \) be one of these processors, and let \( P \) and \( Q \) be the \( k \)-diamonds centered at \( p \) and \( q \), respectively. By the definition of \( \text{sum}(e') \), there can not be a chain in \( e' \) for \((P,Q)\). By Lemma 7.1, processors \( p \) and \( q \) are congruent with respect to \((P,Q)\) at the end of \( e' \). By the definition of a comparison algorithm, \( q \) also sends a message in the direction \( \text{dir} \) at the last step of \( e \). □

Lemma 7.2 is also true when \( \max(\text{max-east}(e'), \text{max-south}(e'), \text{max-west}(e'), \text{max-north}(e')) \leq k \).

4.7.4 Replication symmetry

We will prove the lower bound on the number of messages that comparison algorithms require for election in \( n \)-meshes when \( n \) is a power of 4. We first assign identifiers that have a large amount of replication symmetry to the processors in an \( n \)-mesh \( S \).

Define a \( k \)-moddiamond \( A \) to be the set of all processors on the boundary of or enclosed by the figure in Figure 4.8. We say that the square \( abcd \) (in that order) is the center of \( A \). Consider any \( n \)-mesh \( S \), where \( n = 4^l \), for some \( l \geq 2 \). Let \( A_1 \) be a \((2^{l-1}) - 1\)-moddiamond in \( S \). Then the remaining processors in \( S \) form another \((2^{l-1}) - 1\)-moddiamond \( A_2 \) in \( S \).
Figure 4.8. A $k$-moddiamond
We will assign identifiers to the processors as follows:

(1) Assign a 1(0) to the least significant digit of all the processor identifiers in \( A_1(A_2) \).

(2) In what follows we consider \( A_1 \). The same should be done for \( A_2 \). Create 4 \((2^{l-2} - 1)\)-moddiamonds \( A(1,0), A(1,1), A(1,2), A(1,3) \) with centers \( a_0 b_0 c_0 d_0, a_1 b_1 c_1 d_1, a_2 b_2 c_2 d_2 \), and \( a_3 b_3 c_3 d_3 \), respectively, where there are

- \( 2^{l-2} + 1 \) processors between \( a_0 \) and \( a \) (including \( a_0 \) and \( a \)), and \( a_0 \) is to the north of \( a \).
- \( 2^{l-2} + 1 \) processors between \( a_1 \) and \( a \) (including \( a_1 \) and \( a \)), and \( a_1 \) is to the east of \( a \).
- \( 2^{l-2} + 1 \) processors between \( a_2 \) and \( a \) (including \( a_2 \) and \( a \)), and \( a_2 \) is to the south of \( a \).
- \( 2^{l-2} + 1 \) processors between \( a_3 \) and \( a \) (including \( a_3 \) and \( a \)), and \( a_3 \) is to the west of \( a \).

Assign a 0(respectively 1, 2, 3) to the second least significant digit of processor identifiers in \( A(1,0) \)(respectively \( A(1,1), A(1,2), A(1,3) \)).

(3) Recursively create 16 \((2^{l-3} - 1)\)-moddiamonds \( A(1,0,0), A(1,0,1), A(1,0,2), A(1,0,3), A(1,1,0), \cdots, A(1,3,3) \).

For each digit \( x \) in \( \{0, 1, 2, 3\} \), assign a 0(respectively 1, 2, 3) to the third least significant digit of processor identifiers in \( A(1,x,0) \)(respectively \( A(1,x,1), A(1,x,2), A(1,x,3) \)).

(4) Let \( i \geq 4 \). Recursively create \( 4^{i-1} \) \((2^{l-i} - 1)\)-moddiamonds and assign a 0, 1, 2, or 3 to the \( i \)th least significant digit of processor identifiers. The recursion stops after creating 1-moddiamonds and assigning the appropriate digits to the processor identifiers in the 1-diamonds. We then assign the most significant digits to each processor in the 1-moddiamonds as in Figure 4.9. Figure 4.10 shows the processor identifiers in a 64-mesh.

