INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Fault-tolerant distributed algorithms for consensus and termination detection

Wu, Li-Fen, Ph.D.
The Ohio State University, 1994
FAULT-TOLERANT DISTRIBUTED ALGORITHMS FOR CONSENSUS AND TERMINATION DETECTION

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Li-Fen Wu, B.S., M.S.

The Ohio State University

1994

Dissertation Committee:
Professor Ten-Hwang Lai
Professor Ming-Tsan Liu
Professor Mukesh Singhal

Approved by
Adviser
Department of Computer and Information Science
To My Parents
ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Professor Ten-Hwang Lai, for his guidance, advice, support, and encouragement over these years.

My special thanks are to Professor Ming-Tsan Liu and Professor Mukesh Singhal for being in my Dissertation Defense Committee. They have helped me not only at the committee, but also throughout my graduate school study years.

I would like to thank Jeff Ward for his constructive criticisms and suggestions that have considerably improved this work.

I also appreciate the staff in the Department of Computer and Information Science. Their kind assistance made the study here a smooth and pleasant experience. Special thanks are due to Tom Fletcher, Elley Quinlan, and Carl Phillips.

All my family members deserve the greatest merits and special thanks. They have been always very supportive during my graduate study at the Ohio State University. Abundant credits should go to my husband, for his love, support, understanding and sacrifice.

My thanks shall go to all my friends here in Columbus, Ohio, who have shared my ups and downs during the past years. In particular, my special thanks are due to Ming-Jye Sheng, Shu-Hwa Sarah Yu, Ing-Miin Hsu, and Shu-Shang Sam Wei for their persistent support and encouragement. I would like to thank Dan Oblinger,
John Cameron, and Deborah Shands for their kindness to help me to adjust to the new environment.

Last but not least, I am also greatly indebted to Dr. Rong Chang, my manager at IBM, for his understanding and encouragement.
VITA

March 28, 1965 ....................................................... Born: Taiwan, Republic of China

Sept. 1983 - June 1987 ........................................... B.S., Department of Computer Science
and Information Engineering
National Taiwan University, Taipei, Taiwan

Sept. 1987 - Mar. 1989 ......................................... M.S., Department of Computer and Information Science
The Ohio State University, Columbus, Ohio

Mar. 1989 - Dec. 1993 ................................. Graduate Teaching Associate, Department of Computer and Information Science
The Ohio State University, Columbus, Ohio

and Multi-media Services Division
Advantis, White Plains, New York

June 1994 - present ....................... Software Developer, Network Application Services Division
IBM, White Plains, New York

Publications


Fields of Study

Major Field: Computer and Information Science

Studies in:

Parallel and Distributed Computing Prof. Ten-Hwang Lai
Computer Architecture and Networking Prof. Ming-Tsan Liu
Computer Graphics Prof. Wayne Carlson
# List of Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>127</td>
</tr>
</tbody>
</table>

| Performance Measures | 127 |
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Context of Distributed Computing.</td>
</tr>
<tr>
<td>2</td>
<td>State Transition Graph of Basic Process ( \gamma_i ).</td>
</tr>
<tr>
<td>3</td>
<td>Process ( \alpha_i ) of Consensus Protocol ( \alpha ).</td>
</tr>
<tr>
<td>4</td>
<td>The Protocol for Process ( i, 1 \leq i \leq n ).</td>
</tr>
<tr>
<td>5</td>
<td>State Transition of a Basic Process.</td>
</tr>
<tr>
<td>6</td>
<td>Dijkstra-Scholten Algorithm for ( p_i, 1 \leq i \leq n ).</td>
</tr>
<tr>
<td>7</td>
<td>Algorithm for ( p_i, 1 \leq i \leq n ).</td>
</tr>
<tr>
<td>8</td>
<td>An Example.</td>
</tr>
<tr>
<td>9</td>
<td>A Counter Example.</td>
</tr>
</tbody>
</table>
# Table of Contents

DEDICATION ........................................................................................................................ ii

ACKNOWLEDGMENTS .......................................................................................................... iii

VITA ........................................................................................................................................ v

LIST OF TABLES .................................................................................................................. vii

LIST OF FIGURES .............................................................................................................. viii

CHAPTER PAGE

I INTRODUCTION ........................................................................................................... 1

1.1 Contributions of Research ................................................................................ 4
  1.1.1 The Two Proposed Services ........................................................................ 5
  1.1.2 TD Is Harder Than CON ........................................................................... 6
  1.1.3 A Fault-Tolerant Distributed TD Algorithm ........................................... 7
  1.1.4 Other Applications ..................................................................................... 9
  1.2 Organization of Dissertation ........................................................................... 9

II Survey of the Literature .......................................................................................... 12

2.1 Distributed Computing .................................................................................. 13
2.2 Fault-Tolerant Distributed Computing ......................................................... 15
2.3 The Consensus Problem ................................................................................ 20
2.4 The Termination Detection Problem ............................................................. 21

III Consensus and Termination Detection ................................................................. 25

3.1 The Traditional Model .................................................................................... 25
3.2 Problem Descriptions ..................................................................................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1 The Consensus Problem</td>
<td>29</td>
</tr>
<tr>
<td>3.2.2 The Termination Detection Problem</td>
<td>31</td>
</tr>
<tr>
<td>3.3 Impossibility of CON and TD</td>
<td>32</td>
</tr>
<tr>
<td>3.4 Conclusion</td>
<td>38</td>
</tr>
<tr>
<td>IV Fault-Detecting and Fault-Flushing Services</td>
<td>39</td>
</tr>
<tr>
<td>4.1 Motivation</td>
<td>40</td>
</tr>
<tr>
<td>4.2 The Proposed Services</td>
<td>41</td>
</tr>
<tr>
<td>4.3 Service Provider of S1 and S2</td>
<td>44</td>
</tr>
<tr>
<td>4.3.1 Service S1 is impossible in $S$</td>
<td>45</td>
</tr>
<tr>
<td>4.3.2 Service S2 is impossible in $S^1$</td>
<td>46</td>
</tr>
<tr>
<td>4.4 Fault-Detecting Service to Consensus</td>
<td>48</td>
</tr>
<tr>
<td>4.4.1 An N-Resilient Consensus Protocol</td>
<td>49</td>
</tr>
<tr>
<td>4.5 TD Is Harder Than CON</td>
<td>56</td>
</tr>
<tr>
<td>4.6 Discussion</td>
<td>59</td>
</tr>
<tr>
<td>4.7 Conclusion</td>
<td>61</td>
</tr>
<tr>
<td>V An $(N-1)$-Resilient Distributed Termination Detection Algorithm</td>
<td>63</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>64</td>
</tr>
<tr>
<td>5.2 The Model</td>
<td>67</td>
</tr>
<tr>
<td>5.2.1 The Underlying Communication Network</td>
<td>67</td>
</tr>
<tr>
<td>5.2.2 The Problem</td>
<td>70</td>
</tr>
<tr>
<td>5.3 Previous Work</td>
<td>74</td>
</tr>
<tr>
<td>5.3.1 Dijkstra-Scholten Algorithm and Its Variants</td>
<td>74</td>
</tr>
<tr>
<td>5.3.2 A Fault-Tolerant Termination Detector</td>
<td>78</td>
</tr>
<tr>
<td>5.4 $(N-1)$-Resilient Algorithm</td>
<td>79</td>
</tr>
<tr>
<td>5.4.1 Problems Caused by Faulty Processes</td>
<td>79</td>
</tr>
<tr>
<td>5.4.2 Algorithm Description</td>
<td>82</td>
</tr>
<tr>
<td>5.4.3 Postponed Start</td>
<td>92</td>
</tr>
<tr>
<td>5.5 Correctness Proof</td>
<td>94</td>
</tr>
<tr>
<td>5.5.1 Message Counting and Process Status</td>
<td>97</td>
</tr>
<tr>
<td>5.5.2 Structure of Engaged Processes</td>
<td>100</td>
</tr>
<tr>
<td>5.5.3 Completeness</td>
<td>105</td>
</tr>
<tr>
<td>5.5.4 Soundness</td>
<td>109</td>
</tr>
<tr>
<td>5.6 Performance Analysis and Comparison</td>
<td>123</td>
</tr>
<tr>
<td>5.7 Conclusion</td>
<td>127</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

Distributed computing provides a means for sharing information and resources that are widely dispersed around the world. As a result of sharing, machines and bandwidths are more fully utilized, and, users achieve improved job throughput with greatly reduced costs, including both time and capital investments. The importance of distributed computing is readily apparent when one observes, for example, the tremendous growth of the Internet.

As should be expected, the issue of fault-tolerance becomes absolutely critical in distributed computing due to the fact that an increasing amount of resources, hardware and software, interrelate with each other. One tiny process failure may hamper other processes and cause the whole distributed system to behave unexpectedly. Hence, the general objective of ensuring that a distributed system will perform reliably and efficiently in spite of potential failures by individual nodes or components in the system has received much attention.

Designing fault-tolerant distributed applications is a non-trivial task, particularly when the knowledge about the relative process speed and communication delay is not
known to the applications. Some problems have been proven to be unsolvable [43], and some are known to be difficult to solve [7,22] when the environment provided by the system is totally asynchronous. This holds true even when considering the most benign type of failure. The difficulty stems from the impossibility of distinguishing between remote process failure and message delay.

The consensus problem [43] is such a classical unsolvable problem in (totally asynchronous) distributed computing. The problem is so important because it is a fundamental building block for any coordination activity such as the synchronization of physically dispersed clocks and the election of decision making leader(s).

Another important problem existing in distributed computing is the problem of termination detection. When a task is designed to be accomplished by a set of processes cooperating in a distributed manner, knowing when the task is completed is essential but non-trivial.

Intuitively, it may seem that knowing when a task is finished (or knowing when every process involved is in the “terminated” state) is a specialized consensus in which every process decides on the same decision (the terminated state). It is the relationship of these two problems that attracted us to further study on the issues regarding fault-tolerant distributed computing. We were interested in knowing whether any environment that provides the services sufficient for the solution of the consensus problem also provides the services sufficient for the solution of the termination
problem, or vice versa.

In this work we study which services the system should provide to the application environment so that the consensus problem can be solved. It turns out that a new environment (with fault-detecting capability) provides the service sufficient for solving the consensus problem. Surprisingly, this new environment does not seem to provide the services sufficient for solving the termination detection problem.

As a result, we then become interested in the possibility of proving that the intuitive view of the relationship of consensus and termination detection is not correct. We achieve this objective, first, by proving that the problem of consensus can be reduced to the problem of termination detection, and, subsequently, by proving that the problem of termination detection is indeed (strictly) harder than the problem of consensus.

Based on these results, we endeavor to find a service stronger than the one needed for the consensus problem, so that the problem of termination detection will also be solvable in a totally distributed application environment. We identify such a service in the fault-flushing service. An efficient fault-tolerant distributed termination detection algorithm is then constructed based on the proposed services of fault-detecting and fault-flushing.

The results established in this dissertation are summarized in the following section. We believe that the proposed system services will be useful in designing fault-tolerant
distributed algorithms.

1.1 Contributions of Research

In this research, we focus on the study of problems concerning fault-tolerant distributed systems. Particularly, the following are accomplished:

1. Two system services, fault-detecting and fault-flushing, are identified and proposed to facilitate the design of fault-tolerant distributed algorithms.

2. The relationship between the consensus problem and the termination detection problem, two of the most fundamental problems in distributed computing, is established. Contrary to the intuitive view that the problem of termination detection is nothing but reaching a special state of agreement, the termination detection problem is in fact shown to be harder than the consensus problem.

3. Based on the two services proposed, a fault-tolerant termination detection algorithm is constructed and its complexity is shown to be better than the existing one that is known to us.

4. In addition to the consensus problem and the termination detection problem, the proposed system services are also demonstrated to be useful to the design of some other distributed applications, namely, the order process renaming problem and the backup process recovery problem.
1.1.1 The Two Proposed Services

Fault tolerance is one of the most desired properties of distributed systems. While some problems are widely known to be unsolvable in the presence of faulty processes [43], many distributed algorithms are in fact able to tolerate process failures. When a termination detection algorithm is applied to a fault-tolerant distributed system, it is desirable that the algorithm itself also be fault-tolerant.

As the problem of distributed termination detection is known to be unsolvable in a totally asynchronous application environment that is subject to failures, new system services are required to provide system-specific knowledge to the applications to make the existence of a termination detector possible. The following two services are proposed:

$S_1$: Fault-Detecting Service.

$S_2$: Fault-Flushing Service.

These two services are shown to be sufficient for the solution of the termination detection problem in a totally asynchronous application environment.

It would be easier if these two services could be rendered in the application environment. Unfortunately, we found it impossible to realize these services in a totally asynchronous application environment. Instead, we found that the system (or the transport mechanism) is a more reasonable platform to be their service provider. (In
the network level, it is easier to have access to some system services of a synchronous nature such as timeout mechanisms.)

The issue of the importance of the fault-detecting service for fault tolerance has attracted a substantial amount of research [24,25,83]; the fault-flushing service, on the other hand, has received little attention from researchers. This service not only helps to provide more specific communication common knowledge, it may also increase the performance of an application because messages that may carry consistent and productive results are usable by the receiver before it decides whether to take some action based upon some process's failure.

1.1.2 TD Is Harder Than CON

The problem of reaching an agreement and the problem of detecting termination are two of the most fundamental problems in distributed computing. A prevailing view of the relationship between the consensus problem and the termination detection problem is that the latter has been recognized as a specialized consensus problem in the sense that every nonfaulty termination detection process has to cooperate, and then at least one of them has to eventually make a decision as to when the basic application has finished its processing.

Contrary to the intuitive view, we have established that the problem of termination detection is harder than the consensus problem. We established this conclusion through three results obtained in this work:
1. The fault-detecting service is sufficient for the consensus problem to be solvable in a totally asynchronous application environment.

2. The fault-detecting service alone is not sufficient for the termination detection problem to be solvable in a totally asynchronous application environment.

3. The problem of consensus can be reduced to the problem of termination detection. In other words, no matter what services the system provides, as long as there is an algorithm for the problem of termination detection, there is an algorithm (transformed from the termination detection algorithm) that solves the problem of consensus.

With these three observations, we conclude that termination detection is strictly harder than consensus.

1.1.3 A Fault-Tolerant Distributed TD Algorithm

Based on the two proposed services, a fault-tolerant termination detector that outperforms an existing one is constructed. This algorithm is based on a variant (the LTD variant [61]) of the well-known Dijkstra-Scholten (DS) algorithm [38]. It is able to tolerate any number of crash failures. It runs as efficiently as its non-fault-tolerant predecessor if no process actually fails during the computation, and otherwise incurs only a small amount of cost for each actual failure.
In a recent paper [103], Venkatesan proposed a fault-tolerant termination detector, called the V algorithm. The V algorithm is based on a previous non-fault-tolerant version, the CV algorithm [26], and can be $k$-resilient for any prespecified value $k$, $1 \leq k \leq n - 1$. (A protocol or distributed algorithm is said to be $k$-resilient if it works correctly even in the presence of $k$ faulty processes.) The V algorithm requires $\Omega(kM)$ control messages to prepare itself for possible process failures, where $M$ is the total number of basic messages (i.e., messages of the basic system), even if no failure actually occurs during the basic computation. Should failures occur, the algorithm requires about $O(kM/n)$ additional control messages per failure, where $n$ is the total number of processes in the basic application; this cost may be intolerably high in a message-intensive computation. It also requires processes to send multiple messages in an atomic way.

Our algorithm has a very interesting property: it achieves fault tolerance without message replication and thereby avoids all preparation cost (in terms of control messages); if no process ever fails during basic computation, our algorithm is as efficient as its non-fault-tolerant predecessor, which is worst case message-optimal, $O(M + n)$. Recall that the V algorithm needs $\Omega(kM)$ control messages to prepare for possible faults. Our algorithm’s property in this regard is certainly desirable, since in today’s systems it seems that most computations are done without a process failure. Our algorithm also improves over the V algorithm in several other aspects: 1) should a
process fail during the computation, our algorithm recovers with at most $O(n)$ control messages, which is in most cases smaller than the V algorithm's $O(M)$ (assuming $k = n - 1$, as in our case); 2) the space complexity is $O(n^2)$ as compared with $O(nM)$ for the V-algorithm (as $M$ is not known at compile time, the V algorithm requires the more expensive dynamic memory allocation); 3) our algorithm has a shorter detection delay of $O(n)$ as compared with the $O(M)$ required by the V algorithm; and 4) our algorithm does not need the service of sending multiple messages atomically, which, in contrast, is crucial for the V algorithm.

1.1.4 Other Applications

In addition to the problems of consensus and termination, two applications are presented in Chapter VI to demonstrate that the proposed services considerably facilitate the job of designing fault-tolerant algorithms.

First, the fail-detecting service is shown to make the process renaming problem easier to solve without the strong constraint on the size of naming space. Then, the fail-flushing service is shown to be critical to solving the backup process recovery problem in a totally asynchronous system.

1.2 Organization of Dissertation

The rest of this dissertation is organized as follows:

Chapter II contains a broad survey of areas on distributed computing and fault-
tolerant distributed computing. A survey of the two important focus problems of this dissertation, the consensus problem and the termination detection problem, is also given.

In Chapter III, the (traditional) general system model for the issues to be studied is presented. The two focus problems — the consensus problem and the termination detection problem — are also described. We further confirm in this chapter the belief that termination detection is not solvable in a totally asynchronous system by showing that termination detection is at least as hard as consensus.

Chapter IV is primarily concerned with what services a distributed system should provide for a fault-tolerant distributed application and where in the system the services should be rendered so that distributed applications can be resilient to failures even when some system parameters, such as the relative process speed and communication delay, are not known a priori.

From the observations of the effect of the proposed services to the consensus problem and the termination detection problem, we are able to show that the termination detection problem is harder than the consensus problem.

In Chapter V, a fault-tolerant distributed termination detection algorithm is constructed based on the services proposed in Chapter IV. Chapter VI presents and considers two more applications, the process renaming problem and the backup process recovery problem, to demonstrate further the usefulness of the proposed services
in achieving fault-tolerance in distributed computing.

Chapter VIII concludes this dissertation and suggests future research directions.
As faster, cheaper processors and better communication technology become available, distributed computing systems are becoming more and more important in today's information age. Due to the increasing amount of hardware and software participating in a distributed computing system, ensuring that a distributed system still performs efficiently and with high reliability is not a trivial task. Hence, the general objective of ensuring that a distributed system will perform reliably and efficiently in spite of potential failures by individual nodes or component in the system has received much attention.

In this chapter, we first survey previous work done in the areas of distributed computing and fault-tolerant distributed computing. As the consensus problem and termination problem are two of the most fundamental problems in distributed computation, and they are both surprisingly intriguing when the distributed system is subject to failure, previous research effort focused on these two problems are also examined. The relation of these two problems will be discussed in a later chapter for further understanding the proposal of the fail-flushing services.
2.1 Distributed Computing

A distributed computing system [94] consists of a collection of autonomous computers that interact through an underlying communication network [97]. With the gained distributed control, it is easier than ever to achieve higher performance/throughput, increased fault-tolerance capability, and more involved resource/information sharing.

However, there does have a very fundamental problem inherent in almost every distributed system; namely, the problem of lack of instantaneous global state. To cope with this, the clock synchronization algorithm [96] has been proposed to virtually support a physically synchronized clock. Logical clocks [65,73] are also frequently deployed to derive casual ordering among remote events. Algorithms devoted to report a global state (snapshot) [27,51,62,88] have often been used as the basic skeleton for the other distributed algorithms [54].

The nature and problems encountered in designing distributed algorithms are analyzed in [82]. Three issues are concerned: 1) channel properties—such as transmission ordering, transmission reliability, and message delay, 2) network topology, and 3) knowledge initially owned by a process—such as neighbor information, own identity, or global structure.

Many distributed algorithms have been devised in the literature. They are categorized into the following three main areas [82]:

- function computing: e.g., minimum spanning tree [45], depth-first-search [89],
and global function computation [46].

- observation control: e.g., global snapshot [27,51,62,88], mutual exclusion [13,48,65,81,84], and stable property detection [52] such as deadlock detection [28,55,85,93], and termination detection [38].

- protocols: e.g., election [2,20,92], agreement [15,42], atomicity ensuring [19], and fairness [33].

A special category of distributed algorithm is called the total algorithm [98] where the participation of all processes in the distributed system is required before a decision can be taken. It has been shown that for many network control problems, such as propagation of information with feedback, resynchronization, and distributed infimum computation, that an algorithm solving the problem is necessarily total, and that any total algorithm can solve the problem. More topics on distributed algorithms related to total algorithms can be found in [99].

If algorithms solving a distributed problem are necessarily total, then in a situation where processes are prone to failure, the problem becomes unsolvable (or difficult to solve) due to the lack of participation from the failed process. Fault tolerance, thus, becomes a focusing issue in distributed computing.
2.2 Fault-Tolerant Distributed Computing

A long standing problem in the field of distributed computing is that of designing algorithms which can tolerate failure of processes. Fault-tolerant distributed computing has been, and remains an active research topic in computer information science. A large number of techniques have been progressively introduced over a long period of time.

As a multitude of hardware and software is involved in a distributed computing system, the system is subject to failures that may lead it to behave unexpectedly. The unpredictable nature of fault occurrence makes design of a fault-tolerant distributed system difficult and complicated.

The technique most often chosen to provide fault tolerance is replication (or redundancy). Levy and Silberschatz [69] improved availability of files by replications. Nieuwenhuis [79] found an optimal reliable allocation mapping for a replicated process system.

When replication is applied, maintaining consistency among all replicas becomes the next important concern. A checkpointing technique is usually used to fulfill this goal. Son and Agrawals [95] used two phase commit with a temporary commit area for transactions intervened with the checkpoint or recovery. Nicola and Spanje [78] compared checkpointing policies to determine which one optimizes a certain performance measure.
There are many fault-tolerant algorithms available in the literature, such as reliable broadcast \[19,30,47,57,87\], critical section management \[11\], and spanning tree (or forest) construction \[45,59\]. Fault-tolerant leader election algorithms are presented in \[1,12,23\]. However, different system environments are based upon different assumptions: Bar-Yehuda and Kutten \[12\] assumed that channels are non-FIFO, that faults are not detectable, and that each processor knows its ID and a set of majority processes. Chan and Chin \[23\] assumed a node recovers without remembering what it has done on a ring with FIFO transmission, and Abu-Amara \[1\] assumed only links can fail. Both works of Lamport \[66\] and Halpern et al. Lspace \[49\] are fault-tolerant clock synchronization algorithms under the malicious process failure model. While the former assumed message delay is zero, the latter only assumed that message delay is bounded. Wu and Fuchs \[104\] provided fault tolerance to distributed shared memory.

In this dissertation, we only consider the process crash failure where a process stops immediately upon becoming faulty. Other process failure models in the literature are send-omission, general (send-receive) omission \[14\], and Byzantine failures \[67\], etc. A component failed in a Byzantine failure manner can exhibit arbitrary and malicious behavior, possibly conspiring with other faulty components. Some algorithm \[1\] considered link failure instead of process failure. Some algorithms designed to solve problems when process crash failures are possible can be automatically trans-
lated [77] to tolerate Byzantine failures supposing that synchronous protocols are in use. Coan [34] extended the translation work on some asynchronous deterministic protocols.

Unfortunately, not all problems can be solved in the face of failures, especially in a totally asynchronous system, where both the message delay and relative process speeds are both uncertain a priori. As a surprising result in [43] showed, the consensus problem is not solvable in the presence of just single faulty process under the most benign form of process crash failure.

Researches on understanding fault-tolerant computing have diversely launched in different directions. Biran et al. Lspace [17] analyzed the combinatorial characteristics of the input and output graph of a problem and provided sufficient and necessary conditions for a problem to be 1-solvable, i.e., solvable when at most one fault can occur. However, the generalization to multiple faults is still unclear and, thus, left as a challenging open problem.

To make problems solvable, some works relaxed problem constraints such as by increasing the allowable decisions for the agreement problem [31], by allowing a larger size of the naming space for the process renaming problem [7], or by assigning the number of processes required for the task assignment problem [22]. Chaudhuri [31] also sought whether the bound for the number of choices provided is related to the number of tolerable faults.
As mentioned, only process crash failure is considered throughout this dissertation. If the system can supply accurate fault diagnosis situation of process status to the applications, problems in distributed systems may easily be made fault-tolerant. Some works [18,36,50,70,91] provide fault-tolerant hardware architecture fault-tolerance; however, due to the limitation of finite amount of hardware, usually a predefined bound on the number of tolerable faults is imposed. Many works [16,106] have been focused on distributed system-level diagnosis. The goal is for every nonfaulty process to correctly reach an accurate independent diagnosis of the fault conditions. The underlying assumption these works are based upon is that each node (or process) of a distributed system can test other nodes and determines them to be faulty or nonfaulty. The outcome of a test performed by a nonfaulty node is accurate and equals the fault state of the node being test. (Tests performed by faulty nodes are inaccurate and results of such tests may be arbitrary.) How to implement such an assumption is not provided.

Recently, some software failure detectors [24,25,83] have been proposed in the application level. Ricciardi and Birman [83] took an alternative approach to modeling the presumed failure of a process by removing it from the group. A process is a presumed failure if it has been "perceived" to have failed by majority of processes. If no one attempts to interact with one that has actually crashed, it will never be perceived to have failed, and some processes that may not actually be crashed may
be perceived as faulty and removed from the group. Essentially, this approach forces
the system to conform with the view of the failure detector.

The abstract properties of completeness and accuracy for a failure detector are
characterized in [25], as opposed to giving specific implementations such as referring
to the exact duration of time-outs. Completeness requires that the failure detector
eventually suspects every process that actually crashes, while accuracy restricts the
mistakes that a failure detector can make. A distributed oracle assuming that there
is one process $p$ that never crashes and that the failure detector eventually stops
suspecting $p$ is used. Under the oracle, the failure detector may make an unbounded
and possibly infinite number of mistakes, but it still correctly solves the consensus
problem.

The weakest information about failures for solving consensus in asynchronous
distributed systems subject to crash failures but with a majority of correct processes
is determined in [24]. The properties of this weakest detector are as follows.

- There is a time after which every process that crashes is always suspected by
  some correct process.

- There is a time after which some correct process is never suspected by any
  correct process.

A system of $k$-fail-stop interconnected processors [86] is implemented to meet the
real-time response constraints in spite of failures. Failures are detected by reading
and comparing values from the stable storage through a solution to the Byzantine General Problem [67]. A processor marks time until all clocks have the same value or a "time-out" period has elapsed. In the latter case, a failure has occurred and it is signaled.

In short, as noted, all the above mentioned "failure detectors" supposedly designed for the asynchronous systems eventually resort to time-outs as the underlying mechanism to fault detection.

2.3 The Consensus Problem

The consensus problem is to reach agreement among distributed processes. This is a fundamental problem in distributed computing and has been extensively studied in the literature. (See [42] for a brief survey.) There are many interesting faces of the consensus problem. Interested readers may refer to [102].

The problem itself is part of any distributed system that involves with coordinated activity. Thus, its solution constitutes an important building block for many other distributed applications ranging over a variety of areas [102]— from the synchronization of clocks, and the election of leaders, to commit protocols for distributed databases, and protocols for ordered atomic broadcast, etc.

The consensus problem is particularly interesting when the system is subject to failures. It can be easy or difficult to achieve consensus depending on the type of
system, synchronous or asynchronous, and the failure model considered. When the correctness of the protocol only involves decisions made by nonfaulty processes, the consensus problem belongs to the class of general coordination problems; otherwise, if the action performed by a faulty process also has to be constrained by those performed by the correct ones, the problem is in the class of consistent coordination problems [14]. In this dissertation, we are only interested in the decisions made by nonfaulty processes.

Chaudhuri [31] extended the consensus problem to the $k$-set consensus problem, in which each processor decides on a single value such that the set of decided values in any run is of size at most $k$. The larger goal is to characterize the boundary between possibility and impossibility in asynchronous systems given multiple faults. A more complicated version of the coordination problem is called the committee coordination problem, which deals with multiway rendezvous of process synchronization and can be found in [10].

2.4 The Termination Detection Problem

The problem of termination detection is to determine whether a distributed system has terminated. An intensive amount of research on distributed algorithms has been devoted to the problem of termination detection. Some algorithms assume synchronous communications [35,37,44,71,76,101], while others work equally well for asynchronous
systems [27, 38, 54, 60, 72, 75]. Some algorithms in the latter category require FIFO channels [27, 75], and some do not [38, 54, 60, 72]. Most algorithms are intended for static systems only; algorithms for dynamic systems can be found in [35, 60]. The termination detectors in [53, 54] are fully distributed in the sense that no predesignated detector is known to exist in the system. The detector in [74] performs a test and decides in a single round whether an underlying computation has terminated. All these algorithms are fault-sensitive, i.e., they may not work correctly in the presence of faulty processes.

Lai and Yang [63] proposed a termination detector in a broadcasting network. It is shown in [100] that the termination detection problem for distributed computations can be modeled as an instance of the garbage collection problem.

Fault tolerance is one of the most desirable properties for a distributed system, and when a termination detector is applied to a fault-tolerant basic system, it is desirable that the detector itself be fault-tolerant. Unfortunately, the problem of termination detection is harder than the consensus problem [105], and the latter is well known to be unsolvable in the face of faulty nodes under the common assumption of reliable asynchronous communications in the underlying network [43]. Thus, in order to solve the termination detection problem in the presence of faulty processes, the model of the underlying network must be stronger than that assumed in [43].

In a recent paper [103], Venkatesan proposed a fault-tolerant termination detector,
called the V algorithm, in which he assumed that the underlying network has the following additional features: a) should a processor fail, each of its neighbors knows of the failure within a finite amount of time; b) a message sent to a failed processor \( x \) is returned to the sender \( y \) after all messages from \( x \) to \( y \) have been received by \( y \); and c) each processor is able to send two or more messages atomically. (Note that in an environment with these extra features, the consensus problem is solvable.) The V algorithm is based on a previous non-fault-tolerant version, the CV algorithm \[26\], and can be \( k \)-resilient for any prespecified value \( k \), \( 1 \leq k \leq n - 1 \). (A protocol or distributed algorithm is said to be \( k \)-resilient if it works correctly even in the presence of \( k \) faulty processes.) The V algorithm has several drawbacks. First, the parameter \( k \) needs to be prespecified by the user. Second, the algorithm requires \( \Omega(kM) \) control messages to prepare itself for possible process failures, where \( M \) is the total number of basic messages (i.e., messages of the basic system). This cost cannot be waived even if no failure actually occurs during the basic computation. Third, should failures occur, the algorithm requires about \( O(kM/n) \) additional control messages per failure, where \( n \) is the total number of processes in the basic system; this cost may be intolerably high in a message-intensive computation. Fourth, it requires processes to send multiple messages in an atomic way.

Assuming the system provides such services as reliable end-to-end communication and the proposed services, (this model is not much different from the one used in
we develop an \((n - 1)\)-resilient algorithm presented in Chapter V that is able to tolerate any number of crash failures. It runs as efficiently as its non-fault-tolerant predecessor if no process actually fails during the computation, and otherwise incurs only a small amount of cost for each actual failure.
CHAPTER III

Consensus and Termination Detection

In this chapter, we describe the traditional model of a distributed system and the two focus problems — the consensus problem and the termination detection problem. We further confirm the belief that termination detection is not solvable in a totally asynchronous system by showing that termination detection is at least as hard as consensus.

3.1 The Traditional Model

![Diagram of Distributed Computing Context](image)

Figure 1: Context of Distributed Computing.
Roughly speaking, a distributed system is defined as a collection of $n$ processes, $P = \{p_1, p_2, \ldots, p_n\}$, communicating through a communication network. As Figure 1 shows, each process runs on a station (computer), on top of a communication subsystem (which, for instance, has seven layers in the ISO OSI reference model). All of the stations are connected to a communication network.

The set of distributed processes cooperating together is called a distributed application. The environment that a distributed application works on will be referred to as a distributed application environment. For our presentation purposes, the functionality of a distributed system will be characterized by what application environments it can provide. In other words, we separate a traditionally defined distributed system into two parts: the distributed application part that performs a user specified task, and the distributed system part that provides services to the application. An application environment is, hence, specified by the services a distributed system (part) supports. If no confusion arises, a distributed system is often used to refer to the distributed system part.

Basic Service of A Distributed System Part

A distribute system part (referring to Figure 1, we mean the communication network, together with the communication subsystems of the stations) normally provides the following basic service:
SO: All-Pair Reliable End-to-End Communications. Each process $i$ is able to communicate with every other process $j$ attached to the system by sending messages to the logical channel $(i,j)$ and receiving messages from the logical channel $(j,i)$. The network guarantees delivery of every message to its destination without error, loss, or duplication — unless the destination process crashes before receiving the message, in which case the message is said to be undeliverable and is discarded by the network without notifying the sender. Messages may not necessarily be delivered in the order they were sent.

A distributed system can be further categorized as synchronous or asynchronous or even partially synchronous [41] according to restrictions imposed on the relative process speed, the communication delay for message delivering, and other system parameters.

A distributed system is said to be totally asynchronous if the following two specifications are met:

- The relative speed among processes is finite but not known a priori.

- The communication delay is finite but arbitrary.

Throughout this dissertation, we will focus our discussion on totally asynchronous systems only. There are several reasons for this. As the number of network nodes increases, it becomes more difficult to predict the communication delay and to guar-
antee totally synchronous services. More importantly, an application written without presuming any system characteristics can be directly applied to or is at least easier to port to other systems. And last, but not the least, the services provided by a totally asynchronous system can be used by a synchronous system for applications running on it as well.

Process Behavior of A Distributed Application Part

A distributed algorithm (or protocol) $\alpha$ running on a distributed system $S$ is a set of $n$ cooperating processes. A process is subject to the most benign crash failure, upon which all its activities halt. A process with crash failure is called faulty, and nonfaulty otherwise. We assume a faulty process does not restart. This model of process failure is the same as that assumed in [43,103]. A totally asynchronous distributed system that is subject to at most $k$ crash failures will be denoted as $S^k$.

A process can be either active, idle, or faulty. A nonfaulty process can be either active or idle. An active process can send or receive messages. Upon finishing its computation, it becomes idle. An idle process does nothing. Upon receiving a message, it becomes active. A protocol is $k$-resilient if it works correctly if there are $k$ or fewer faulty processes.

The state of a process is assumed to be local information and can be obtained by other processes only through message exchanges. (As knowledge cannot be transferred
in a totally asynchronous system without message exchange [29], message exchanges are required.) No nonvolatile storage is assumed to be available, so when a process fails, all its state information is lost (and thus assumed to be undefined).

More than one protocol may run concurrently on a system. A protocol $P$ is said to be monitored by another protocol $P'$ if for every process $p$ of protocol $P$, there is a monitoring process $p'$ of protocol $P'$ such that $p$ and $p'$ reside at the same node and all activities of process $p$, such as receiving a message, sending a message, and updating a local variable, are known to process $p'$. However, $p'$ can neither predict nor alter the behavior of $p$. The notion of monitoring here is similar to the concept of superimposition [21].

In this dissertation, we focus on the problems incurred in a totally asynchronous system where processes may experience crash failure.

3.2 Problem Descriptions

After presenting the traditional model of distributed systems, we now describe two of the most fundamental problems in distributed computing: the problem of reaching consensus among remote processes and the problem of detecting system termination.

3.2.1 The Consensus Problem

The consensus problem is to reach agreement among distributed processes. This is a fundamental problem in distributed computing and has been extensively studied in
the literature. (See [42] for a brief survey.) There are many interesting faces of the consensus problem. Interested readers please refer to [102].

In this dissertation, we consider the strong version of the consensus problem. A consensus protocol is strong if it satisfies the validity condition [102] that the agreed-upon value must have been some process's input. (Note that a strong consensus protocol is also nontrivial as every allowable input value is a possible decision value for (possibly different) assignments of initial values [39].)

The consensus problem is particularly interesting when the system is subject to failures. When the correctness of the protocol only involves decisions made by nonfaulty processes, the consensus problem belongs to the class of the general coordination problems; otherwise, if the action performed by a faulty process also has to be constrained by those performed by the correct ones, the problem is in the class of consistent coordination problems [14]. In this dissertation, we are only interested in the decisions made by nonfaulty processes.

Generally, each process has an initial value, 0 or 1. A consensus protocol is considered correct if eventually each nonfaulty process irrevocably decides on a common value that is one of the n initial values. The definition for the consensus problem is formally given in Definition 1.

**Definition 1** A consensus protocol has to satisfy the following three properties [24]:

1) **Termination:** every nonfaulty process eventually decides;
2) Validity: a process always decides on an input value of some process; and
3) Agreement: no two nonfaulty processes decide on two different values.

3.2.2 The Termination Detection Problem

The termination detection problem [38] is to design a protocol that monitors another protocol, often called the basic protocol, and determines whether the basic protocol is terminated or, in other words, has no possible future changes regarding both computational and communication processing. In a distributed system where processes communicate by message passing and no global clock is available, a process has no up-to-date knowledge about an instantaneous global state. Consequently, to decide whether or not a global state is one in which a distributed computation has terminated becomes nontrivial.

A large amount of research on distributed algorithms has been devoted to the problem of termination detection. (Please refer to Chapter II.)

Traditionally, the basic protocol is said to have terminated if every process of it is idle and no messages are in transit. In the presence of failures, the adapted definition is given in Definition 2. It is clear that the new definition is consistent with the traditional one if no failures occur.

Definition 2 A basic protocol $\alpha$ is said to have terminated if

1) every nonfaulty process of $\alpha$ is idle, and
2) no messages of \( \alpha \) addressed to a nonfaulty process are still in transit.

The *termination detection problem* is to detect the situation that the basic protocol has terminated.

### 3.3 Impossibility of CON and TD

In a totally asynchronous system, many problems become very difficult to tackle when the system is subject to failures. The main reason for this is the asynchrony itself. In a *synchronous* system where processes may fail, the system guarantees that a sender obtains the most up-to-date status of the receiver within a bounded amount of time. This includes the process status, whether a process is faulty, and the channel status, whether there are in-transit messages from the receiver to it. The process failure status can be obtained by time-out and, after failure has been known, the channel from a faulty process can be known to be empty after a bounded amount of delivery time.

Unfortunately, all these nice properties are not attainable in a totally asynchronous system where processes may fail. Therefore, when failures are possible in an asynchronous system, a message sent cannot be assumed to be eventually received, and a message expected cannot be guaranteed to be sent any more (even assuming the underlying protocol is correctly written). This causes many problems to become difficult to solve, or even unsolvable.
In a now well known result, Fischer et al.

Lspace [43] showed the impossibility of solving a (non-trivial) consensus problem in a totally asynchronous distributed system subjected to at most one process crash failure. Because of this result and the view that termination detection seems to be no more than reaching a special kind of consensus — the consensus of whether the system has terminated — the problem of termination detection is believed to be unsolvable in the presence of faulty processes [103]. In the following, we confirm this conjecture by showing that the termination detection problem is \textit{at least as hard} as the consensus problem in a totally asynchronous system. Therefore, we conclude that termination detection is also unsolvable in a totally asynchronous system that is subject to failures.

Throughout the dissertation, CON stands for the strong consensus problem described in Definition 1, and TD for the termination detection problem depicted in Definition 2. The \textit{CON}^k problem requires that every nonfaulty process eventually decide on the same value in a system with at most \( k \) faulty processes. The \textit{TD}^k problem is that of solving the termination detection problem under the assumption that at most \( k \) process failures may occur.

In the following, we show that TD is at least as hard as CON in any totally asynchronous system \( S^k \), \( 0 \leq k \leq n \). (Recall that a distributed system provides service S0.) Since the consensus problem could not be solved in a totally asynchronous sys-
tem with at most one process failure, it would be impossible to solve the termination detection problem in the same system setting. We first explain what it means for one problem to be at least as hard as another.

**Reducibility**

Problem $A$ is *reducible* to problem $B$ in system $S$, denoted as $A \preceq_S B$, if for any protocol solving $B$ in system $S$, it can be employed, with some modification, to solve $A$ in the same system $S$. Thus, if $A \preceq_S B$, then in system $S$ problem $B$ is *at least as hard as* problem $A$ in the sense that 1) if $B$ is solvable in $S$, then so is $A$, and 2) if $A$ is unsolvable in $S$, then neither is $B$, but 3) if $A$ is solvable in $S$, then $B$ may or may not be solvable in $S$. Intuitively speaking, if $A \preceq_S B$, then in system $S$ problem $B$ is never easier than problem $A$ under any condition.

Let $S$ be a class of distributed systems. Problem $B$ is said to be *harder* than problem $A$ in class $S$ if 1) $A \preceq_S B$ for every system $S$ in $S$, and 2) there exists a system $S$ in $S$ such that $A$ is solvable in $S$ but $B$ is not.

The following lemma shows that CON can be reduced to TD in any distributed system $S^k$, $0 \leq k \leq n$, and so TD is at least as hard as CON in $S^k$.

**Lemma 1** \(\text{CON} \preceq_S \text{TD}\) for any distributed system $S$ and any $k$, $0 \leq k \leq n$.

**Proof.** Let $\beta$ be any protocol that solves TD in system $S^k$. We construct a protocol $\alpha$ that solves CON in system $S^k$ making use of $\beta$. The basic idea is to construct a
Figure 2: State Transition Graph of Basic Process $\gamma_i$. 

Figure 3: Process $\alpha_i$ of Consensus Protocol $\alpha$. 

$\gamma_i$ is active \rightarrow decide 1

$\beta_i$ is terminated \rightarrow decide 0
basic protocol $\gamma$ to be monitored by $\beta$ and then the consensus protocol $\alpha$ can make decisions based upon the results of the monitoring.

**Protocol $\gamma$:** Each process $\gamma_i$ in $\gamma$ has a binary input that determines its initial state: if the input to $\gamma_i$ is 1 (0, respectively) then $\gamma_i$ is initialized to be active (idle). During the computation of $\gamma$, whenever an idle process receives a message, it becomes active immediately; and once a process is active (either by initialization or by receiving a message), it remains active and keeps sending messages to every other process. The state transition graph of process $\gamma_i$ is given in Figure 2.