4.7.5 Proof of the lower bound

Theorem 7.1: Assume that \( n \) is \( 4^l \), and \( l \geq 2 \). Let \( \Pi \) be a comparison algorithm that elects a leader in every synchronous \( n \)-mesh. Then there is an execution \( e \) of \( \Pi \) that uses at least \( \frac{57}{32} n \) messages.

Proof: Consider the \( n \)-mesh \( S \) whose processor identifiers are assigned as described in Section 4.7.4. Let \( messages(e) \) denote the number of messages that \( e \) uses in \( S \). We will prove the theorem by showing that
First, we show that \( \text{messages}(e) \geq n \). Initially, no processor knows the identifier of any other processor. Thus the state of every processor \( p \) is order-equivalent to the state of every other processor. Since \( \Pi \) is a comparison algorithm, every processor sends a message to the east (respectively south, west, north) if, and only if, \( p \) sends a message to the east (respectively south, west, north). Thus \( \text{messages}(e) \geq n \).

Second, we show that \( \text{messages}(e) \geq n + \frac{n}{2} \). Let \( 0 \) be the first round in \( \Pi \) in which \( p \) sends a message. In round 0, if \( p \) sends more than one message, then every processor sends more than one message, and so \( \text{messages}(e) \geq 2n \). Thus suppose that \( p \) sends only one message in round 0. Without
loss of generality, suppose that $p$ (hence every processor) sends the message to the east. Recall that $p_w$ is $p$'s west neighbor, that $\text{state}(p, i)$ is the state of $p$ at the start of round $i$, and that the state of a processor at the end of round $i$ is the same as the state of the processor at the start of round $i+1$. Then $\text{state}(p, 1) =$
(ID(p),nil,nil,ID(pw),nil). At the end of round 0, there are two types of states. The state \((p, 1)\) is of the first type if \(ID(pw) < ID(p)\). The state \((p, 1)\) is of the second type if \(ID(pw) > ID(p)\). Suppose that state \((p, 1)\) is of the first type, and state \((q, 1)\) is of the second type. The state of every processor at the end of round 0 is order-equivalent to either \(p\) or \(q\). By the way we assigned identifiers to processors, it suffices to compare the most significant digits of the identifiers to determine whether \(ID(pw) < ID(p)\).

Thus there are \(\frac{n}{2}\) processors whose states are order-equivalent to state \((p, 1)\), and there are \(\frac{n}{2}\) processors whose states are order-equivalent to state \((q, 1)\). Hence \(P\) cannot terminate at the end of round 0. Let \(i_1\) be the first round, after round 0, in which a processor sends a message. Then there will be at least \(\frac{n}{2}\) messages sent in \(i_1\) in the same direction. Thus \(messages(e) \geq n + \frac{n}{2}\).

Third, we show that \(messages(e) \geq n + \frac{n}{2} + \frac{n}{4}\). If both \(p\) and \(q\) send messages in round \(i_1\), then all the processors send messages in round \(i_1\), and so \(messages(e) \geq 2n\). If either \(p\) or \(q\) sends more than one message in round \(i_1\), then at least \(\frac{n}{2}\) processors send more than one message in round \(i_1\), and so \(messages(e) \geq 2n\). Thus consider what happens when either \(q\) or \(p\) sends only one message in round \(i_1\).

We will show that the state of every processor at the end of round \(i_1\) is order-equivalent to the states of at least \(\frac{n}{4}\) processors.