**Protocol $\alpha$:** Each process $\alpha_i$ is given an initial value $x_i$, and all the nonfaulty processes in $\alpha$ want to reach a consensus. To achieve that, protocol $\alpha$ invokes the basic protocol $\gamma$, with $\gamma_i$ initialized according to the value of $x_i$, and invokes $\beta$ to determine whether $\gamma$ is terminated. Specifically, process $\alpha_i$ of $\alpha$ runs process $\beta_i$ of $\beta$ to monitor process $\gamma_i$ of $\gamma$ until either $\beta_i$ declares termination or $\gamma_i$ is active. In the former case, $\alpha_i$ decides on 0, and in the latter case, 1. Process $\alpha_i$ is sketched in Figure 3.

To show $\alpha$ solves CON in system $S^k$, we show $\alpha$ satisfies the three properties, Termination, Validity, and Agreement, described in Definition 1.

For the Termination property, we need to show that, for each nonfaulty process $\alpha_i$, either $\beta_i$ will eventually declare termination or $\gamma_i$ will eventually become active. Since $\beta$ solves TD in system $S^k$, each nonfaulty $\beta_i$ will eventually declare termination,
should $\gamma$ ever terminate. Thus, it suffices to show that either protocol $\gamma$ will eventually terminate or each nonfaulty $\gamma_i$ will eventually become active. Assume that protocol $\gamma$ never terminates and process $\gamma_x$ stays nonfaulty. We show that $\gamma_x$ will eventually become active. By definition of termination, at any time during the computation there is always an active process $\gamma_i$ or a message in transit toward a nonfaulty process $\gamma_j$. Either $\gamma_i$ or $\gamma_j$ eventually will send a message to activate $\gamma_x$ unless it crashes before doing so. Since $\gamma$ never terminates and since there are only a finite number of processes in $\gamma$, it is not hard to see that eventually there will be some $\gamma_i$ (or $\gamma_j$) that has a chance to activate $\gamma_x$ with a message.

The Validity property evidently holds: If a nonfaulty process $\alpha_i$ decides on 0 then it is not hard to derive that $\gamma_i$ has not been activated and the input $x_i$ is 0. If a process decides on 1 then it must either be active at the initialization or have received a message from another process. In the later case, as the number of processes is finite, there is always a process being initially activated. Therefore, a process decides on 1 only when there is some process with input value 1.

As to the Agreement property, suppose, by contradiction, that $\alpha_i$ and $\alpha_j$ are both nonfaulty at some time $t_d$, and by that time $\alpha_i$ has decided on 1 while $\alpha_j$ has decided on 0. According to the protocol, $\alpha_i$ decided on 1 because $\gamma_i$ was active; while $\alpha_j$ decided on 0 because $\gamma$ had terminated. This implies that at time $t_d$, $\gamma_i$ is active while $\gamma$ has terminated — an impossible scenario.
So, if \( \beta \) solves TD in a system \( S^k \), then protocol \( \alpha \) solves CON in \( S^k \). This proves the lemma.

**Theorem 1** TD is unsolvable in a totally asynchronous system \( S^k \), for any \( k \), \( 0 \leq k \leq n \).

**Proof.** Directly from Lemma 1 and the result that CON is not solvable in a totally asynchronous system \( S^1 \) [43].

In this chapter, we have only shown that termination detection is at least as hard as consensus. In the next chapter, we will show the termination detection problem is indeed harder than consensus.

### 3.4 Conclusion

It has been conjectured in the literature that, in a totally asynchronous system, as consensus is impossible in the presence of faulty processes, so is termination detection. In this chapter, we have confirmed this conjecture by showing that termination detection is at least as hard as consensus.
CHAPTER IV

Fault-Detecting and Fault-Flush ing Services

This chapter is primarily concerned with what services a distributed system should provide for a fault-tolerant distributed application and where in the system the services should be rendered so that distributed applications can be resilient to failures even when some system parameters, such as the relative process speed and communication delay, are not known a priori. Two services, fault-detecting and fault-flushing, are proposed. It is explained that they could not be rendered as services implemented in application environment. Instead, the underlying system has to provide such services.

In Chapter III, the problem of termination detection was proven to be at least as hard as the consensus problem. We further observe the effect of the proposed services to the solvability of these two problems. The fault-detecting service is observed to be sufficient for consensus to have an n-resilient solution in a totally asynchronous application environment. However, it is not sufficient for termination detection to be solvable even under a system that is subject to at most one failure. Hence, together with the result from Chapter III, we are able to claim that the termination detection
problem is harder than the consensus problem.

It is worthy of noting that though all the discussions are centered around totally asynchronous environments, it is straightforward to see that the design of synchronous fault-tolerant distributed applications will also be benefit from the proposed services.

In Section 4.1, we first describe the motivation behind the proposal of services for a fault-tolerant distributed application environment. Then, the two services, fault-detecting and fault-flushing, are proposed in Section 4.2. Where and how these two services should be implemented are discussed in Section 4.3 and Section 4.6. Section 4.4 and 4.5 show that the problem of termination detection is harder than the problem of consensus.

4.1 Motivation

As described in Section 3.3 of Chapter III, due to the asynchronous nature of the system, many problems, including the problems of consensus and termination detection (the two of the most fundamental problems in distributed computing) become very difficult to solve or even impossible to tackle when the system is subject to process failures. The main reason for this arises from the lack of up-to-date knowledge about a process in the system. The uncertainty is not only regarding process status—whether a process is faulty or is just being slow, but is also with respect to channel status—whether messages have been sent out but not yet received.
So, what services are appropriate for the system to be able to cope with process failures? First of all, it is reasonable that the system has to provide the basic fault-detecting capability, so a faulty process will not be recognized as a slow one. The importance of the fault-detecting service for fault-tolerance has been recognized in the literature [25,83,86]. In fact, some existing protocols have already assumed the existence of a failure detector [6,8].

It seems that by providing the ability of fault-detecting, every distributed problem automatically obtains a fault-tolerant version of its original algorithm, and becomes apparently solvable. However, as the knowledge about the channel status is not a logical consequence of knowing the process status, messages in transit from a faulty process may bring a process to an unpredictable state if no precaution is taken. We demonstrate not only the process status is an important constituent in solving fault-tolerant distributed problems but the channel status as well. The fault-flushing service, which manages to obtain all messages sent by a faulty process, is proposed to considerably facilitate the job of designing fault-tolerant distributed systems while retaining the asynchronous system nature.

4.2 The Proposed Services

When process failures are possible, two uncertain factors that may affect the design of fault-tolerant protocols arise: the loss of the (faulty) process status and the loss of
channel status.

Unable to distinguish a faulty process from being recognized as a slow one is one of the main reasons that cause problems to be difficult to solve in a totally asynchronous system. Because a receiver or a sender may fail before receiving or sending out an expected message and the lack of knowledge of a process's faulty status, a protocol can no longer be designed in a way such that a sender acts upon the assumption that the receiver will eventually receive the message or, a receiver blocks for an expected message upon the assumption that the sender will eventually send out the message.

Even when failure is detectable, messages in transit from a faulty process may bring a receiver that has detected the failure to an unpredictable state if no precaution is taken.

Therefore, two services, fault-detecting and fault-flushing\(^1\), are described in the following to cope with problems incurred by process failures and hopefully to facilitate the design of fault-tolerant distributed algorithms. To request a service \(SN\), a process \(p_i\) sends an \(SN\).request primitive. Upon receiving such a primitive, the service provider of a service named \(SN\) will monitor every process \(j\) in the system. Whenever the status of process \(j\) fulfills the requirement of service \(SN\), the service provider will eventually send an \(SN\).confirm\((j)\) primitive to process \(p_i\). Any request (or confirm)

---

\(^1\)The concept of flush used in fault processing has appeared in [68], but no actual message flushing was performed. The message \(f(i)\) is created by failure detector (time-out) and the flushing is achieved through the assumption that “the last message that was broadcast by a malfunctioning processor is received before the failure is announced.”
primitive will reach the service provider (or arrive at the request issuer) within a finite but arbitrary amount of time.

These two services are defined more specifically as follows:

**S1: Fault-Detecting.** A process $i$ may issue a `FAULT-DETECT.request(Q)` primitive, where $Q$ is a set of processes (their identities, actually), to request that $i$ be informed in case that any process in $Q$ ever fails. Every time a process $j \in Q$ fails, the service provider will issue a `FAULT-DETECT.confirm(j)` primitive within a finite but arbitrary amount of time to inform $i$ of the failure of $j$.

**S2: Fault-Flushing.** A process $i$ may issue a `FAULT-FLUSH.request(j)` primitive to request for this service, where $j$ is a process known to have failed. This primitive requests that the service provider clear the logical channel from $j$ to $i$. The service provider issues a `FAULT-FLUSH.confirm(j)` primitive to process $i$ after all messages sent from $j$ to $i$ have been delivered (these messages, if any, were sent before $j$'s failure). Thus, after receiving the `FAULT-FLUSH.confirm(j)` primitive, process $i$ knows for sure that no more messages will come from process $j$.

When a system $S$ is capable of providing service $SN$, the system is denoted as $S(SN)$. 
Service S1 clearly distinguishes a faulty process from being recognized as a slow one, so the process status can be learned without difficulty; remedial procedures such as aborting or recovering actions, can be taken to avoid a process blocking forever.

As mentioned, the importance of service S1 for fault tolerance has been recognized in the literature; however, the fault-flushing mechanism provided by service S2, on the other hand, has received little attention from researchers. The motivation for the fault-flushing service arises from the observation that in-transit messages from a faulty channel may carry not only consistent, but also productive results that are essential to the computation. Service S2 still retains the asynchronous flexibility of a system provided to the application processes and at the same time reveals the most up-to-date knowledge of a faulty process from pushing a nonfaulty process to receive all messages sent by the failed process. Notice that in service S2 only messages sent by a faulty process are flushed by the network. For messages sent to a faulty process, the network can either choose to discard the message or return it to the sender, depending upon the underlying protocols involved.

4.3 Service Provider of S1 and S2

The assumption that the underlying network (or the distributed system part) provides fault-detecting and fault-flushing services might be undesirable. It would be nice if the proposed services could be directly implemented as a totally asynchronous
distributed application to assist the design of fault-tolerant algorithms without requesting any new facility from the underlying network. Unfortunately, due to the inherent uncertainties incurred by process failures of a totally distributed system, these two services are shown to be not renderable solely in a totally asynchronous distributed system $S$.

Thus, it would seem necessary to install them as part of services supplied by the network. Since network protocols may utilize some synchronous nature presented by lower layers, such as timeout, or some other extra hardware to support fault tolerance, the uncertainties involved in an asynchronous system can be cleared up and, thus the provision of the proposed services made feasible.

In Sections 4.3.1 and 4.3.2, we explain why these two services are impossible to be directly implemented in a totally asynchronous distributed system.

### 4.3.1 Service $S_1$ is impossible in $S$

The result that it is impossible to implement service $S_1$ in a totally asynchronous application can be easily obtained from the following two well-known previous works. First, the problem of consensus is unsolvable even in system $S^1$ [43]. (Recall that $S^1$ represents a system that is subject to at most one failure.) Second, there is an $n$-resilient consensus protocol in system $S^n(S_1)$ [25]. It is not hard to see that if $S_1$ could be implemented in $S$, then consensus would be solvable in $S^n$ (from second result), which contradicts the classical work mentioned in the first result. ([83] also pointed
out that the fault-detecting service is impossible in an asynchronous environment in which the underlying network provides only the S0 service.) So, we have to rely on the underlying network to provide service S1.

4.3.2 Service S2 is impossible in S1

In order to solve the fault-tolerant termination detection problem, it is important to know whether the logical channel from a faulty process to a nonfaulty process is free of basic messages. Like the fault-detecting service, this job cannot be done by the basic processes themselves, and once again we need to rely on the underlying network to do it.

The following shows that service S2 is impossible to implement in $S^1$ even if the network provides service S1; in other words, we show that service S2 is impossible to implement in system $S^1(S1)$. The proof is done by contradiction.

Suppose there is a protocol $\delta$ that solves S2 running in system $S^1(S1)$.

Let $C$ be a computation of an interleaved sequence of all the computations of protocols running in the system. Let $\alpha_i$ be a process of protocol $\alpha$ running in the system, and suppose that $\alpha_i$ fails at time $t_f$ in $C$. Since $\delta$ provides S2, a process $\alpha_j$ that requested the service will receive S2.confirm(i) rendered by protocol $\delta$ indicating that $\alpha_i$ has failed and there are no in-transit messages in channel $(i,j)$ at some time $t_r > t_f$.

Let $C'$ be another computation which is identical to $C$ except that in $C'$, process
\( \alpha_i \) sends out a message \( m \) to \( \alpha_j \) immediately before \( t_f \), and the message \( m \) arrives at process \( \alpha_j \) far after \( t_r \), say \( t_m \). Because not any process can predict the behavior of \( \alpha_i \), all other processes behave the same as they do in \( C \); in particular, no messages of protocol \( \delta \) that are not sent in \( C \) will be sent out in \( C' \). Therefore, the only difference of \( C \) and \( C' \) during \([0, t_r]\) is the sending event occurring at \( \alpha_i \) right before \( t_f \). It is not hard to verify the existence of \( C' \) as the system is totally asynchronous.

As knowledge cannot be transferred in a totally asynchronous system without message exchange [29], the existence of \( m \), thus, was only known to node \( i \), which unfortunately has died. All other nodes will not know anything about it until after time \( t_r \), when \( m \) arrives at its destination. Due to the fact that the local state of a faulty process is undefined after \( t_f \), its states in computations \( C \) and \( C' \) are the same at any time between \( t_f \) and \( t_m \). Consequently, protocol \( \delta \) cannot distinguish \( C' \) from \( C \) until then. As a result, \( \delta \) will, by mistake, send \( \alpha_j \) in \( C' \) the notice at time \( t_r \). This contradicts the assumption that \( \delta \) correctly implements \( S_2 \) in \( S^1(S1) \).

We have explained why it is impossible to implement these two services directly in a totally asynchronous application environment, so we have to depend on the underlying network to realize them. The implementation issues will be further discussed in Section 4.6. From now on, when we say a system \( S \) has service capability \( S_1 \) or \( S_2 \), or both, we assume that the service provider is the underlying network or the distributed system part (not the application part).
4.4 Fault-Detecting Service to Consensus

Before we can prove that the termination detection problem is harder than the consensus problem, we first show that fault-detecting service is sufficient for the consensus problem to be solvable in a totally asynchronous application environment.

As described in Section 3.2.1, the consensus problem \([43]\), denoted as \(CON\), is to achieve an agreement among nonfaulty processes of a given protocol. Each process has an initial value, 0 or 1. Eventually each nonfaulty process has to irrevocably decide on a common value that is one of the \(n\) initial values. An algorithm that solves the consensus problem is called a consensus protocol.

The problem is particularly interesting in the presence of faulty processes. In a surprising result \([43]\), Fischer et al. showed that in a totally asynchronous system, the consensus problem cannot be solved even assuming no more than a single failure. This is true even if the type of failure considered is the most benign one, the process crash failure. In other words, in a totally asynchronous system, no 1-resilient consensus protocol is possible.

On the other hand, Dolev et al. \([39]\) identified two “minimal” cases in which the consensus problem can be \(n\)-solved without using broadcast transmission. They are

1. synchronous processes and synchronous communication;
2. synchronous processes and synchronous message order.
These two conditions are each said to be "minimal" because any weakening of either condition by changing one parameter from synchronous to asynchronous is sufficient for a proof that there is no $k$-resilient protocol where $k$ is either 1 or 2.

Each of the above two conditions implies that process failure is detectable. However, that faults can be detected within a finite amount of time (or fault detectability) does not necessarily imply any of these conditions. That is, fault detectability is weaker than them. It is thus natural to make this weaker assumption and ask whether fault detectability is sufficient for the existence of an $n$-resilient consensus protocol.

In the following, we answer this question in the affirmative.

4.4.1 An $N$-Resilient Consensus Protocol

As mentioned above, the condition of fault detectability is weaker than the two minimal cases described in [39]. In this section, we show that fault detectability is indeed sufficient for the existence of an $n$-resilient consensus protocol in a totally asynchronous system.

To solve the consensus problem, one can simply let a nonfaulty process with the smallest identity, say $p_1$, decide on its input value, and have all other nonfaulty processes decide on that value as well. After $p_1$ decides, it broadcasts its decision to every nonfaulty process known to it. Upon receiving a decision $d$, a process decides on $d$. Notice that 'broadcast' here means 'sending the same message to every desired destination' and the action itself may not be atomic. As the system is asynchronous,
not every nonfaulty process will receive such a decision simultaneously. It is possible that after receiving the decision message sent by $p_1$ before its failure, $p_3$ decides on the decision made by $p_1$, say 1. Meanwhile, $p_2$ is notified by the network of $p_1$'s failure before the arrival of the decision message from $p_1$, if any, and makes its own decision on its own input 0. An erroneous state has been reached due to an inconsistent view of the set of faulty processes. Therefore, to correctly resolve this situation, a (logical) synchronization step to obtain consistent knowledge about the set of faulty processes is needed before a process regards itself as the one with smallest identity and prepares to decide.

The synchronization mechanism works as follows. Every process that is notified of the failure of another process sends a report to the process with the smallest identity known to it. The report carries the set of faulty processes it knows of and the decision if it has been made. A process that recognizes itself as the process with smallest identity and prepares to decide on a common value for the system should wait to receive a report from every nonfaulty process. Upon receiving all the necessary reports, it can then correctly decide on its own input if no process has decided. It is not hard to verify that all decisions made and carried in a report to the same process are the same, and, thus eventually each nonfaulty process will irrevocably decide on a common value that is one of the $n$ initial values.
The Protocol

We design an $n$-resilient consensus protocol under system $S$ with fault-detecting service $S1$ provided. The protocol, referred to as Protocol $\gamma$, is presented in Figure 4 as four event-driven actions. Each action is triggered when its “guarded” condition is satisfied.

Each process has a unique ID number, which, without loss of generality, is assumed to be between 1 and $n$. At any time, the nonfaulty process with smallest ID is defined to be the leader. Each process $i$ maintains the following four variables.

1. $d_i$ is the output decision register indicating the decision made by $p_i$. Initially, $d_i = \bot$ (undefined).

2. $leader_i$ is a boolean indicating whether $p_i$ is the process with smallest ID among all nonfaulty processes, so as to qualify for making a final decision. Initially, only $leader_1$ is true.

3. $F_i$ records the set of faults known to $p_i$. Initially, $F_i = \emptyset$ (empty set).

4. $FaultSet_i$ is an array whose element is a set of integers. $FaultSet_i[j]$ indicates the set of faulty processes detected by $p_j$ that are known to $p_i$. Initially, $FaultSet_i[j] = \emptyset$ for all $j$.

The protocol employs two kinds of messages: Decide($v$) and Check($F, v$). A Decide($v$) message is used for the leader to inform other processes about the value $v$
it has decided on. A Check\((F, v)\) message is used for a process to inform the leader of the faults \(F\) it has detected and the value \(v\) it has reached.

In A1, when \(p_i\) receives a Decide\((v)\) message from \(p_j\), if i) according to \(p_i\)'s knowledge \(p_j\) is not faulty and ii) \(p_i\) has not yet decided on a value for \(d_i\), then \(p_i\) makes the same decision as \(p_j\). In A2, upon detecting a process's failure, if process \(p_i\) is the one with smallest ID among nonfaulty processes, then \(p_i\) becomes the leader. Otherwise, \(p_i\) reports to the process with ID equal to \(\min(P - F_i)\), which is likely to be the current leader. It is not too hard to see that at any time there is at most one nonfaulty process with \(leader\) set to true. Only a leader may receive a Check\((F, v)\) message. Upon receiving such a message in A3, the leader records the detected faults \(F\) and the publicized value \(v\). A4 indicates how and when the leader decides on and publicizes a value. After a process \(i\) has recognized itself as the leader (i.e., processes \(1, \ldots, i - 1\) have been detected by \(p_i\) to be faulty), if the failures of processes \(1, \ldots, i - 1\) are also known by every other nonfaulty process, then \(p_i\) decides on a value and publicizes it. The correctness of Protocol \(\gamma\) is established in the following.

In a distributed system, an intuitive claim usually leads to erroneous conclusions because of the numerous possible interleaved executions that a distributed algorithm can present. In order to convince that fault detectability is sufficient for the consensus problem, we formally prove in the next theorem that protocol \(\gamma\) n-solves \(CON^n\) assuming each fault is eventually detectable.
A1: (Upon receiving Decide(v) from $p_j$)
   if ($d_i = \bot$) and ($j \notin F_i$) then $d_i := v$;

A2: (Upon receiving FAULT-DETECT.confirm(x))
   
   \[ F_i := F_i \cup \{x\}; \]
   
   \[ \begin{align*}
   & \text{if } i = \min(P - F_i) \text{ then } \text{leader}_i := \text{true}; \\
   & \text{else send Check}(F_i, d_i) \text{ to } \min(P - F_i);
   \end{align*} \]

A3: (Upon receiving Check($F, v$) from $p_j$)
   
   \[ \begin{align*}
   & \text{FaultSet}_i[j] := \text{FaultSet}_i[j] \cup F; \\
   & \text{if } (d_i = \bot) \text{ and } (v \neq \bot) \text{ then } d_i := v;
   \end{align*} \]

A4: (When (leader$_i = \text{true}$) and
   
   \[ \{1, \ldots, i - 1\} \subseteq \text{FaultSet}_i[j] \text{ for all } j \in P - F_i - \{i\} \])
   
   \[ \begin{align*}
   & \text{if } d_i = \bot \text{ then } d_i := x_i; \\
   & \text{for } j \notin F_i \text{ do send Decide}(d_i) \text{ to } p_j;
   \end{align*} \]

\[ \text{(* } x_i \text{ is the input register of } p_i. \text{ *)} \]

Figure 4: The Protocol for Process $i$, $1 \leq i \leq n$
Theorem 2 Protocol \( \gamma \) solves the consensus problem even in the presence of any number of process failures.

Proof. It is sufficient to establish the progress and safety properties for the protocol: 1) every nonfaulty process eventually decides (i.e., assigns a value to \( d_i \)), and 2) the decision values reached by nonfaulty processes are all equal.

To prove the first property, consider any execution of the protocol in which exactly \( f \) processes fail, where \( 0 \leq f \leq n \). After the \( f \) failures have all occurred, let \( p_i \) be the surviving process with smallest ID number. By the assumption of fault detectability, every nonfaulty process eventually detects all the failures and recognizes \( p_i \) as the leader. At the time, if every nonfaulty process has decided then we are done. Otherwise, suppose there is a nonfaulty process that has not yet decided. On detecting the last failure, each process reports to \( p_i \) via a Check\((F, v)\) message, with \( F \) containing the ID numbers of all failures (A2). After all these reports are received, \( p_i \) executes A4 and publicizes its decision with Decide messages. By A1, every nonfaulty process will at latest decide upon receiving such a message. This establishes the progress property.

To prove the safety property, again consider any execution of the protocol. At any time \( t \) during the execution, a Decide message from \( p_i \) to \( p_j \) that has not been received or is being received is said to be valid if \( p_j \) is nonfaulty and \( i \notin F_j \). Also, call a not-yet-received Check\((F, v)\) message valid if \( v \neq \bot \) and the destination is a
nonfaulty process that has not decided. We establish the safety property by proving the following proposition: At any time $t$, all decided nonfaulty processes, as well as all valid Decide and Check messages, carry the same decision value. Only the following five events may possibly affect the set $X$ of decided nonfaulty processes or the set $Y$ of Decide and Check messages and thereby change the validity of the proposition: E1) when a (nonfaulty) process decides, E2) when a (valid) Decide message is sent, E3) when a (valid) Check message is sent, E4) when a (valid) Decide message is nullified; i.e., when a fault is detected or the destination process crashes, E5) when a (valid) Check message is nullified; i.e, when the destination process crashes or decides.

Initially, at time 0, the proposition is obviously true, as no process has decided and no Decide or Check messages are in transit. Assume that immediately before an event $E$ the proposition holds. We show that the proposition holds after event $E$.

Let $d$ be the common value referred to in the proposition. Let $d$ be $\perp$ if $X = Y = \emptyset$.

E4 and E5 only reduce $X$ or $Y$ and obviously do not affect the proposition.

When E2 or E3 occurs, the decision value carried in the Decide or Check message must be $d$ and the proposition holds. E1 may occur only in A1, A3, or A4. The first two cases obviously do not falsify the proposition. In the case of A4, we claim that, before event $E$, no nonfaulty process has decided and no valid Check or Decide message is in transit (i.e., $X = y = \emptyset$ or $d = \perp$). To see this, suppose that a process $j > i$ with $j \notin F_i$ has decided. (Recall that $p_i$ is the leader and all processes $p_\ast$
with \( x < i \) are faulty.) The decision value of \( p_j \) must come from a then-valid Decide message sent by some \( p_x, \ 1 \leq x < i \). Let \( p_j \) decided at time \( t_1 \). We know \( x \notin F_j \) at time \( t_1 \). Let \( t_2 \) be the time at which \( p_j \) sent a Check message with \( x \in F_j \) to \( p_i \), which did happened because \( p_i \) had received a Check\((F, v) \) message from \( p_j \) that included \( \{1, \ldots, i - 1\} \) in the \( F \) set. Clearly, \( t_2 > t_1 \). That means that the decision value \( v \) carried in the Check message was not \( \bot \). So, \( p_i \) would have made its decision upon receiving that Check message in \( A_3 \), a contradiction. So there is no such decided nonfaulty process \( p_j \) at the time \( p_i \) decides on its \( x_i \). This in turn implies that no valid Check messages are in transit. From the guarded condition of \( A_4 \), there are no valid Decide messages in transit because all previous valid Decide messages have been nullified. Therefore, \( d \) equals \( \bot \) before \( A_4 \), and after \( A_4 \) \( d \) becomes \( x_i \), and the proposition still holds. \( \Box \)

### 4.5 TD Is Harder Than CON

In Section 3.3 of Chapter III, we have shown TD to be at least as hard as CON: whenever CON is not solvable, neither is TD. If the problem of termination detection is viewed as nothing but reaching a special kind of consensus, one might suspect that TD is possibly solvable in system \( S^n(S1) \). Unfortunately, this is not the case. As a matter of fact, we show that TD is not a special case of consensus. There are environments in which CON is solvable but TD is not. We identify \( S(S1) \) as one such
environments. Since the existence of an $n$-resilient protocol for CON under system $S^n(S1)$ has been shown in Lemma 2 of Section 4.4, we show there does not exist a protocol solving TD in system $S$ with service $S1$ provided. Together with the result from Section 3.3 in Chapter III, we conclude TD is in fact harder than CON.

**Lemma 2** TD is *not solvable* in $S^1(S1)$.

**Proof.** Assume there is a protocol $\beta$ that solves the problem of TD for any given basic protocol $\gamma$ running in system $S^1(S1)$. Let a combined computation be an interleaved sequence of the computation of $\gamma$ and the computation of $\beta$. Let a combined computation $c$ possess the following properties: 1) one of $\gamma$'s process, say $\gamma_i$, failed at time $t_f$; 2) every other process was notified by the perfect failure detector provided by the underlying network of $\gamma_i$'s failure by time $t_n > t_f$, and 3) $\gamma$ terminates at time $t_\ell > t_n$. As protocol $\beta$ correctly solves the problem of TD in system $S^1(S1)$, at some time $t_d > t_\ell$, $\beta$ will declare $\gamma$'s termination.

Let $c'$ be another combined computation which is identical to $c$ except that in $c'$, process $\gamma_i$ sends out a basic message $m$ immediately before $t_f$, and the message $m$ arrives at its destination far after $t_d$, say at $t_m$. In $c'$, because the process of $\beta$, say $\beta_i$, that monitors $\gamma_i$ cannot predict the behavior of $\gamma_i$, it will not send out any control messages that are not sent in $c$. Therefore, the only difference of $c$ and $c'$ during $[0, t_d]$ is the sending event occurring at $\gamma_i$ right before $t_f$. (It is not hard to verify the existence of $c'$ as the system is totally asynchronous.)
As knowledge cannot be transferred in a totally asynchronous system without message exchange, the existence of \( m \), thus, was only known to node \( i \) which unfortunately has died. All other nodes will not know anything about it until after time \( t_d \) when \( m \) arrives at its destination. Due to the fact that the local state of a faulty process is undefined after \( t_f \), its states in computations \( c \) and \( c' \) are the same at any time between \( t_f \) and \( t_m \). Consequently, protocol \( \beta \) cannot distinguish \( c' \) from \( c \) until then. As a result, \( \beta \) will, by mistake, declare termination in \( c' \) at time \( t_d \). This contradicts the assumption that \( \beta \) correctly solves TD in \( S^1(S1) \), and thus there does not exist such a protocol that solves TD in \( S^1(S1) \).

Hence, we conclude in the following theorem that TD is actually harder than CON in any \( S^k \), \( 0 \leq k \leq n \).

**Theorem 3** TD is harder than CON in \( S^k \), for any \( k \), \( 0 \leq k \leq n \).

There is also another system in which CON is solvable, but TD is not. Consider a system \( S \) with the existence of an immortal process. The agreement of decision can be easily monopolized or judged by the immortal process, but there is no easy way to make known to the immortal process of the messages in transit from a faulty process to a nonfaulty one (including the immortal process). It is not too hard to verify that termination detection is not solvable in such an environment.
4.6 Discussion

As shown in Section 4.3, directly implementing the fault-detecting and fault-flushing services in a totally asynchronous distributed application environment is not feasible. However, these two services have been shown critical for fault-tolerant distributed applications. Inevitably, we have to rely on the underlying communication network as the platform for providing these two services. Some remarks on the feasibility of services S1 and S2 to be provided by the underlying network are pointed out below.

Normally, timeout mechanism, a synchronous service that is available at the lower layers, can be deployed to accomplish service S1. Several works in fault-tolerant computing used time-outs primarily (or exclusively) for the purpose of failure detection. As commented in [8], this approach "can be viewed as an asynchronous algorithm that uses a fault detection (e.g., timeout) mechanism." Therefore, we can assume that a network node is capable of testing whether or not a station (host) directly connected to it is faulty — this is a common assumption in the literature of fault diagnosis [9,16,80] — and so the communication network is able to provide the fault-detecting service. The reader is referred to [25] for an interesting survey of a variety of failure detectors. Though a timeout mechanism may be used by the underlying network to provide service S1, it should not be viewed as the only mechanism sufficient to achieve service S1. Thus, it should not be misunderstood that the system has been changed to the synchronous mode. From the application point of view, the system still retains
the totally asynchronous characteristic, so the protocol design can still be based on that.

 Guaranteeing that a channel from a faulty node is free of messages is not a trivial task as messages transmitted via a logical channel may actually travel along more than one path. The fault-flushing service as depicted in S2 is a concept inspired by the forward-flush of [4,5,40,58] and can be implemented using similar techniques. A message \( m \) is sent as forward-flush if every message \( m' \) sent before it to the same destination has to arrive first. A forward-flush message is a non-fault-processing flush primitive defined and implemented in [3,4]. Its purpose is to provide non-FIFO concurrency to most messages and only imposes ordering on some specified messages. (Note that a FIFO transmission implicitly provides the forward-flush service.)

 If the underlying network is reliable, and only computing nodes, where application processes are running, may fail, the service S2 can be simply rendered by generating forward-flush \texttt{FAULT-FLUSH} messages. Whenever a network node detects the failure of its host computing node \( x \) and delivers all messages requested to be sent by \( x \) (though not necessarily arrived to the destinations yet), it generates and sends forward-flush \texttt{FAULT-FLUSH}(\( x \)) messages to all other (nonfaulty) computing nodes. Upon receiving the \texttt{FAULT-FLUSH}(\( x \)) message at node \( y \), the network has guaranteed that all messages sent by \( x \) to \( y \) before the \texttt{FAULT-FLUSH} message have arrived at the destination \( y \) and thus no more in-transit messages will come from \( x \) to \( y \).
If the network nodes are prone to failure, the issue would become a very complex matter. As mentioned in [86], it is impossible to implement a perfect system in spite of failures by using only a finite amount of hardware, only a network node with high reliability can possibly be achieved by hardware replication with nonvolatile storage available. The degree of replication should be dictated by the failure rate to reflect the expected reliability. For an architecture that maps the application processes and the network process to different boards [32], the network process board can be supplied with a more reliable power source to reduce the likelihood of crashing. More advanced approaches to the implementation of fault-flushing service that incur less overhead are left open to possible novel hardware technology and software methodology.

4.7 Conclusion

Two services, fault-detecting and fault-flushing are augmented to a totally asynchronous system to facilitate the design of fault-tolerant distributed applications. The fault-detecting service has been recognized in the literature; however, the fault-flushing service has received little attention. We showed that the ability to detect faulty processes plays an important role in both consensus and termination detection problems. This feature is sufficient for the consensus problem to be solvable. While fault detectability is necessary for termination detection to be 1-solvable, it alone is not sufficient. Augmented with the fault-flushing service, the termination detection
problem is shown to be solvable in the next chapter.

We also showed that no existing solution which provides these two services based directly upon a totally asynchronous application environment. Instead, the underlying network is a more suitable platform for providing these services. Some implementation issues related to this are also presented.

Though the problem of termination detection has been viewed as nothing but that of reaching a consensus state (namely, terminated), we show, on the contrary, that termination detection is harder than consensus. Recently, a stronger version of the consensus problem, called agreement consensus, has been shown to be harder than the strong consensus problem discussed [31]. It would be interesting to investigate this problem's relationship with termination detection, as well as its relationship with services S1 and S2.
CHAPTER V

An \((N - 1)\)-Resilient Distributed Termination Detection Algorithm

This chapter presents a fault-tolerant termination detection algorithm that is based on a previous fault-sensitive one by Dijkstra and Scholten. The proposed algorithm is able to tolerate any number of crash failures. It runs as efficiently as its non-fault-tolerant predecessor if no process actually fails during the computation, and otherwise incurs only a small amount of cost for each actual failure. It is assumed that the underlying communication network provides such services as reliable end-to-end communication (S0) described in Chapter III, and fault-detecting and fault-flushing services described in Chapter IV.

In this chapter, for the presentation and performance comparison reasons, the model for a distributed system is depicted following in a traditional manner, and the problem description for the termination detection is restated for clarity.
5.1 Introduction

A distributed system is a set of processes running on top of a communication network (directly or indirectly above the transport layer of the ISO OSI reference model [97]. In the absence of faulty processes, the system is said to have terminated if two conditions are satisfied: i) all processes are idle and ii) there are no messages in transit.

The problem of termination detection is to determine whether a distributed system has terminated. A protocol or algorithm that solves the problem is called a termination detector. The system whose termination is the object of the detection is called the basic system.

Many algorithms for termination detection are available in the literature. Some algorithms assume synchronous communications [35,37,44,71,76,101], while others work equally well for asynchronous systems [27,38,54,60,72,75]. Some algorithms in the latter category require FIFO channels [27,75], and some do not [38,54,60,72]. Most algorithms are intended for static systems only; algorithms for dynamic systems can be found in [35,60]. All these algorithms are fault-sensitive and may not work correctly in the presence of faulty processes.

Fault tolerance is one of the most desirable properties of distributed systems, and when a termination detector is applied to a fault-tolerant basic system, it is desirable that the detector itself be fault-tolerant. Unfortunately, the problem of termination detection is harder than the consensus problem [105], and the latter is well known to
be unsolvable in the face of faulty nodes under the common assumption of reliable asynchronous communications for the underlying network [43]. Thus, in order to solve the termination detection problem in the presence of faulty processes, the model of the underlying network must be stronger than that assumed in [43].

In a recent paper [103], Venkatesan proposed a fault-tolerant termination detector, called the V algorithm, in which he assumed that the underlying network has the following additional features: a) should a processor fail, each of its neighbors knows of the failure within a finite amount of time; b) a message sent to a failed processor \( x \) is returned to the sender \( y \) after all messages from \( x \) to \( y \) have been received by \( y \); and c) each processor is able to send two or more messages atomically. (Note that in an environment with these extra features, the consensus problem is solvable.) The V algorithm is based on a previous non-fault-tolerant version, the CV algorithm [26], and can be \( k \)-resilient for any prespecified value \( k, 1 \leq k \leq n - 1 \). (A protocol or distributed algorithm is said to be \( k \)-resilient if it works correctly even in the presence of \( k \) faulty processes.) The V algorithm has several drawbacks. First, the parameter \( k \) needs to be prespecified by the user. Second, the algorithm requires \( \Omega(kM) \) control messages to prepare itself for possible process failures, where \( M \) is the total number of basic messages (i.e., messages of the basic system). This cost cannot be waived even if no failure actually occurs during the basic computation. Third, should failures occur, the algorithm requires about \( O(kM/n) \) additional control messages per failure, where
n is the total number of processes in the basic system; this cost may be intolerably high in a message-intensive computation. Fourth, it requires processes to send multiple messages in an atomic way.

We assume an underlying communication network that provides such services as reliable end-to-end communication (S0) described in Chapter III, and fault-detecting (S1) and fault-flushing (S2) described in Chapter IV. (This model is not much different from the one used in [103], and will be summarized in the next section.) With the proposed model, we develop an \((n - 1)\)-resilient algorithm based on a variant (the LTD variant [61]) of the well-known Dijkstra-Scholten (DS) algorithm [38], where \(n\) is the total number of processes in the basic system. The LTD variant, like the original DS algorithm, is worst-case message optimal and is one of the most efficient termination detectors available in the literature. Our algorithm has a very interesting property: it achieves fault tolerance without message replication and thereby avoids all preparation cost (in terms of control messages); if no process ever fails during basic computation, our algorithm is as efficient as its predecessor, the LTD algorithm. (For comparison, recall that the V algorithm needs \(\Omega(kM)\) control messages to get ready for possible faults.) This property is certainly desirable, since in today's systems it seems that most computations are done without a process failure. Our algorithm also improves over the V algorithm in several other aspects: 1) should a process fail during the computation, our algorithm recovers with at most \(O(n)\) control messages, which
is in most cases smaller than the V algorithm's $O(M)$ (assuming $k = n - 1$, as in our case); 2) the space complexity is $O(n^2)$ as compared with $O(nM)$ for the V-algorithm (as $M$ is not known at compile time, the V algorithm requires the more expensive dynamic memory allocation); 3) our algorithm has a shorter detection delay of $O(n)$ as compared with the $O(M)$ required by the V algorithm; and 4) our algorithm does not need the service of sending multiple messages atomically, which, in contrast, is crucial for the V algorithm.

We assume that all processes are able to communicate with each other and that the underlying network provides such services as S0, S1 and S2. These services are described in the next section, where the problem of termination detection is also defined again. We will review the DS algorithm and the LTD variant in Section 5.3, and describe our algorithm in Section 5.4. The algorithm will be proved correct and analyzed in Sections 5.5 and 5.6, respectively.

5.2 The Model

In this section, we model the underlying communication network and formally define the problem of fault-tolerant termination detection.

5.2.1 The Underlying Communication Network

A distributed system is a set of processes communicating through a communication network. As Figure 1 in Chapter III shows, each process runs on a station (com-
puter), on top of a communication subsystem (which, for instance, has seven layers in the ISO OSI reference model). All the stations are connected to a communication network. We assume that the communication network, together with the communication subsystems of the stations where the basic processes reside, provides the following services:

**S0: All-Pair Reliable End-to-End Communications.** Every process is able to communicate with every other process. The network guarantees delivery of every message to its destination without errors, loss, or duplication — unless the destination process crashes before receiving the message, in which case the message is said to be undeliverable and is discarded by the network without notifying the sender. Messages may experience finite but arbitrary delay in the network, and they are not necessarily delivered in the order they were sent.

**S1: Fault-Detecting.** The communication network provides the service of fault-detecting. A process $i$ may issue a `FAULT-DETECT.request(Q)` primitive, to request that it be informed should any process in $Q$ fails, where $Q$ is a set of process identities. Every time a process $j \in Q$ fails, the network sends to $i$ a `FAULT-DETECT.confirm(j)` primitive to report that $j$ has failed. We do not assume that failures be reported to processes in any particular order. Thus, it is possible that the network reports failures to different processes in different orders.
S2: Fault-Flushing. The network also provides a fault-flushing service. A process $i$ may issue a `FAULT-FLUSH.request($j$)` primitive to request this service, where $j$ is a process known to have failed. This primitive requests that the network clear the logical channel from $j$ to $i$. The network issues a `FAULT-FLUSH.confirm($j$)` primitive to process $i$ after all messages sent from $j$ to $i$ have been delivered. (These messages, if any, were sent before $j$'s failure.) Thus, after receiving the `FAULT-FLUSH.confirm($j$)` primitive, process $i$ knows for sure that no more messages will come from process $j$.

We make the following definition just for convenience in describing our algorithm in subsequent sections.

**Definition 3** We say that “process $i$ has detected process $j$’s failure” iff process $i$ has received a `FAULT-DETECT.confirm($j$)` primitive.

It must be emphasized that the processes themselves are not capable of detecting failures. It is the underlying communication system that provides the fault-detecting service. The reader is referred to [25] for an interesting survey of a variety of failure detectors, and to [9,16,80] for further discussions on fault diagnosis.

The fault-flushing service as depicted in S2 was inspired by the *forward-flush* of [4] and by the *return flush* of [103]. In a return flush, if the destination of a message has failed, the network returns the message to its source node after the network has cleared (flushed) the channel from the destination to the source node. Both return-
flush and fault-flushing serve the same purpose: to ensure that no basic messages from a faulty process are pending.

5.2.2 The Problem

The basic system is a set of n processes, $P = \{p_1, p_2, \ldots, p_n\}$, which communicate with one another by sending and receiving messages through a communication network. Each process has a unique identity and is aware of other processes' identities. Without loss of generality, let their identities be $1, 2, \ldots, n$. We shall interchangeably refer to the process labeled with $i$ either as $p_i$ or $i$.