Case 1: Suppose that \(p\) sends only one message to the direction \(dir\), and \(q\) does not send any message, where \(dir \in \{east, south, west, north\}\). There are three subcases:

Case 1.1: \(dir = east\). Then state \((p, i_1 + 1) = state(p, 1)\). Thus there are \(\frac{n}{2}\) processors whose states are order-equivalent to state \((p, i_1 + 1)\), namely, the processors whose states were congruent to state \((p, 1)\) at the beginning of round 1. On the other hand, by the way identifiers are assigned to processors, it suffices to compare the most significant digits of the identifiers to show that state \((q, i_1 + 1)\) is order-equivalent to the states of \(\frac{n}{4}\) processors. Since the state of every processor at the beginning of round 0 is order-
equivalent to either state \((p, 1)\) or state \((q, 1)\), the state of every processor is order-equivalent to the state of at least \(\frac{n}{4}\) processors at the end of phase \(i_1\).

Case 1.2: \(\text{dir} = \text{south}\). Then state \((q, i_1+1) = state \((q, 1)\). Thus there are \(\frac{n}{2}\) processors whose states are order-equivalent to state \((q, i_1+1)\), namely those processors whose states were congruent to state \((q, 1)\) at the beginning of round 1. On the other hand, by the way identifiers are assigned to processors, it suffices to compare the most significant digits of the identifiers to show that state \((p, i_1+1)\) is order-equivalent to the states of \(\frac{n}{4}\) processors. Thus the state of every processor is order-equivalent to the state of at least \(\frac{n}{4}\) processors at the end of phase \(i_1\).

Case 1.3: \(\text{dir} = \text{west}\) or \(\text{dir} = \text{north}\). By the way identifiers are assigned to processors, it suffices to compare the most significant digits of the identifiers to show that state \((p, i_1+1)\) is order-equivalent to the states of \(\frac{n}{4}\) processors, and state \((q, i_1+1)\) is order-equivalent to the states of \(\frac{n}{4}\) processors. Thus the state of every processor is order-equivalent to the state of at least \(\frac{n}{4}\) processors at the end of phase \(i_1\).

Case 2: Suppose that \(q\) sends only one message to the direction \(\text{dir}\), and \(p\) does not send any message, where \(\text{dir} \in \{\text{east}, \text{south}, \text{west}, \text{north}\}\). This case is similar to Case 1.

Let \(i_2\) be the first round, after round \(i_1\), in which a processor sends a message. Since the state of every processor is order-equivalent to the state of at least \(\frac{n}{4}\) processors at the end of phase \(i_1\), there will be at least \(\frac{n}{4}\) messages sent in \(i_2\) in the same direction. Hence \(\text{messages}(e) \geq n + \frac{n}{2} + \frac{n}{4}\).

Finally, we show that \(\text{messages}(e) \geq n + \frac{n}{2} + \frac{n}{4} + \frac{n}{32}\). As we explained, if a processor sends more than one message in rounds \(0, i_1,\) or \(i_2\), then \(\text{messages}(e) \geq 2n\). Thus suppose that all processors send only one message in each of these rounds. Let \(i_3\) be the first round, after round \(i_2\), in which a processor sends a message. Let \(e'\) be the execution fragment consisting of all the rounds up to, and including, round \(i_3-1\). By the definitions of \(\text{sum}(e'), i_1, i_2,\) and \(i_3, \text{sum}(e') \leq 3\). By the way identifiers are
assigned to processors, it suffices to compare the two most significant digits of the identifiers to show that every 4-diamond has \( \frac{n}{32} \) order-equivalent 4-diamonds. By Lemma 7.2, there are at least \( \frac{n}{32} \) messages sent in round \( i_3 \). Thus, \( \text{messages}(e) \geq n + \frac{n}{2} + \frac{n}{4} + \frac{n}{32} \).

In the proof of Theorem 7.1, we used only the two most significant digits of the processor identifiers. In the future, we hope to use all of the digits to yield a better lower bound.
CHAPTER 5
FUTURE WORK

In this chapter, I present some of the problems I hope to solve in the future.