Associated with each process is a process execution context. A process execution context consists of the state of a process, and the computation of a process. The computation of a process in the basic system is called basic computation. Before its computation gets started, a process's state is either undefined or faulty, meaning the process has not started its computation or has crashed, respectively. Once a process starts its computation, it is in exactly one of three possible states: idle, active, or faulty. A process is said to be idle (or active or faulty) if it is in the idle (or active or faulty) state. Each basic computation is a sequence of such events as receiving a message, sending a message, performing arithmetic/logical (A/L) operations, changing the process's state, and becoming faulty (crashing) and is subject to the following rules:

- While in the active state, a process can perform A/L operations, send messages,
and receive incoming messages without any restriction. It may become idle in an unpredictable fashion.

- Once in the idle state, a process does not perform any A/L operations and cannot send any messages; but it can receive an incoming message, upon which it becomes active immediately.

- An idle or active process may fail (crash). Once a process fails, it stops running and never recovers. (This model of process failure is the same as that assumed in [43,103].)

- A faulty process is one that has failed. So a faulty process can neither send nor receive messages.

The state transition graph is given in Figure 5.

The messages sent by the processes during their basic computations are called basic messages, as to be distinguished from the control messages employed by a termination detector. A basic message is said to be in transit if it has been sent but has not yet been received. A process which is not faulty is nonfaulty.

The basic computations of the n processes, taken together as a whole, are called the computation of the basic system. In the literature, the computation of a basic system without faulty processes is said to have terminated iff all processes are idle and there is no basic message that is still in transit; and the problem of termination detection is to design a protocol capable of determining whether the computation of
In this paper we study the problem of termination detection in the presence of faulty processes. The exact formulation of the problem partly depends on how the underlying communication network handles undeliverable messages, namely, those addressed to faulty processes. We assume that the communication network discards undeliverable messages without notifying their senders, and accordingly define our problem as follows.

**Definition 4** The computation of a basic system is said to have terminated iff 1) every nonfaulty process is idle and 2) no basic message whose destination is a nonfaulty
process is still in transit.

Note that we ignore undeliverable messages in the above definition since they are assumed to be eventually discarded by the network. This definition of termination is consistent with the traditional one if no process ever fails during the basic computation.

Definition 5 The problem of fault-tolerant termination detection is to design a protocol to determine whether the basic computation has terminated. It is required that 1) after the basic computation terminates, some process in the system will eventually declare termination (unless all processes crash before any process has a chance to do so) and 2) if some process declares termination at some point of time, then the basic computation has really terminated by that moment.

An algorithm that solves the termination detection problem is called a termination detector. We adopt the common view that a termination detector will be "superimposed" on the basic system. That is, the detector will be implemented as an integrated part of the basic system, so that the same process at each station will run both its basic computation and the termination detection algorithm. There is an interesting discussion on the notion of superimposition in [21].

Remarks: In our model, we assume that the network discards all undeliverable messages. If undeliverable messages are instead returned back to their senders (as
assumed in the V-model), the basic computation of the system should accordingly be
defined to have terminated iff 1) every nonfaulty process is idle and 2) the source and
destination processes of all basic message in transit are presently faulty. In that case,
a return-flush service is needed; and with some minor modifications, our algorithm
offered in this paper will still work.

5.3 Previous Work

As mentioned in the introduction, our algorithm is based on the LTD variant of the
Dijkstra-Scholten algorithm [38], and will be evaluated against the resilient protocol
of Venkatesan [103]. In this section, we review the DS algorithm and summarize
the V algorithm. A thorough understanding of the DS algorithm will later help in
understanding ours.

5.3.1 Dijkstra-Scholten Algorithm and Its Variants

The DS algorithm is applicable to diffusing computing, in which a distinguished pro-
cess, called the root, starts the basic computation, with all others initially idle. (Ex-
cept for this constraint, diffusing computing is no different from a general distributed
computation as described in the preceding section. In particular, each process is either
idle or active during the computation.) Each process is regarded either as neutral or
as engaged, depending on its status. The concept of “neutral/engaged” is a product
of the algorithm; it is related, but not equivalent, to that of “idle/active.” Initially,
only the root is engaged; all others are neutral. If a neutral process \( p \) receives a basic message, say, from \( q \), it becomes engaged as well as active; \( p \) is said to be engaged by \( q \) and an edge \( p \rightarrow q \) is introduced between the two processes; the involved message is called an engagement message. In this way, all engaged processes form a tree with edges directed from children to parents. Every basic message is acknowledged by its receiver with a SIGNAL. A process can acknowledge an incoming non-engagement message any time after the receipt. However, it can acknowledge an engagement message only if the following three conditions are satisfied: i) the process is idle, ii) it has acknowledged all incoming non-engagement messages, and iii) all its outgoing messages have been acknowledged. An engaged process becomes neutral again once it has acknowledged its incoming engagement message. The diffusing computation is declared to have terminated when the root becomes neutral (or more precisely, when the root satisfies the above three conditions).

The algorithm is spelled out in Figure 6 with \( p_1 \) acting as the root. Each process \( p_i \) maintains three variables: \( in_i[1..n] \), \( out_i \), and \( parent_i \), where \( in_i[j] \) is the number of yet-to-be-acknowledged messages received from \( p_j \), \( out_i \) is the total number of \( p_i \)'s outgoing messages which have not been acknowledged, and \( parent_i \) is the process that engaged \( p_i \). (Initially, \( out_i = in_i[j] = 0 \) and \( parent_i = \text{NULL} \) for all \( i \) and \( j \).) Note that a process \( p_i \) is neutral iff \( p_i \) is idle, \( \sum_j in_i[j] = 0 \), and \( out_i = 0 \). Also note that the variables \( parent_i \) define a tree consisting of all engaged processes. The tree
D1: (When $p_i$ sends a basic message)
\[ out_i := out_i + 1; \]

D2: (When $p_i$ receives a basic message from $p_j$)
\[
\text{if } (p_i \neq p_1) \text{ and } (parent_i = \text{NULL}) \\
\text{then } parent_i := j; \\
\]
\[
{\text{(* $p_i$ becomes engaged. *)}}
\]

D3: (When $\sum_k in_i[k] > 1$)
\[
\text{select a process } p_j \text{ such that } (in_i[j] > 0) \\
\text{and } (j \neq parent_i \text{ if } in_i[parent_i] = 1); \\
\text{send a SIGNAL to } p_j; \\
\text{in}_i[j] := \text{in}_i[j] - 1; \\
\]

D4: (When $p_i$ is idle, $out_i = 0$ and ($p_i = p_1$ or $\sum_k in_i[k] = 1$))
\[
\text{if } p_i = p_1 \text{ then } \\
\text{declare termination; } \\
\text{else } \\
\text{send a SIGNAL to } parent_i; \\
\text{in}_i[parent_i] := \text{in}_i[parent_i] - 1; \\
parent_i = \text{NULL}; \\
\text{(* $p_i$ becomes neutral. *)} \\
\text{endif}
\]

D5: (When $p_i$ receives a SIGNAL from $p_j$)
\[ out_i := out_i - 1; \]

Figure 6: Dijkstra-Scholten Algorithm for $p_i, 1 \leq i \leq n$. 
changes shape as time passes: a neutral process may become engaged and join the
tree as a leaf and a leaf may become neutral and drop off the tree. When the root
becomes neutral, the computation is declared to be terminated.

The DS algorithm requires exactly $M$ control messages, where $M$ is the number of
basic messages. This is optimal in terms of worst-case complexity. The DS algorithm
has two drawbacks: it always requires $M$ control messages and it works only for
diffusing computations. These, however, can be easily overcome as shown in the LTD
variant [61]:

1. Instead of acknowledging one message per signal as in the original DS algorithm,
let each SIGNAL carry an integer $c$ so that it acknowledges $c$ basic messages at a
time. (This in general will reduce the number of control messages, although in the
worst case it still needs $M$.)

2. Instead of initializing for all $i$, $parent_i := \text{NULL}$, $in_i := 0$ and $out_i := 0$ as in
the original DS algorithm, the LTD variant initializes these variables as follows:

- Let $parent_1 := \text{NULL}$ and $parent_i := 1$ for all other processes $i$ (so that the $n$
  processes form a tree of height 1 with $p_1$ at the root).

- Set $in_i$ and $out_i$ in accordance with the above tree: 1) for all processes $i, j$, let
  $in_i[j] = 1$ if $j = parent_i$, and 0 otherwise; and 2) for all $i$, let $out_i = \sum_j in_j[i]$.

These simple changes in initialization enable the DS algorithm to work for non-
diffusing computations. The reader is referred to [61] for a proof.
The worst-case message complexity of the LTD variant is \( M + n - 1 \), which is optimal. We shall adopt the LTD extension in our effort to make the DS algorithm fault-tolerant.

The LTD algorithm is asymmetric. In [90], a non-diffusing computation is treated as a collection of diffusing computations, resulting in a beautiful symmetric termination detector that works for both diffusing and non-diffusing computations. Its worst-case message complexity is \( O(M + n \times E) \), where \( E \) is the number of communication links.

5.3.2 A Fault-Tolerant Termination Detector

Venkatesan's algorithm [103] is a fault-tolerant version of a previous algorithm by Chandrasekaran and Venkatesan [26].

The main idea of the CV algorithm is similar to that of the DS algorithm. But, instead of counting messages, the CV algorithm uses stacks, one per process, to keep record of incoming and outgoing messages. Each process's transactions of sending or receiving a message are sequentially logged into its stack. When a process becomes idle, it removes all "receiving" entries on the top of the stack until a "sending" entry is reached. The process acknowledges the messages corresponding to these removed receiving entries. A sending entry is removed from the stack when an acknowledgment to the corresponding message is received. Like the original DS algorithm, the CV algorithm, once invoked, needs a control message for each basic message. The most
interesting feature of the CV algorithm is that it need not start running right from the beginning of the basic computation. It may start later than the basic computation and still work correctly. This useful feature of "postponed start" has been incorporated into the DS algorithm as shown in the LTD variant [61].

The CV algorithm was transformed into the fault-tolerant V algorithm by message replication. Each node is associated with \( k \) other nodes, called its representatives, where \( k \) is the number of tolerable faults. (The user of the algorithm needs to specify the value of \( k \).) Whenever a node enters an entry into its stack, it atomically replicates the entry and sends one copy to each representative. Each representative keeps a stack for each node represented by it. In this way, should a node fail, at least one of its representatives is still alive and will be able to simulate the faulty node's local stack.

5.4 \((N - 1)\)-Resilient Algorithm

As was mentioned earlier, our algorithm is based on a variant of the DS algorithm. We first discuss some of the problems that need to be solved in order to make the DS algorithm fault-tolerant; then we describe our algorithm.

5.4.1 Problems Caused by Faulty Processes

Recall that in the DS algorithm all engaged processes form a tree \( T \). When the tree degenerates to nil, the computation is known to have terminated. There are at least three problems that need to be solved in order to make the algorithm fault-tolerant.
First, if the coordinator (the root) fails, some surviving process must succeed as the coordinator. Second, a nonfaulty process must be able to tell whether an inbound channel from a faulty process is empty. Third, if an engaged process, say $p$, fails, then the subtree $T_p$ rooted at $p$ will break off from $T$ and the nonfaulty nodes in $T_p$ will be unable to report to the coordinator when they become neutral. To be resilient to faulty processes, it is thus necessary to re-attach to $T$ the nonfaulty nodes in $T_p$ or, in other words, to establish new child/parent relationships between the offspring of $p$ and the remaining nodes in $T$. This requires making both a child and its new parent aware of the new relationship.

The second problem can be easily solved using the FAULT-FLUSHING service provided by the network (see Section 5.2.1). To solve the first problem, let process $p_1$ be the coordinator initially. If it fails, let process $p_2$ take over the job of coordination. If process $p_2$ also fails, then process $p_3$ succeeds as the coordinator. In general, we let the nonfaulty process with the smallest index be the coordinator. In our algorithm a process $p_i$ ($i \neq 1$) which becomes a coordinator will know its new status only after it has received from the network a FAULT-DETECT.confirm($j$) for each $j < i$.

The third problem is considerably harder, especially if more than one process fails. There are three issues here:

a) Which nodes in $T_p$ should be attached back to $T$ — every node in $T_p$ or just the children of $p$?
b) *Which nodes in* $T$ *should these nodes be attached to?*

c) *How is a node* $q$ *in* $T_p$ *actually attached to a node* $r$ *in* $T$? *It is not sufficient for* $q$ *to simply send a notice to inform* $r$ *of the new child/parent relationship, because due to message delay, it is possible that* $r$ *becomes neutral and leaves tree* $T$ *before receiving the notice.*

Issues a) and b) are not trivial because each process has only a little knowledge of the structures $T$ and $T_p$. The only thing an engaged process knows about tree $T$ is the identity of its own parent and that of the root; the latter may or may not be up-to-date information. A process in general does not know whether it is a descendant of the detected fault $p$ (unless $p$ is its parent), and which processes are in $T$ and which are not. With little information about $T$ and $T_p$, we will simply let every engaged process that detects a fault become a child of the coordinator. That is, when an engaged process detects a fault, it adopts as its new parent the process that it believes is the coordinator. When the engaged process becomes neutral, it will send a report (signal) to the coordinator.

Now consider issue c). With the above strategy, the coordinator now knows that all engaged processes will eventually become its children after the failure of a process. The problem is that the coordinator does not know which processes are engaged and which are not. We need a scheme to ensure that the coordinator will know whether it has received a report from every child. Note that all the problems mentioned above
are compounded by the facts that there may be multiple process failures, that not all processes detect a failure immediately or simultaneously, that the coordinator may fail, and that processes may have out-of-date knowledge about the coordinator.

Despite these tangled problems, the basic ideas of our solution are simple. First, when a nonfaulty process, engaged or neutral, detects a fault in the system, let it adopt the coordinator as its parent and let it send a signal (report) to the latter when it is done. Second, let the signal not only carry a number as in the DS algorithm to acknowledge basic messages, but also a set of processes that contains all faults the process has so far detected. The set of detected faults is intended to help the coordinator know whether every child has signaled its completion of processing. Roughly speaking, if \( S_i = F \) for every process \( i \in F \), then the coordinator has received the most updated report (signal) from each child and thus has sufficient information for checking if the system is terminated, where \( S_i \) is the set of faults reported to the coordinator by process \( i \) and \( F \) is the set of all faults detected by the coordinator.

Having pointed out the major problems caused by faulty processes as well as our basic strategies, we are in a position to describe the algorithm.

5.4.2 Algorithm Description

Let \( P = \{p_1, p_2, \ldots, p_n\} \) be the set of processes of the basic system. Process \( p_i \) is identified by the integer \( i \) and is often referred to as process \( i \) or just \( i \). A faulty process is simply referred to as a fault.
Overview. Our fault-tolerant algorithm is based on the LTD variant of the DS algorithm. It is basically the LTD variant plus the following extensions:

- Each process $i$ records the faults that have been reported to it by the underlying network. Let $DF_i$ denote the set of all such faults.

- When a process $i$ sends a signal as in the DS algorithm, the set $DF_i$ is included in the signal. In this way, processes share information about faults.

- Each process $i$ keeps track of the faults that have been reported to it by every other process. Let $RF_i[j]$ denote the set of faults reported by process $j$. (One can assume that $RF_i[i] = DF_i$ for the time being.)

- When a process $i$ is informed of a fault by the underlying network, it issues a FAULT-FLUSH.request to clear the channel from the faulty process to $i$ itself, and changes its parent to the coordinator.

- The coordinator, say process $r$, makes use of $DF_r$ and $RF_r[1..n]$ to avoid premature declaration of termination. Before the coordinator can check whether the basic computation has terminated, it first checks whether $DF_r = RF_r[j]$ for every $j \notin DF_r$. The latter condition roughly means that the coordinator has received a most recent signal from each nonfaulty process. Only if this condition holds can the coordinator go ahead to check for system termination in a way similar to the DS algorithm.

Local Variables and Control Messages. A control message SIGNAL($c, S$) is used in our algorithm to acknowledge the receipt of $c$ basic messages and to report that
the sender has detected all the processes in \( S \) as faulty ones. A single \( \text{SIGNAL}(c, S) \) message of our algorithm has the effect of \( c \) signals of the DS algorithm. The \( S \) parameter is used to help the coordinator determine whether it has received a report from every nonfaulty process.

Each process \( p_i \) maintains the following local variables:

- \( \text{in}_i[1..n] \) — an integer array, where \( \text{in}_i[j] \) indicates the number of messages that \( p_i \) has received from \( p_j \) but has not yet acknowledged.

- \( \text{out}_i[1..n] \) — an integer array, where \( \text{out}_i[j] \) records the number of messages sent to \( p_j \) for which an acknowledgment has not yet been received.

- \( \text{parent}_i \) — indicating which process to report to when \( p_i \) becomes neutral.

- \( \text{crd}_i \) — a boolean variable indicating whether \( p_i \) is the coordinator.

- \( DF_i \) — the set of all processes \( x \) for which \( p_i \) has received a \( \text{FAULT-DETECT}.\text{confirm}(x) \). By Definition 3, this set consists of all processes that \( p_i \) has detected as faulty. Thus, one may regard \( DF \) as meaning detected failures.

- \( RF_i \) — the union of all sets \( S \) contained in the \( \text{SIGNAL} \) messages that \( p_i \) has so far received. The processes in this set are those which have been reported to be faulty by other processes. The mnemonics \( RF \) stands for reported failures.

- \( \text{NumFault}_i[1..n] \) — an integer array, where \( \text{NumFault}_i[j] \) indicates \( p_i \)'s knowledge about \( |DF_j| \), the number of faults that \( p_j \) has detected. (Rather than keep track of the set \( RF_i[j] \) for each \( j \) as mentioned above in the overview, it turns out to be
sufficient to just record the number \( \text{NumFault}_i[j] = |RF_i[j]| \) for each \( j \), and keep one single set \( RF_i \).

Note that a process may know of a fault through a SIGNAL or a \( \text{FAULT-DETECT.confirm} \) primitive, and it stores this information in \( RF_i \) or \( DF_i \) accordingly. In this paper, whenever we say that a process "detects" a fault, we exclusively refer to the event as defined in Definition 3, not counting that of knowing a fault through a SIGNAL.

Initialization of Variables. The above mentioned variables are initialized as follows.

- Each process \( i \) initially does not know of any process failure, so \( DF_i = RF_i = 0 \) and \( \text{NumFault}_i = 0 = (0, \ldots, 0) \).
- Let \( p_1 \) be the first coordinator. So initially \( \text{crd}_1 = \text{true} \) and \( \text{crd}_i = \text{false} \) for \( i \neq 1 \).
- Let \( \text{parent}_1 = \text{NULL} \) and \( \text{parent}_i = 1 \) for all other processes \( i \), so that each process will later report to \( p_1 \). Note that the \( n \) processes form a tree with \( p_1 \) at the root.
- Initialize \( in_i \) and \( out_i \) in accordance with the above tree: for all processes \( i, j \), let \( in_i[1] = out_1[i] = 1 \) for all \( i \neq 1 \), and \( in_i[j] = out_j[i] = 0 \) otherwise. (This tree structure accommodates the cases where there are multiple initially active processes.)

The Algorithm. The algorithm for each process \( p_i \), \( 1 \leq i \leq n \), is given in Figure 7
as a number of event-driven actions. After the process gets started, there are six events that may trigger an action — sending or receiving a basic message, becoming idle, receiving a signal, receiving a FAULT-FLUSH.confirm, and detecting a fault (i.e., receiving a FAULT-DETTECT.confirm) — some of them may involve multiple actions. Actions A1-A5 correspond to actions D1-D5 of the DS algorithm, and are responsible for termination detection. The other two, F1 and F2, cope with faulty processes.

A0 initializes the variables \(i_n, o_{ut}, etc.\) as described in the above, and issues a FAULT-DETTECT.request(\(P\)). It takes place when a process starts its basic computation. Action A1 simply counts outgoing messages, and A2 counts incoming ones. Moreover, in A2, if, upon receiving a basic message, \(p_i\) is not the coordinator and has no parent, then it becomes a child of the process from which the message came. Intuitively, this means \(p_i\) becomes engaged.

Action A3 is executed on two occasions: 1) when a process becomes idle (i.e., when the process's state changes to idle from either undefined or active, or 2) when a process detects a fault. In the latter case, F1 and A3 are both triggered and must be executed in that order. The main function of A3 is to acknowledge non-engagement messages. So a SIGNAL is sent to every nonfaulty process \(j (j \neq parent_i)\) from which there are still unacknowledged messages. The signal carries two pieces of information: the number of messages \(p_i\) wants to acknowledge (\(in_i[j]\)) and the set of faults \(p_i\) has
A0: (When \( p_i \) starts its basic computation)
initialize the variables \( out_i \), \( in_i \), etc., and issue a FAULT-DETECT.request(\( P \));

A1: (When \( p_i \) sends a basic message to \( p_j \))
\[
out_i[j] := out_i[j] + 1;
\]
\( (* \text{counts outgoing messages.} *) \)

A2: (When \( p_i \) receives a basic message from \( p_j \))
\[
in_i[j] := in_i[j] + 1;
\]
\( (* \text{counts incoming messages.} *) \)
if \((-\text{crd}_i) \text{ and (parent}_i = \text{NULL}) \text{ then parent}_i := j; \)
\( (* \text{becomes engaged.} *) \)

A3: (When \( p_i \) becomes idle or whenever F1 is executed)
if \( p_i \) is idle then
for each \( j \) such that \( j \neq \text{parent}_i \), \( j \notin DF_i \cup RF_i \) and \( in_i[j] \neq 0 \) do
send a SIGNAL(in[i],[DF_i]) to \( p_j \);
\( in_i[j] := 0; \)
\( (* \text{has acknowledged p}_j \text{'s messages.} *) \)
endfor

A4: (Whenever A3, A5, or F2 is executed)
if \( p_i \) is idle, \( out_i = 0, \sum_{k \neq \text{parent}_i} in[k] = 0 \) and \( RF_i \subseteq DF_i \) then
if \( \text{crd}_i = \text{false} \) then
send a SIGNAL(in[i],[parent_i],[DF_i]) to parent_i;
\( in_i[\text{parent}_i] := 0; \text{ parent}_i := \text{NULL}; \)
elseif \( \forall j \in P - DF_i - \{i\}: \text{NumFault}_i[j] = |DF_i| \) then
declare termination;
\( (* p_i \text{ has detected termination.} *) \)
endif
\( (* p_i \text{ becomes neutral if it was engaged.} *) \)

A5: (When \( p_i \) receives a SIGNAL(c, S) from \( p_j \))
\[
out_i[j] := out_i[j] - c;
\]
\( (* c \text{ more messages are acknowledged by p}_j \text{.} *) \)
\[
 RF_i := RF_i \cup S;
\]
if \( \text{NumFault}_i[j] < |S| \) then \( \text{NumFault}_i[j] := |S|; \)

F1: (When \( p_i \) receives a FAULT-DETECT.confirm(x) primitive)
issue a FAULT-FLUSH.request(x) primitive;
\( DF_i := DF_i \cup \{x\}; \)
\( out_i[x] := out_i[x] + \epsilon; \)
\( (* p_i \text{ is waiting for the FAULT-FLUSH.confirm(x).} *) \)
if \( i = \text{min}(P - DF_i) \) then
\text{crd}_i := true; \text{ parent}_i := \text{NULL};
\( (* \text{becomes the coordinator.} *) \)
else \text{parent}_i := \text{min}(P - DF_i);
\( (* \text{becomes a child of the coordinator.} *) \)

F2: (When \( p_i \) receives a FAULT-FLUSH.confirm(x) primitive)
\( in_i[x] := 0; \text{ out}_i[x] := 0; \)
\( (* \text{no in-transit deliverable messages.} *) \)

Figure 7: Algorithm for \( p_i, 1 \leq i \leq n \).
so far detected \((DF_i)\). Note that no signal is sent to \(parent_i\). So, should \(p_i\) ever change the value of \(parent_i\) after it has become idle, it may need to send a signal to its original parent. Since \(parent_i\) may change value in \(F1\), \(A3\) is executed after \(F1\).

The purpose of action \(A4\) is for \(p_i\) to send a signal to its parent \((parent_i)\) or, if it is the coordinator, to check if the system has terminated. This must be done when \(p_i\) is ready to become neutral, i.e., when these conditions are satisfied: \(p_i\) is idle, \(out_i = 0\), \(\sum_{k \neq parent_i} in_i[k] = 0\), and \(RF_i \subseteq DF_i\). So \(A4\) is executed immediately following \(A3\), \(A5\), or \(F2\), each of which has a chance to bring about at least one of the above conditions. In this action, it is first checked whether \(p_i\) is ready to become neutral. If so and if \(p_i\) is not the coordinator, then it reports to its parent with a SIGNAL and updates \(in_i[parent_i]\) and \(parent_i\) to reflect its neutral status. If \(p_i\) is the coordinator, then it checks whether \(\text{NumFault}_i[j] = |DF_i|\) for all \(j \in P - DF_i - \{i\}\). If so, \(p_i\) declares termination. We shall prove in the next section that if \(p_i\) is the coordinator, then the conditions that \(p_i\) is idle, \(out_i = 0\), \(\sum_{k \neq parent_i} in_i[k] = 0\), \(RF_i \subseteq DF_i\), and \(\text{NumFault}_i[j] = |DF_i|\) for all \(j \in P - DF_i - \{i\}\) together imply the termination of the basic computation.

\(A5\) is triggered when \(p_i\) receives a SIGNAL\((c, S)\) from \(p_j\). The signal carries two pieces of information: 1) an integer \(c\) for \(p_j\) to acknowledge \(c\) messages and 2) a set \(S\) for \(p_j\) to report detected faults. So in this action, the number of unacknowledged messages \(out_i[j]\) is reduced by \(c\) and the set \(S\) is added to \(RF_i\), the set of faults that
have been reported to \( p_i \) by other processes through SIGNALs. Also, \( \text{NumFault}_{i|j} \) is raised to \( |S| \) if its current value is smaller than \( |S| \). This variable records how many faults have been reported by \( p_j \). It is updated only if the new signal reports more faults than the previous ones (from \( p_j \)). It is possible that the \( S \) contained in the current signal be smaller than previous ones (in which case \( \text{NumFault}_{i|j} \) will not be changed) because the signals of \( p_j \) may arrive at \( p_i \) in a non-FIFO order.

When \( p_i \) receives a FAULT-DETECT.confirm(\( x \)) primitive, \( F1 \) is invoked. The newly detected fault is added to \( DF_i \) and \( out_i[x] \) is incremented by \( \epsilon \), any positive number. (For understanding the algorithm, it is convenient to think of \( \epsilon \) as a small number. In implementation, one may let \( \epsilon = 1 \).) In normal cases, \( out_i[x] \) counts only outgoing basic messages. Here, we artificially increase the value of \( out_i[x] \) so that \( p_i \) can ensure that there are no messages in the channel from \( x \) to it. It works as follows: 1) \( p_i \) issues a FAULT-FLUSH.request(\( x \)) primitive; 2) the network will clear the logical channel from \( x \) to \( i \) and inform \( p_i \) of the fact by a FAULTFLUSH.confirm(\( x \)) primitive; 3) when \( p_i \) receives a FAULT-FLUSH.confirm(\( x \)) primitive in \( F2 \), \( in_i[x] \) and \( out_i[x] \) are set to zero to reflect the fact that the channel between \( p_x \) and \( p_i \) is now free of deliverable basic messages; 4) before that, \( out_i[x] \) stays positive, thereby falsifying the condition of the outer if-statement of A4 and preventing \( p_i \) from prematurely becoming neutral or announcing termination. In \( F1 \), since a new fault is detected, there is a chance that \( p_i \) emerges as the new coordinator. So \( p_i \) checks if its index is
the smallest among all processes which it has not detected as faulty. If so, \( p_i \) becomes

the coordinator and accordingly updates \( \text{crd}_i \) and \( \text{parent}_i \). Otherwise, \( p_i \) becomes a

child of process \( r = \min(P - DF_i) \), which is believed to be the coordinator (that of course may or may not be true, depending on whether or not \( r \) is still alive). Note that in \( F1 \), one may substitute \( P - DF_i - R_F \) for \( P - DF_i \) in the if-else-statement.

Before proceeding to prove the algorithm correct, we demonstrate how it works by examples.

**Example 1.** Consider a distributed system consisting of seven processes

\( p_1, p_2, \ldots, p_7 \), in which \( p_7 \) is neutral and the others are engaged as shown in Figure 8 (a). If no process fails during the rest of the computation, the tree defined by the \( \text{parent}_i \) variables will evolve as in the DS algorithm. For instance, if processes \( p_6, \ldots, p_2 \) successfully become neutral, they will leave the tree one after another. On leaving the tree, \( p_2 \), in particular, will send a signal to \( p_1 \). That will enable \( p_1 \) to
become neutral and declare termination as soon as it becomes idle.

Example 2. Consider again the system as shown in Figure 8 (a), but this time $p_4$ fails, resulting in a structure as shown in Figure 8 (b). Because of $p_4$'s failure, $p_5$ is no longer able to propagate its signal along the original path $(p_4, p_3, p_2, p_1)$. Our algorithm lets every surviving process (except for $p_1$ itself) attach to $p_1$ as a child on detecting $p_4$’s failure. Suppose that $p_3$ is the first process to detect the failure. It not only changes its parent to $p_1$ (see Figure 8 (c)), but also sends a signal to its original parent, $p_2$, to acknowledge all $p_2$'s messages including the engagement message (see A3, which is invoked after F1). So $p_2$ no longer holds responsibility for $p_3$’s status.

Suppose now that $p_2$ becomes neutral and so signals $p_1$, while $p_3$ is still active (see Figure 8 (d)). In Example 1, $p_2$’s signal to $p_1$ enabled $p_1$ to declare termination. Here, $p_2$’s signal to $p_1$ will not cause a premature declaration of termination because the information about $p_4$’s failure has been passed to $p_1$ along the path $(p_3, p_2, p_1)$. Aware of $p_4$’s faulty status, $p_1$ will not declare termination until after it has received from each nonfaulty process a signal that reports $p_4$’s faulty status. In particular, when $p_1$ receives the signal (reporting $p_4$’s failure) from $p_3$, $p_1$ can ensure that $p_3$ (which was engaged by $p_2$) has finished. The following are some other subtle points about our algorithm:

- Although $p_7$ is neutral, it also has to send a signal to $p_1$ after detecting $p_4$’s failure, because $p_1$ expects to receive a signal from every nonfaulty process.
• After detecting $p_4$'s failure and before signaling $p_1$ (i.e., while waiting for a FAULT-FLUSH.confirm($p_4$) primitive), $p_7$ may possibly receive a basic message (either from $p_4$ or from any other process). If that happens, $p_7$ becomes active and remains as a child of $p_1$; it does not change parent to the sender of that message (see A2).

5.4.3 Postponed Start

As mentioned in Section 5.3.2, both the CV and the V algorithms start to run only after they switch to detection mode. While the original DS algorithm does not have a "postponed start" feature, the LTD variant implements it in a way such that no control message is sent before the detector switches to detection mode; although the detector is actually running while in non-detection mode, its work involves only simple computation such as counting outgoing and incoming basic messages and is cost-negligible. For clarity and simplicity, we chose in the previous subsection not to include the postponed-start feature in the algorithm. In the following we briefly discuss how to incorporate this feature into the algorithm.

A new local variable $mode_i$ is introduced to indicate whether process $i$ is in detecting-termination (DT) or non-detecting-termination (NDT) mode; and a new type of control message, called START, is used. All the actions in Figure 7 remain the same except for A0, A3 and A4. We need to modify A0 so that $mode_i$ gets initialized to NDT. The other two actions are modified as follows. First, a new con-
dition "mode\textsubscript{i} = DT" is added to the (outer) if-statements of A3 and A4, so that no process in NDT mode will send any SIGNALs. Second, the event "\(p_i\) changes from NDT to DT mode" is added to the two actions' lists of guarding conditions, so that both actions will be invoked soon after a process enters DT mode. Thus, for instance, A3 will become:

**A3:** (When \(p_i\) becomes idle or enters DT mode
or whenever F1 is executed)

\[
\text{if } \text{mode}\textsubscript{i} = \text{DT} \text{ and } p_i \text{ is idle then } \ldots;
\]

Two new actions are introduced to deal with START messages:

**A6:** (The coordinator spontaneously executes this action
exactly once)

\[
\text{if } \text{mode}\textsubscript{i} = \text{NDT then } \text{mode}\textsubscript{i} := \text{DT};
\]

\[
\text{for all } j \in P - DF_i - RF_i - \{i\} \text{ do}
\]

\[
\text{send a START message to } p_j;
\]

**A7:** (When \(p_i\) receives a START message)

\[
\text{if } \text{mode}\textsubscript{i} = \text{NDT then } \text{mode}\textsubscript{i} := \text{DT};
\]

**A6** indicates that the algorithm must be implemented in a way such that the coordinator (i.e., the process with \(crd_i = \text{true}\)) will execute the action once, and only
once, unless it fails before having a chance to do so. In this action, the coordinator sends a START message to every nonfaulty process. Once a process receives a START message, it enters DT mode (A7). Note that the coordinator may enter DT mode by itself or by receiving a START message from a former coordinator. The reason for a coordinator already in DT mode to execute A6 is that the command "send a START message to every nonfaulty process" is not atomic. If the former coordinator from which the present coordinator received its START message had crashed in the course of sending START messages, then some processes may not receive a START from it. So, the new coordinator has to send START's to ensure that every nonfaulty process will eventually enter DT mode.

In the next section we will establish the correctness of the algorithm as presented in Figure 7. It will be not hard to see that, with the above modifications, the algorithm still can be proved along the same line of reasoning, requiring only a few minor changes.

5.5 Correctness Proof

We establish the correctness of the algorithm by proving that: 1) if some process declares termination at some point of time, then the basic computation is really terminated at that moment, 2) after the basic computation terminates, some process in the system will eventually declare termination (unless all processes crash before
any process has a chance to detect the termination).

As was mentioned earlier, we assume that the termination detector will be superimposed on the basic system. The computation of a basic system and that of a superimposed termination detector together are called a combined computation, which is the computation of \( n \) concurrent processes.

As in [64], we adopt the common approach of defining an execution of a concurrent algorithm to be a sequence of atomic actions in which concurrent actions of separate processes are assumed to be interleaved in an arbitrary manner. It is thus necessary to specify which of the algorithm's operations are atomic. There is a Folk Theorem in [64] that reads "when reasoning about a multiprocess program, one can combine into one atomic action any sequence of operations that contains only a single access to a single shared variable." Applying this theorem, we model the combined computation (of the basic system and the superimposed termination detector together) as a single sequence of atomic events, where each event is one of the following.

**AE1** A process fails.

**AE2** A process becomes idle.

**AE3** A process issues or receives a FAULT-DETECT or FAULT-FLUSH primitive.

**AE4** A process performs a sequence of operations that involve only local variables.

**AE5** A process receives or sends a single (basic or control) message and performs a sequence of operations that involve only local variables.
In particular, we regard the body of action $A_1$ together with its "guarding condition" as an atomic event (of type $AE_5$); the same for $A_2$ and $A_5$. Each iteration of the for-loop of $A_3$ (i.e., sending a signal and updating $in_i[j]$) is an $AE_5$ event. In $A_4$, sending a signal and updating $in_i[parent_i]$ and $parent_i$ is also an $AE_5$ event.

Conventions: For ease of presentation, we adopt the following conventions throughout this section:

- We assume the existence of an external clock to which the processes have no access. If $C = (e_1, e_2, \ldots, e_m)$ is a combined computation, let $(t_0, t_1, \ldots, t_m)$ be any sequence of points of time (according to the external clock) such that $e_i$ occurs between $t_{i-1}$ and $t_i$.
- When we say any time, we mean any time between two consecutive events, not within an event.
- When we refer to a local variable of a faulty process, we mean its value as of the time when the process failed.
- For any variable $x$, $x(t)$ indicates the value of $x$ as of time $t$; $t$ may be omitted if no confusion may arise.

With these conventions, we formally state as follows the two properties, soundness and completeness, of our algorithm that will be established later.

Soundness: If $(e_1, e_2, \ldots, e_d)$ is a combined computation such that some process declares termination in event $e_d$, then the basic computation has really terminated
by time $t_{d-1}$.

**Definition 6** A combined computation $C$ is said to be *complete* if either of the following two statements is true.

i) $C$ contains an infinite number of events.

ii) $C = (e_1, \ldots, e_m)$ is finite and satisfies both of the following two conditions:

   ii.a) there is at least one nonfaulty process at time $t_m$;

   ii.b) unless one of these processes crashes (after $t_m$), no event of any other type may occur after $t_m$ (thus, the only type of event that may occur directly following $C$ is the crash of a process; only after such a crash other types of events such as receiving a FAULT-DETECT.confirm, etc., may occur).

Note that an incomplete combined computation $C$ can always be extended to a combined computation $C'$ such that either $C'$ is complete or all processes fail in $C'$. Thus, we formulate the completeness property of our algorithm as follows.

**Completeness:** If $C = (e_1, e_2, \ldots, e_t, \ldots)$ is a complete combined computation in which the basic computation terminates in $e_t$, then there is an event $e_d$ in $C$, where $d > t$, such that some process declares termination in $e_d$.

### 5.5.1 Message Counting and Process Status

Let $(e_1, e_2, \ldots, e_m, \ldots)$ be any computation. With respect to this computation, we first state two simple facts concerning the numbers of unacknowledged basic messages.
Consider any two nonfaulty processes $p_i$ and $p_j$. Recall that $\text{out}_i[j]$ indicates the number of basic messages that $p_i$ has sent to $p_j$ for which $p_i$ has not received an acknowledgment, and that $\text{in}_j[i]$ is the number of basic messages from $p_i$ that $p_j$ has received but has not yet acknowledged. Also recall that the $c$ parameter of a $\text{SIGNAL}(c, -)$ message is meant to acknowledge $c$ basic messages. Let $b$ be the number of basic messages in the channel from $p_i$ to $p_j$ (which have been sent by $p_i$ but not yet received by $p_j$). Let $\text{SIGNAL}(c_1, -), \text{SIGNAL}(c_2, -), \ldots, \text{SIGNAL}(c_q, -)$ be all the $\text{SIGNAL}$ messages in the channel from $p_j$ to $p_i$ (which have been sent by $p_j$ but not yet received by $p_i$). We immediately have the following.

**Lemma 3** At any time during the computation, if $p_i$ and $p_j$ are both nonfaulty, then

$$\text{out}_i[j] = b + \text{in}_j[i] + \sum_{k=1}^{q} c_k.$$  

(5.1)

**Lemma 4** If process $j$ receives a basic message from process $i$ in event $e_k$ and does not send any $\text{SIGNAL}$ to $i$ during the period $[t_k, t_{k'}]$, where $k < k'$, then $\text{out}_i[j] > 0$ at time $t_{k'}$.

The terms *neutral* and *engaged* were employed in the description of our algorithm without formal definition. They basically mean the same things as in the DS algorithm, but there is a subtle difference. In the DS algorithm, a process $i$ is neutral iff $p_i$ is idle and $\text{in}_i = \text{out}_i = 0$. In our algorithm, a process $i$ is neutral if $p_i$ is idle, $\text{in}_i = 0$, $\text{out}_i \approx 0$ (i.e., $\text{out}_i[j] = 0$ or $\epsilon$ for every $j$), and $p_i$ has reported (via a signal) to its
parent parent_. (The difference is due to our design that \( p_i \) may increase the value of \( \text{out}_i[x] \) and adopt a new parent on detecting fault \( x \) in \( F4 \).) The above statement is not a definition of \textit{neutral}, because the expression "\( p_i \) has reported to its parent" is not unambiguous and it does not apply very well to the coordinator, who has no parent. The neutral/engaged status of a process is formally defined below.

\textbf{Definition 7} Every process is \textit{neutral} before it gets started, and becomes \textit{engaged} in the event of its initialization (i.e., \( \text{A0} \) action). An engaged process becomes \textit{neutral} in the last event of a "successful" \( \text{A4} \) action. A neutral process becomes \textit{engaged} in an \( \text{A2} \) action. Should a process fail, its engaged/neutral status stays unchanged thereafter.

In the above, a "successful" \( \text{A4} \) action is one in which the condition of the outer if-statement is true and so the inner if-command is executed.

The notion of the neutral/engaged status of a process plays a critical role in our correctness proof. It follows from \( \text{A4} \) and from the above definition that if process \( i \) is neutral, then it is idle, and \( \text{in}_i = 0 \), and \( \text{out}_i = 0 \).

\textbf{Definition 8} A neutral process \( i \) is said to be \textit{strongly} neutral if \( \text{out}_i = 0 \), and \textit{weakly} so otherwise.

Note that a strongly neutral process \( i \) becomes weakly neutral on setting \( \text{out}_i[x] := \epsilon \) in \( F1 \), in which a \texttt{FAULT-FLUSH.request}(x) is also issued. A weakly neutral process
becomes engaged on receiving a basic message. If it does not get engaged in this way, then after all its FAULT-FLUSH.requests have been confirmed, it will become strongly neutral again.

5.5.2 Structure of Engaged Processes

Definition 9 At any time $t$, the root of the system or system root, denoted by $r(t)$ (or simply by $r$ if there is no confusion), is the process whose index is smallest among all nonfaulty processes, i.e., $r(t) = \min\{i : i \text{ is nonfaulty at time } t\}$.

The system always has a root. When the current root fails, another process immediately succeeds (by definition) as the root. The new root, however, is not aware of its having become the root until later when it detects the failure of the previous root. When the root recognizes that it is the root, then it becomes the “coordinator.”

Definition 10 The system root $r$ is also called a coordinator if $\text{crd}_r = \text{true}$.

The variables $\text{parent}_i$ define a directed graph $G$ with vertex set $V = P$ and edge set $E = \{(i, \text{parent}_i) : \text{parent}_i \neq \text{NULL}\}$. The edges of the graph indicate whom a process will report to when it becomes strongly neutral. Though $G$ can be directly used in our correctness proof, we found it more convenient to deal with the following subgraph of $G$ that includes only essential edges and, as will be seen, has a more regular structure than $G$. 
Definition 11 The hierarchy graph of the system is the directed graph
\[ H = (V, E) \]
where \( V = P \) and \( E = \{ (i, \text{parent}_i) : i \neq r, p_i \text{ is engaged and nonfaulty} \} \).