5.1 Problems Associated with Chapter 2

An important problem is to generalize the results of the paper "Memory requirements for agreement among unreliable asynchronous processes" [2] to the case when a memory cell’s value spontaneously oscillates between several values. There are two reasons to study memory oscillation. First, memory oscillation models the situation where some faulty cells are repaired on-line while some processes may be accessing them. Requiring such processes to abort their executions and restart their algorithms would be inefficient and inconvenient. Second, by examining memory oscillation, I wish to study whether cheap but possibly faulty memory can be used in place of expensive but reliable memory without significant performance degradation. Therefore, I plan to formulate lower and upper bounds on the number of three-valued cells needed in test-and-set agreement protocols when processes die and memory cells fail undetectably. Elaborate coding schemes do not immediately solve this problem since the processes cannot set more than one cell at a time, and the processes may die at any step. For the lower bounds, I will generalize the concept of computation graphs [2] to handle memory failure. A faulty cell will be modeled as a reliable cell that a Byzantine process p spontaneously writes. Process p does not access any other cell. For the upper bounds, I intend to build on Jaffe’s techniques [31].

5.2 Problems Associated with Chapter 3

Consider a general network in which each edge has a weight that corresponds to the cost of sending a message along the edge. Minimum weight spanning trees of the network are used for broadcasting in computer networks with point-to-point links. Tsin [48] presented a distributed algorithm for updating a minimum spanning tree when a new vertex is added to the underlying graph. I hope to study the problem of constructing a minimum-weight spanning tree on the network when some of the network links fail intermittently. Building on the work of Korach, Moran, and Zaks [33], I will first consider the simpler
problem of constructing spanning trees (not necessarily minimum-weight) on general networks with faulty links. An algorithm that solves the simpler problem will be an algorithm for election in general networks with faulty links, since the root of a spanning tree will be the leader of the network. Next, by using the techniques of Frederickson [21], Gallager, Humblet, and Spira [24], and Tsin [48], I expect to generalize the solution of the simpler problem to the problem of constructing minimum-weight spanning trees when some of the network links may be faulty.

*Deadlock detection is a fundamental* problem in distributed databases and distributed operating systems. *Deadlock occurs when a cycle of* \(n\) *processes* \(P_0, P_1, \ldots, P_{n-1}, P_0\) *forms such that each* \(P_i\) *waits for a resource held by* \(P_{(i+1)\mod n}\). *I plan to construct deadlock detection algorithms for asynchronous complete networks with Byzantine links. Complete networks in this case may be either physical networks or virtual networks at the session layer of the OSI model of computer networks. I expect to use the techniques of Chandy, Misra, and Haas [10] and Awerbuch and Micali [6].

### 5.3 Problems Associated with Chapter 4

Peterson [41] proposed an efficient algorithm for election in reliable asynchronous meshes. Although the algorithm uses \(O(n)\) messages, which is asymptotically optimal, the constant in the big \(O\) notation is large. Since the number of messages that an algorithm uses is independent of machine implementations, it is desirable to have a small constant. I plan to improve the lower and upper bounds on the number of messages for *synchronous* meshes as a first step in deriving a lower bound and a matching upper bound on the number of messages in *asynchronous* meshes. The analysis of the lower bound given in Chapter 4 can be improved in two ways. First, I hope to have a better analysis of the number of messages when the processor identifiers are distributed as in Section 4.7.4. Second, I plan to obtain a distribution of processor identifiers that yields a better lower bound. On the other hand, the upper bound in Chapter 4 can be improved by using the solutions to the *firing squad* problem [26], [38], [50] to efficiently remove the requirement that all the processors start the algorithm simultaneously.
REFERENCES


VITA

Hosame Hassan Abu-Amara obtained his B.S. degree from the University of California, Berkeley, in March 1983. He received his M.S. degree in May 1985, and he will obtain his Ph.D. degree in October 1988; both degrees are from the University of Illinois at Urbana-Champaign. In August 1988, he will join IBM's Thomas J. Watson Research Center in Hawthorne, New York, for a post-doctoral position.

Hosame Abu-Amara's research interests are in computer engineering, particularly fault-tolerance, distributed and parallel algorithms and systems, computational complexity, and computer organization.