Note that \( \text{parent}_i \neq \text{NULL} \) if \( p_i \) is engaged and is not the system root. So \( H \) is
well-defined and is really a subgraph of \( G \). The edges of \( G \) that are missing from \( H \)
are those emanating from faulty nodes, neutral nodes, and the root \( r \).

We distinguish between two types of edges in \( H \): engagement edges and fault-
detection edges.

Definition 12 An edge \( (i, \text{parent}_i) \) in \( H \) is said to be a fault-detection edge due to
\( p_x \) if \( \text{parent}_i \) was set in F1 because of \( p_i \) detecting the failure of \( p_x \). An engagement
edge is one that was created in A2 when \( p_i \) received a basic message.

It should be clear from the algorithm that an edge in \( H \) is either an engagement
edge or a fault-detection edge.

\( H \) is a dynamic graph, its structure changing during the computation. In the
following, we list all the events that may possibly change the structure of \( H \), and
then show that \( H \) is always a forest. (A forest is a number of mutually disjoint trees.)

There are six events that may add edges to or delete edges from \( H \):

E0: A process \( i \neq r \) initializes its local variables in execution of A0. (Edge \( (i, 1) \) is
introduced.)

E1: The present system root fails.
**E2:** An engaged nonfaulty process $x \neq r$ fails. (The edge emanating from $x$ disappears from $H$.)

**E3:** An engaged nonfaulty process $i \neq r$ executes the inner if-statement of A4 and becomes neutral. (The outbound edge of $i$ is deleted.)

**E4:** A neutral nonfaulty process $i \neq r$ receives a basic message and performs A2 (and thus becomes engaged). (A new edge $(i, parent_i)$ is introduced.)

**E5:** An engaged process $i \neq r$ updates the value of $parent_i$ (and other local variables) in $F1$. (A new edge $(i, parent_i)$ replaces an old one, if the value of $parent_i$ changes in $F1$.)

Note that each of these events is atomic and belongs in one of the event categories as listed in the beginning of Section 5.5. For instance, E4 is in category AE5, and E5 in AE4. Also note that in event E1 the edge from the new root, if it exists, is, by definition, immediately excluded from $H$, even though $parent_r$ may still have a non-NULL value. Defining $H$ like this results in a simple structure that would otherwise be hard to describe. The following lemma shows that $H$ is a forest of trees. The four categories (of trees) described in the lemma are not necessarily disjoint: for instance, a tree in category 3 may also be in category 2 and/or category 4, because a former coordinator could be engaged or neutral. Figure 8 shows the structure of an example $H$ at various times.
Lemma 5  At any time \( t \) during the computation, \( H \) is a forest comprising the following trees:

1) A single tree rooted at the system root — called the main tree;
2) Zero or more trees rooted at engaged, faulty nodes;
3) Zero or more trees rooted at former coordinators;
4) Zero or more neutral isolated nodes (faulty or nonfaulty).

Proof. Let \( (e_1, e_2, \ldots, e_m) \) be any computation. At \( t_0 \), before any event occurs, every process is neutral and there is no edge at all in \( H \). So \( H \) contains the main tree and \( n - 1 \) trees of type 4 — the lemma is true at the beginning. (The main tree is also of type 4, but that doesn’t matter.)

Assume that the lemma is true at time \( t_{k-1} \), before event \( e_k \). We show that the lemma remains valid after the event (i.e., at \( t_k \)). It suffices to consider the events \( E_0 - E_5 \) since other events do not change the structure of \( H \).

\( E_0 \): A process \( p_i, i \neq r \), executes \( A0 \), resulting in edge \( \langle i, 1 \rangle \) being added to \( H \). Before the event, \( p_i \) constitutes a single-node tree in category 4. In the event, \( p_i \) joins the tree rooted at \( p_1 \), which is in category 3 or 1 depending on whether \( p_1 \) is faulty or non-faulty at \( t_{k-1} \). The lemma holds at \( t_k \).

\( E_1 \): The present root of the system \( r \) fails. In this case, a process \( r' \) succeeds as the new system root and edge \( \langle r', \text{parent}_r \rangle \), if it existed at \( t_{k-1} \), now disappears from \( H \) (for \( H \) does not include any edge emanating from the system root). Thus, after
the event, the tree rooted at \( r \) becomes a tree in category 3, and the subtree rooted at \( r' \) becomes the main tree. The lemma is still true.

\[ \text{E2: An engaged nonfaulty process } x \neq r \text{ fails. In this event, edge } (z, \text{parent}_x) \text{ disappears from } H \text{ (by definition } H \text{ does not include outgoing edges of faulty processes) and the original subtree rooted at } x \text{ now becomes an individual tree with } x \text{ at the root. The new tree is in category 2. The tree from which } x \text{ breaks off is still a tree. So the lemma remains valid after the failure of } x. \]

\[ \text{E3: An engaged nonfaulty process } i \neq r \text{ becomes neutral in the last event of A4. The edge } (i, \text{parent}_i) \text{ is deleted from } H. \text{ The precondition } out_i = 0 \text{ together with the assumption } i \neq r \text{ implies that node } i \text{ has no child in } H \text{ (for, by Lemma 3, } in_j[i] = 0, \text{ and thus } \text{parent}_j \neq i \text{ for every nonfaulty process } j.). \text{ Thus, after the event, } i \text{ becomes a neutral isolated node and the lemma holds.} \]

\[ \text{E4: A neutral nonfaulty process } i \neq r \text{ receives a basic message from } p_j \text{ (and hence executes A2) and becomes engaged. There are two cases depending on whether } p_i \text{ is strongly or weakly neutral.} \]

\[ \bullet \text{parent}_i = \text{NULL before A2 (} p_i \text{ strongly neutral)} \text{ — Process } j \text{, whose message makes } i \text{ engaged, must be engaged itself (whether it is faulty or nonfaulty) and therefore belongs in a tree in categories 1-3. Setting } \text{parent}_i \text{ to } j \text{ attaches the tree rooted at } i \text{ to the tree containing } j \text{ without violating the lemma.} \]
• \textit{parent}, \neq \text{NULL} before the action (\textit{p}, weakly neutral)— In this case, \textit{parent},
points either to the present root or to a former root of the system. The edge
\langle i, \textit{parent}_i \rangle, which was not in \textit{H} because of \textit{i}'s neutral status, is now added to \textit{H}. It
connects the tree containing \textit{i} either to the main tree or to a tree in category 3. The
lemma remains valid.

\textbf{E5:} An engaged process \textit{i} \neq r performs action \textbf{F1}. This results in process \textit{i}
changing its parent in \textit{H} either to the present system root \textit{r} or to a former one.
Again, the lemma remains true. \hfill \Box

5.5.3 Completeness

The following theorem establishes the completeness property of our algorithm.

\textbf{Theorem 4} \textit{Given any complete combined computation} \textit{C} = (e_1, e_2, \ldots, e_t, \ldots) \textit{in}
\textit{which the basic computation terminates in event} \textit{e_t}, \textit{there is an event} \textit{e_d} \textit{in} \textit{C}, \textit{where}
\textit{d} > \textit{t}, \textit{such that some process declares termination in} \textit{e_d}.

\textbf{Proof.} Let \textit{C} = (e_1, e_2, \ldots, e_t, \ldots) \textit{be a complete combined computation in which}
the basic computation terminates in event \textit{e_t}. We first show that \textit{C} contains only a
finite number of events..

As the basic computation has terminated, events occurring after \textit{e_t} in \textit{C} are those
of the termination detector. Each action of our algorithm consists of a finite number
of events. Thus, \textit{C} is finite if the number of actions occurring after \textit{e_t} is finite. Actions
A0, A1, and A2 do not occur after $e_i$. At most $O(n)$ F1s (and the same for F2s, and A3s) may occur at each process after $e_i$. There are at most $O(M + n^2)$ in-transit SIGNALs — $O(M + n)$ for acknowledging basic messages and $O(n^2)$ for fault reporting — so the number of A5s that may occur is finite. Since the execution of A4 depends on the occurrence of A3, A5, or F2, the number of A4 occurrences is also finite. This establishes that the number of events in $C$ is finite.

Let $C = (e_1, e_2, \ldots, e_t, \ldots, e_m)$. Since $C$ is a finite complete computation, by Definition 6 there is at least one nonfaulty process at time $t_m$, and no other events will occur after time $t_m$ unless some currently nonfaulty process will crash (after $t_m$). Obviously, $m > t$, since a basic computation always ends with either a process becoming idle or with a process crashing, in either case there being actions after $e_t$.

Let $F$ be the set of all faulty processes as of time $t_m$. By condition ii.a), $F \neq P$. Let $e_k$ be the last event in $C$ that occurs at a process $i \notin F$. Due to condition ii.b), the following observations are true at $t_k$, the time before event $e_k$.

1. Process $i \notin F$ has detected all faults in $F$ (i.e., $DF_i(t_k) = F$), and all those F1, A3, and F2 actions associated with detecting of the failures in $F$ have been finished. This also means that process $i$ has no unfinished (unconfirmed) fault-flush requests.

2. The A3 action triggered because of process $i$ becoming idle has been finished.

3. No basic message or SIGNAL is in transit to $i$. 

From these observations, one may readily verify that \( e_k \) is an event associated with A4. Defining a *leaf* to be a node without any child, we make two claims.

*Claim 1:* If in computation \( C \) process \( i \notin F \) is a leaf (in the hierarchy graph \( H \)) at time \( t_{k-1} \) and remains so thereafter, then the inner if-elseif-statement of A4 is executed in event \( e_k \).

*Proof of Claim 1:* It suffices to show that the condition of the outer if-statement of A4 is satisfied at time \( t_{k-1} \). Obviously, process \( i \) is idle at \( t_{k-1} \) as \( e_k \) is its last event. Since \( e_k \) is an event associated with action A4, it follows from observation 1 that \( RF_i(t_{k-1}) \subseteq F = DF_i(t_{k-1}) = DF_i(t_{k-1}), \) and that \( in_i[z] = 0 \) and \( out_i[x] = 0 \) for all \( x \in F \). Observation 2 implies that \( in_i[j] = 0 \) for all \( j \neq \text{parent}_i \). Since \( i \) is a leaf, all basic messages that \( i \) has sent to other nonfaulty processes must have been acknowledged; i.e., \( in_j[i] = 0 \) for all nonfaulty \( j \) and no basic message is in transit from \( i \) to any nonfaulty \( j \). (Otherwise, if \( in_j[i] \neq 0 \) for some nonfaulty \( j \), then when \( j \) becomes idle or when it becomes a coordinator, \( j \) will send a SIGNAL to \( i \), contradicting the assumption that \( e_k \) is the very last event at \( i \). Similarly, if there exists a basic message in transit from \( i \) to a nonfaulty \( j \), then either \( j \) will become a child of \( i \), or \( j \) will send a SIGNAL to \( i \) upon becoming idle or upon becoming a coordinator — a contradiction in either case.) By observation 3 and Lemma 3, \( out_i[j] = 0 \) for all \( j \notin F \). Thus, at time \( t_{k-1} \), \( out_i = \vec{0} \) and \( in_i[j] = 0 \) for all \( j \neq \text{parent}_i \), and the claim is proved.
Claim 2: At time $t_{m-1}$, $r$ is a leaf in $H$.

Proof of Claim 2: Let $e_i$ be any event between $e_t$ and $e_m$ such that by time $t_i$ not only all faults in $F$ have been detected by all nonfaulty processes, but all $F_1$ and $F_2$ actions associated with these fault detections have been finished. After time $t_i$, no edge will be added to $H$; if $H$ changes shape, it will be that edges are removed in $E_3$ events. Consider any leaf $i$ in the main tree (the one rooted at $r$) as of time $t_i$. If $i = r$, we have verified $r$'s leaf status. If $i \neq r$, we show that $i$'s very last event (i.e., $e_{k_i}$) occurred after time $t_i$. Assume otherwise that $e_{k_i}$ occurred before $t_i$. At that time, process $i$ was already a leaf. This is because 1) $i$'s former children (former as relative to time $t_i$ at which $i$ has no child), if any, each sent a SIGNAL to $i$ when terminating their child/parent relationship with $i$; and 2) these signals obviously had arrived at $i$ before the occurrence of $e_{k_i}$. Thus, by Claim 1, $i$ executed the inner if-statement of A4 and terminated its child/parent relationship with its parent, contradicting the fact that $i$ still has a parent at $t_i$. So, $e_{k_i}$ could not have occurred before $t_i$: it must have occurred after $t_i$ (while $i$ is a leaf). By Claim 1 again, $i$ executes the inner if-statement of A4 and leaves the main tree in event $e_{k_i}$. Applying this argument to each leaf leads to the conclusion that every leaf in the main tree eventually leaves the tree. So by the time $e_m$ occurs, $r$ itself has become a leaf, and claim 2 is proved. □

Now we show that process $r = \min(P - F)$ declares termination in event $e_m$. First, $r$ already recognizes itself as the coordinator by time $t_{m-1}$, since all faults have
been detected. Second, \( e_m \) is an event at process \( r \), as otherwise there would be a SIGNAL in transit at time \( t_m \) towards a \( j \notin F \). Third, by Claim 1 and Claim 2, process \( r \) executes the elseif statement of A4. So we only have to show that the condition \( \forall j \in P - DF_r - \{r\}: \text{NumFault}_r[j] = |DF_r| \) holds true at time \( t_{m-1} \), where we already know \( DF_r = F \). Note that every nonfaulty process except for \( r \) has once changed its parent to \( r \) (after detecting the last fault in \( F \)), but now \( r \) has no child at time \( t_{m-1} \). So, each process \( j \in P - F - \{r\} \) must have sent a SIGNAL(\( - , S \)) to \( r \) with \( S = F \), and, by observation 3, the signal has been received by \( r \) by time \( t_{m-1} \). So \( \text{NumFault}_r[j] = |F| \) for all \( j \in P - F - \{r\} \) at time \( t_{m-1} \) and, therefore, \( r \) declares termination in event \( e_m \). (Note that \( r \) may or may not be the only process that declares termination in \( C \); some other processes — former coordinators — might have done so before \( r \).) This proves the theorem.

\( \square \)

5.5.4 Soundness

We now prove the soundness of our algorithm, which, as will be seen, is a much harder task than proving the completeness.

Outline

Suppose in a combined computation \( (e_1, e_2, \ldots, e_d) \) some process \( r \) declares termination in event \( e_d \). (Process \( r \) must be the coordinator of the system, since only the coordinator may declare termination, owing to the design of A4.) We show that the
basic computation has terminated by time $t_d$. As $e_d$ is not an event of the basic computation, it will then follow that the basic computation had actually terminated by time $t_{d-1}$.

We shall establish three important properties of the algorithm:

1. At time $t_d$, if $x \in DF_r$ (i.e., $x$ has been detected by $r$ to be faulty), then every nonfaulty process has reported fault $x$ to $r$ in a SIGNAL.

2. At time $t_d$, if $x$ is an engaged faulty process, then $x \in DF_r$.

3. At time $t_d$, there is no edge in the hierarchy graph $H$.

From these properties, the soundness of the algorithm can be readily proved as follows.

**Theorem 5** If some process declares termination, then the basic computation is terminated at that time.

**Proof.** Assume that in a computation $(e_1, e_2, \ldots, e_d)$ some process $r$ declares termination in event $e_d$. Property 3 and Lemma 5 together imply that, at time $t_d$, all nonfaulty processes are neutral and hence idle. (Note that $r$ is also neutral because it executes action $A4$ and declares termination.) As neutral processes have no unacknowledged basic messages (by definition and Lemma 3), only a channel $[x, i]$ from an engaged faulty process $x$ to a nonfaulty one $i$ may possibly contain deliverable basic messages. Properties 1 and 2 together indicate that $i$ has reported $x$ to $r$ in a SIGNAL, which, according to the algorithm, may happen only if $i$ has received a
FAULT-FLUSH.confirm(x) primitive (see F2). Thus, channel [x, i] is free from basic messages. By definition, the basic computation is terminated at td.

So, all what we need to do is to prove the three mentioned properties.

**Property One**

This section establishes property one: If x ∈ DFr(td), then every nonfaulty process (except for r itself) has reported x’s failure to r through a SIGNAL. (Recall that DFr(td) is the value of DFr as of time td.)

**Lemma 6** For each i ∈ P − DFr(td) − {r}, process r has received by time td a SIGNAL(−, Si) from i with Si = DFr(td).

**Proof.** Suppose i ∈ P − DFr(td) − {r}. When r declares termination in event \( e_d \), these two conditions hold: \( RF_r ⊆ DF_r \) and \( |DF_r| = NumFault_r[i] \). From the statements of A5, we know \( NumFault_r[i] ≤ |RF_r| \). Combining all these conditions yields \( |RF_r| = |DF_r| = NumFault_r[i] \), which implies that coordinator r has received a SIGNAL(−, Si) from i such that \( Si = RF_r = DF_r \). □

**Property Two**

We establish property two in this section: if a faulty process x is engaged when it fails, then x ∈ DFr(td). The task turns out to be very difficult. Our strategy is to identify all faults that are definitely detected by r at time td. We will define a class
of faults called critical faults which includes engaged faults, and show that all critical faults are detected by \( r \) at time \( t_d \).

Let \((e_1, e_2, \ldots, e_d)\) be the computation in which \( r \) declares termination in event \( e_d \).

If \( r = 1 \), let \( c = 0 \); otherwise, let \( e_c \) be the event in which \( r \) becomes the system root.

We will focus on the period between \( t_c \) and \( t_d \) during which \( r \) is the system root.

**Definition 13** At any time \( t \in [t_c, t_d] \), a faulty process \( x \) is said to be a critical fault if at least one of the following is true:

- \( x \) is a former system root (i.e., \( x < r \), where \( r \) is the system root at time \( t_d \));
- \( x \) was engaged at the time of its failure;
- \( x \) was once detected by some engaged process (i.e., there was a time \( t' \leq t \), and a process \( i \), such that \( i \) was engaged at \( t' \) and \( z \in DF_i(t') \)).

Let \( CF(t) \) denote the set of all critical faults as of time \( t \). \( CF(t) \) is a "nondecreasing" function of \( t \), since a fault remains critical once it is critical.

Our goal here is to show \( CF(t_d) \subseteq DF_r(t_d) \). The question facing us is: Under what conditions can one assert that a certain fault \( x \) will definitely be in \( DF_r(t_d) \)?

The definition of coordinator suggests one such condition: every former system root is in \( DF_r(t_d) \), since by time \( t_d \) process \( r \) already recognizes itself as the coordinator (or it wouldn’t have declared termination). We show some other conditions in Lemmas 9 and 10. These conditions will be used to prove property two in Lemma 11.
Call an ordered pair of processes \( \langle a, b \rangle \) a \emph{semi-engagement edge} if \( \text{in}_a[b] > 0 \). This term is only introduced to facilitate the next definition. Note that an engagement edge is always a semi-engagement edge, and a semi-engagement edge may or may not be an edge in the hierarchy graph.

**Definition 14** A sequence of \( l \geq 1 \) nonfaulty processes \( \langle a_1, \ldots, a_l \rangle \) is said to be a \emph{semi-engagement chain} if the first edge \( \langle a_1, a_2 \rangle \), if it exists, is a semi-engagement edge and the rest of the edges (i.e., \( \langle a_2, a_3 \rangle, \ldots, \langle a_{l-1}, a_l \rangle \)), if they exist, are all engagement edges.

If \( \langle a_1, \ldots, a_l \rangle \) is a semi-engagement chain, \( a_{l+1} \) is said to be the \emph{predecessor} of \( a_l \) in the chain. In most cases, the predecessor of a node is also its parent in the hierarchy graph. By definition, every nonfaulty process \( a \) alone constitutes a semi-engagement chain \( \langle a \rangle \), whether \( a \) is neutral or engaged. If a semi-engagement chain has a non-zero length, every process on it is engaged and nonfaulty.

**Definition 15** Denote by \( \text{\overline{a} \leftarrow b \ at \ t_k} \) that at time \( t_k \) process \( x \) is faulty and there exists a semi-engagement chain \( C \) from \( b \) to \( a \) such that at least one of the following statements is true:

(a) \( x \in DF_b(t_k) \).

(b) All edges in \( C \) are engagement edges and \( \langle x, b \rangle \) is a semi-engagement edge.

(c) All edges in \( C \) are engagement edges and there is some \( \text{SIGNAL}(c, S) \) in transit for \( b \) such that \( c > 0 \) and \( x \in S \).
The intuitive meaning of "$a \leftarrow b$ at $t$" is that, if no process on the chain (from $b$ to $a$) ever fails after $t$, then the information "$x$ is faulty" will propagate along the chain and eventually reach $a$. This property is due to the following lemma that indicates that if $a \leftarrow b$ holds at $t_k$, then after time $t_k$, $b$ may be neutral only if it has detected $x$'s failure.

**Lemma 7** If $a \leftarrow b$ holds at time $t_k$, then for all $k'$, $k \leq k' \leq d$, either $x \in DF_b(t_k')$ or $b$ is engaged at $t_k'$.

**Proof.** Assume $a \leftarrow b$ holds at $t_k$. Also, assume $x \notin DF_b(t_k)$, or we are done. Thus, condition (b) or (c) of Definition 15 holds. In either case, $b$ still has unacknowledged outgoing messages (i.e., $out_b[j] \geq 1$ for some $j$) as of time $t_k$. So, $b$ is engaged at time $t_k$, and it remains so as long as $out_b[j] \geq 1$ for some $j$. In case (b), $out_b[x]$ remains positive until $b$ has detected $x$'s failure and subsequently finished the corresponding $F_1$ and $F_2$ actions. In case (c), $out_b$ may become 0 only after $b$ has received the SIGNAL containing $x$. Thus, $b$ may become neutral only after it has detected $x$'s failure. □

In the algorithm, a SIGNAL is sent either in $A_3$ or in $A_4$. A SIGNAL sent in an $A_4$ action is called a major SIGNAL.

**Definition 16** If $a \neq z$ and $a \neq r$, we denote by "$r \leftarrow a$ at time $t_k$" that by time $t_k$ process $a$ has not yet sent a major SIGNAL($-, S$) to $r$ with $z \in S$. 


Definition 17 Write "\( r \not< a \not< b \) at \( t_k \)" iff both \( r \not< a \) and \( a \not< b \) hold true at time \( t_k \).

Intuitively, if \( r \not< a \not< b \) at \( t_k \) and no process in the chain (from \( b \) to \( a \)) ever fails after time \( t_k \), then the information about \( z \)'s failure will propagate along the chain to \( a \) and be reported together with \( z \) by \( a \) to root \( r \). Before we proceed to prove this, we first establish some conditions that ensure \( r \not< a \).

Lemma 8 If any one of the following statements is true, then \( r \not< a \) holds at time \( t_k \).

\begin{enumerate}
  \item \( (a,j) \) is a fault-detection edge in \( H(t_k) \) due to a faulty process \( z \).
  \item \( (a,z) \) is an edge in \( H(t_k) \).
  \item Process \( a \) becomes engaged in event \( e_k \) because of receiving a basic message from process \( z \).
\end{enumerate}

**Proof.** In case \( a) \), edge \( (a,j) \) would no longer exist in \( H(t_k) \) if \( a \) has reported the failure of \( z \) in a major SIGNAL to \( r \). In case \( b) \), \( a \) has not detected \( z \)'s failure as of time \( t_k \). In case \( c) \), one observes that \( a \) may report \( z \) in a major SIGNAL only after it has received the FAULT-FLUSH.confirm(\( z \)). But after the FAULT-FLUSH.confirm(\( z \)), no basic message from \( z \) will be received by \( a \). \( \square \)

We are now ready to show that under certain conditions it can be concluded that \( z \in DF_r(t_d) \). This is done in the next two lemmas.
Lemma 9 Assume \( z \in DF_r(t_d) \) and \( a \neq r \). If, by time \( t_d \), \( r \) has not received a major \( SIGNAL(-, S) \) with \( z \in S \) from process \( a \), then \( a \in DF_r(t_d) \). In particular, if \( r \neq a \) at time \( t_d \), then \( a \in DF_r(t_d) \).

Proof. We show that if \( a \notin DF_r(t_d) \), where \( a \neq r \), then \( r \) has received from \( a \) by time \( t_d \) a major \( SIGNAL(-, S) \) with \( z \in S \) for all \( z \in DF_r(t_d) \).

By Lemma 6, process \( r \) has received a \( SIGNAL(-, S) \) with \( S = DF_r(t_d) \) from \( a \). This \( SIGNAL \) is evidently a major one (i.e., at the time the \( SIGNAL \) is sent, \( parent_a = r \)), since \( parent_a \) was set to \( r \) after \( a \) had detected all faults in \( S \). □

We next show that if \( r \not< b \) or \( r \not> a \not< b \) holds at some point of time between \( t_c \) and \( t_d \), where \( z \in DF_r(t_d) \), then \( x \in DF_r(t_d) \).

Lemma 10 If \( r \not< b \) or \( r \not> a \not< b \) at time \( t_k \in [t_c, t_d] \), where \( z \in DF_r(t_d) \), then \( x \in DF_r(t_d) \).

Proof. The proof is by induction on \( t_k \), for \( k = d, d - 1, \ldots, c \). Recall that event \( e_k \) occurs between time \( t_{k-1} \) and time \( t_k \).

Induction Base: Suppose \( r \not< b \) or \( r \not> a \not< b \) holds at time \( t_d \) for some \( z \in DF_r(t_d) \). The latter case is impossible since by Lemma 9 every nonfaulty process (\( a \) in particular) has reported \( z \) to \( r \) by time \( t_d \). In the case of \( r \not< b \), it must be \( r \not< r \) as \( r \) is neutral at \( t_d \). It follows from Lemma 7 that \( x \in DF_r(t_d) \).

Induction Hypothesis: Assume the lemma is true for \( t_{k+1}, t_{k+2}, \ldots, t_d \), where \( k \geq c \).
Induction Step: We show the lemma true for $t_k$. Thus, assume \( r \xleftarrow{z} b \) at time $t_k$ or \( r \xrightarrow{z} a \) at time $t_k$ with $z \in DF_r(t_d)$. We need to show $x \in DF_r(t_d)$. Let $C$ be the semi-engagement chain involved in $r \xleftarrow{z} b$ or in $a \xrightarrow{z} b$, and for simplicity let $k' = k + 1$. Consider all possible cases for $e_{k'}$, the event occurring between $t_k$ and $t_{k'}$.

E1: The present coordinator $r$ crashes in event $e_{k'}$. This case is impossible by definition of $t_c$.

E2: A process $y \neq r$ in $C$ fails in event $e_{k'}$. By Definition 15(b) and 16, we have the following:

- If $r \xleftarrow{z} b$ at time $t_k$, then $r \xleftarrow{z} p$ at time $t_{k'}$, where $p$ is the predecessor of $y$ in chain $C$.

- If $r \xrightarrow{z} a \neq b$ at $t_k$ and $y \neq a$, then $r \xrightarrow{z} a \xrightarrow{z} p$ at $t_{k'}$, where $p$ is the predecessor of $y$ in chain $C$.

- If $r \xleftarrow{z} a \xrightarrow{z} b$ at $t_k$ and $y = a$, then $r \xleftarrow{z} a$ at $t_{k'}$, which implies $r \xrightarrow{z} a$ at $t_d$.

In the first two cases, the induction hypothesis indicates that $y \in DF_r(t_d)$. In the third case, $y (= a)$ is also in $DF_r(t_d)$ as implied by Lemma 9. So, in all cases, $y \in DF_r(t_d)$, and as $y$'s failure occurs after $x$'s, it follows from Lemma 9 that $x \in DF_r(t_d)$.

E3: An engaged process in $C$ either sends a SIGNAL($c, S$) to its predecessor $p$ in the chain or becomes neutral (or both). This process must be $b$, for the only process in a semi-engagement chain that may send a SIGNAL to its predecessor or become
neutral is the process where the chain starts. First observe that if \( b \) becomes neutral in \( e_{k'} \) then \( x \in DF_b(t_{k'}) \) (by Lemma 7), which implies \( x \in DF_b(t_k) \) since \( e_{k'} \) occurs in an A4 action in which \( DF_b \) does not change value. There are three different cases which we need to consider separately:

(i) \( b \) becomes neutral in \( e_{k'} \) without sending a SIGNAL to its predecessor in \( C \). This case is possible only if \( b \) has no predecessor in \( C \) (i.e., \( b = r \) or \( b = a \)).

- If \( r \leftarrow r \) at \( t_k \), then \( x \in DF_r(t_d) \).
- If \( r \neq a \leftarrow a \) at \( t_k \), then \( a \) sends a SIGNAL to \( parent_a \) in \( e_{k'} \). The SIGNAL contains \( x \). If \( parent_a \neq r \), then \( r \leftarrow a \) still holds at \( t_{k'} \) and hence \( x \in DF_r(t_d) \) by the induction hypothesis. Otherwise, suppose \( parent_a = r \) (so the SIGNAL is destined for \( r \)). If the SIGNAL is received by \( r \) before time \( t_d \), then \( x \in DF_r(t_d) \). If it is not received before \( t_d \), then no signal from \( a \) that contains \( z \) is received before \( t_d \). By Lemma 9, \( a \in DF_r(t_d) \), and hence \( x \in DF_r(t_d) \) (since \( r \neq x \)).

(ii) \( b \) sends a SIGNAL(\( c, S \)) to \( p \) and becomes neutral. In this case, \( c > 0 \) (because \( in_b[p] > 0 \) at \( t_k \)) and \( x \in S \). By Definition 15(c), the following hold:

- If \( r \leftarrow b \) at \( t_k \), where \( r \neq b \), then \( r \leftarrow p \) at \( t_{k'} \).
- If \( r \neq a \leftarrow b \) at \( t_k \), where \( a \neq b \), then \( r \neq a \leftarrow p \) at \( t_{k'} \).

In both cases, \( x \in DF_r(t_d) \) by the induction hypothesis.

(iii) \( b \) sends a SIGNAL(\( c, S \)) to \( p \) and remains engaged. In this case, \( e_{k'} \) occurs in an A3 action, which implies that \( in_b[p] > 0 \) and \( (b, p) \) is not an engagement edge.
Condition (a) of Definition 15 holds at time $t_k$ (i.e., $x \in DF_k(t_k)$) and thus $c > 0$ and $x \in S$. An argument similar to (ii) yields $x \in DF_r(t_d)$.

E5. A process $q$ in $C$ performs F1 (in which it updates $parent_q$ and thus may affect chain $C$). If $q = r$ or $q = a$, then $r \underset{\succ}{\leftarrow} b$ or $r \nLeftarrow a \underset{\succ}{\leftarrow} b$ continues to hold at $t_{k'}$ and thus $x \in DF_r(t_d)$ by the induction hypothesis. So assume $q \neq r$ and $q \neq a$. Let $y$ be the fault that triggered the F1 action.

- If $r \underset{\succ}{\leftarrow} b$ at time $t_k$, then using the subchain (of $C$) from $q$ to $r$, we have $r \Upnu q$ at $t_{k'}$ by Definition 15(a). By the induction hypothesis, $y \in DF_r(t_d)$. Since $q$ just detected $y$, $r \nLeftarrow q$ at $t_{k'}$. Combining it with $q \underset{\succ}{\leftarrow} b$ yields $r \nLeftarrow q \underset{\succ}{\leftarrow} b$ at $t_{k'}$, which implies $x \in DF_r(t_d)$ by the induction hypothesis.

- If $r \nLeftarrow a \underset{\succ}{\leftarrow} b$ at $t_k$, then $r \nLeftarrow a \Upnu q$ at $t_{k'}$. An argument similar to the preceding one yields $r \nLeftarrow q \underset{\succ}{\leftarrow} b$ at $t_{k'}$ and, hence, $x \in DF_r(t_d)$.

Others. It is not hard to see that in any other event (in particular, E0 or E4), $r \underset{\succ}{\leftarrow} b$ at $t_k$ and $r \nLeftarrow a \underset{\succ}{\leftarrow} b$ at $t_k$ imply $r \underset{\succ}{\leftarrow} b$ at $t_{k'}$ and $r \nLeftarrow a \underset{\succ}{\leftarrow} b$ at $t_{k'}$, respectively, and so $x \in DF_r(t_d)$ by the induction hypothesis.

We are now in a position to prove $CF(t_d) \subseteq DF_r(t_d)$, from which will follow the second property of the algorithm that by time $t_d$ all engaged faults are detected by coordinator $r$.

**Lemma 11** $CF(t_d) \subseteq DF_r(t_d)$.

**Proof.** We show by induction that $CF(t_k) \subseteq DF_r(t_d)$ for all $k$, $c \leq k \leq d$. 
Induction Base. For the induction base, we have $CF(t_c) \subseteq DF_r(t_d)$. The induction base holds because the failure that occurred in event $e_c$ has been detected by $r$ by time $t_d$; and every fault that occurred before $t_c$ is in $DF_r(t_d)$, as implied by Lemma 9.

Induction Hypothesis. Now assume as the induction hypothesis that $CF(t_{k-1}) \subseteq DF_r(t_d)$, where $k > c$.

Induction Step. We show $CF(t_k) \subseteq DF_r(t_d)$.

Claim: If $a$ is an engaged process at time $t_k$ such that 1) $a \leftarrow b$ at $t_k$, and 2) event $e_k$ involves no SIGNAL sending and does not change the path from $a$ to the root of the tree in the hierarchy graph that contains $a$, then $x \in DF_r(t_d)$.

To prove the claim, let $T$ be the tree in the hierarchy graph that contains $a$. By Lemma 5, $T$ is rooted at $r$ or at a faulty node $y$. Let $C$ be the path from $a$ to the root of $T$. By condition (2) of the claim, $C$ does not change in event $e_k$. There are three possibilities:

(i) $T$ is rooted at $r$ and all edges in $C$ are engagement edges. (This does not exclude the possibility that $a = r$.) In this case, we have $r \leftarrow b$ at $t_k$, from which it follows, by Lemma 10, that $x \in DF_r(t_d)$.

(ii) $T$ is rooted at $r$ and there is a non-engagement edge in $C$. The edge $\langle c, r \rangle$, where $c$ is the child of $r$ on $C$, is a fault-detection edge that was introduced into $H$ due to $c$’s detection of some fault $z$. (This edge is the only non-engagement edge on the path). At time $t_{k-1}$, as a node with a parent in $H$, $c$ is an engaged process and so
$z$ is critical. By the induction hypothesis, $z \in DF_r(t_d)$. By Lemma 8(a), $r \not\leftrightarrow c$ holds at time $t_{k-1}$; and it continues to hold at time $t_k$ since no SIGNAL is sent in event $e_k$. Thus $r \not\leftrightarrow c \not\leftarrow b$ at $t_k$ and, by Lemma 10, $x \in DF_r(t_d)$.

(iii) The root of $T$, say $y$, is an engaged faulty node or a former coordinator. Since $y$ is either an engaged process or a former coordinator, $y \in CF(t_{k-1})$ by definition, and thus $y \in DF_r(t_d)$ by the induction hypothesis. Let $c$ be the child of $y$ on path $C$. By Lemma 8(b), $r \not\leftrightarrow c$ holds at time $t_{k-1}$, and it continues to hold at $t_k$. So $r \not\leftrightarrow c \not\leftarrow b$ at $t_k$ and, by Lemma 10, $x \in DF_r(t_d)$.

So the Claim is correct, and with it the induction step is relatively easy to establish. Let $x$ be any new critical fault introduced in event $e_k$, if any. We show $x \in DF_r(t_d)$. Such a fault may be introduced only in an $E_2$, $E_4$, or $E_5$ event, where $E_1$, $E_2$, etc. refer to the same events as listed in the paragraph preceding Lemma 5. Consider the three events separately:

**E2.** An engaged process $x \neq r$ fails in event $e_k$ (so $x$ is a new critical fault). Let $y = \text{parent}_x$.

(i) If $y$ is nonfaulty at $t_k$ and $(x,y)$ is an engagement edge at $t_{k-1}$, then $y \not\leftarrow y$ at $t_k$ (by Definition 15(b)), from which it follows $x \in DF_r(t_d)$ (by the Claim).

(ii) If $y$ is nonfaulty at $t_k$ and $(x,y)$ is a fault-detection edge due to some fault $w$, then $r \not\leftrightarrow x$ holds at time $t_k$ (by Lemma 8(a)). Since $w \in DF_w(t_{k-1})$, $w$ is a critical fault at $t_{k-1}$. By the induction hypothesis, $w \in DF_r(t_d)$. Since $r \not\leftrightarrow x$ at $t_k$ and $x$ is
faulty at $t_k, r \not\prec x$ at $t_d$. By Lemma 9, $x \in DF_r(t_d)$.

(iii) If $y$ is faulty at $t_k$ (and hence at $t_{k-1}$ as well), then $y$ is either an engaged faulty process or a former coordinator. By definition $y \in CF(t_{k-1})$; and by the induction hypothesis, $y \in DF_r(t_d)$. Evidently, $r \not\prec x$ at $t_d$ (since $y$ fails before $x$), and so $x \in DF_r(t_d)$ (by Lemma 9).

E4. A neutral process $b$ becomes engaged in event $e_k$. Let $x \in DF_b(t_k)$ be any fault that becomes critical because of $b$'s new engagement due to the receipt of a basic message from, say, process $a$.

(i) If $a$ is nonfaulty at $t_k$, then by Definition 15(a), $a \prec b$ at $t_k$, and by the Claim, $x \in DF_r(t_d)$.

(ii) Now if $a$ is faulty at $t_k$, then it was faulty at $t_{k-1}$, also. By definition, $a \in CF(t_{k-1})$; and by the induction hypothesis, $a \in DF_r(t_d)$. By Lemma 8(c), $r \not\prec b$ holds at $t_k$. Evidently, $r \not\prec b \preceq b$ holds at $t_k$, so $x \in DF_r(t_d)$ (by Lemma 10).

E5. An engaged process $b \not= r$ performs $F1$ after detecting a fault $x$. Let $a$ be the original parent of $b$; i.e., $a = parent_b$ as of time $t_{k-1}$. There are three possibilities:

(i) $a$ is nonfaulty at $t_k$ and $(b, a)$ was an engagement edge at $t_{k-1}$, (ii) $a$ is nonfaulty at $t_k$ and $(b, a)$ was a fault-detection edge due to some fault $w$, and (iii) $a$ is faulty at $t_k$. The arguments for these cases are similar to those for cases E4(i), E2(ii), and E4(ii), respectively. □
Property Three

With properties 1 and 2 having been established, the remaining one is relatively easy to prove.

Lemma 12  When \( r \) declares termination at time \( t_d \), the hierarchy graph \( H \) contains no edge.

Proof. Assume for contradiction that \( E(H) \neq \emptyset \) at \( t_d \). By Lemma 5, there is a nonfaulty engaged process \( i, i \neq r \) with \( \text{parent}_i \) pointing to either \( r \), or a faulty engaged process, or a former coordinator. In the first case, since \( r \) is neutral at \( t_d \), the edge \((i, r)\) must be a fault-detection edge due to detection of some fault \( x \). By definition, \( x \) is a critical fault because of \( i \)'s engaged status. In the other two cases, \( y = \text{parent}_i \) is readily seen to be critical. As a critical fault, \( x \) (or \( y \)) belongs to \( DF_r(t_d) \) (by Lemma 11). It follows from Lemma 9 that \( x \) (or \( y \)) has been reported to \( r \) by \( i \) through a major SIGNAL, in contradiction to Lemma 8. Therefore, \( E(H) = \emptyset \).

5.6 Performance Analysis and Comparison

In this section, we analyze the performance of our algorithm and compare it with the V algorithm.

Let \( M \) be the total number of basic messages sent during the entire basic computation (or just those sent in DT mode if the feature of postponed start is implemented),
and let \( n \) be the number of processes, \( k \) the maximum number of faults tolerable, and \( f \) the number of actual faults.

- **#Faults Tolerable** — Our algorithm can tolerate any number of faults. The \( V \) algorithm can tolerate up to a *prespecified* number, \( k \), of faults.

- **Message Cost** — Our algorithm and the \( V \) algorithm are based on the LTD algorithm and the CV algorithm, respectively. In order to get a better view of the performance of the two algorithms, we decompose the overall message complexity of each algorithm into three components: base cost, preparation cost, and cost per actual fault. The base cost is the number of control messages required by each algorithm's non-fault-tolerant predecessor. The preparation cost is defined to be the number of control messages required by the algorithm in addition to the base cost, even if no process fails during the computation: it is the cost of preparing the algorithm for possible faulty processes. The cost per actual fault accounts for the number of control messages required by the algorithm in order to recover from an actual failure. Each process-to-process message is counted as one.

  - **Base cost** — As shown in [61], both LTD and CV algorithms are message-optimal in the worst case; but the former requires \( M + n - 1 \) messages only in the worst case, while the latter incurs a flat cost \( M + n - 1 \) in all cases.

  - **Preparation cost** — There is no preparation cost at all for our algorithm: if no process fails during the computation, our algorithm essentially becomes the LTD
algorithm. The V algorithm, by comparison, has a preparation cost of $O(kM)$: if both the V algorithm and its predecessor are applied to a system devoid of faulty processes, the V algorithm will use $O(kM)$ more control messages than its predecessor.

- **Cost per actual fault** — In our algorithm, for each fault that actually occurs during the computation, every nonfaulty process needs to send at most one extra SIGNAL message (to the system root). Overall, at most $n$ SIGNALs per fault are needed. (If the postponed start feature is included, then $n$ START messages should be added for each faulty process, since each new coordinator may send that many STARTs.) By comparison, each time a fault occurs, the V algorithm takes a snapshot and redirects each message to $k$ representatives. According to [103], about $O(M/n)$ messages need to be redirected to at most $k$ representatives, resulting in an overhead of $O(kM/n)$ per fault. If the user wants the algorithm to be $(n - 1)$-resilient, the quantity $kM/n$ is on the order of $M$, which is usually greater than $n$.

- **Storage** — Our algorithm uses $O(n)$ space for each process, or $O(n^2)$ in all. The V algorithm uses $O(kM)$, as each message transaction is stacked in $k$ duplicates. Note that $M$ is a quantity unknown at compile time and thus the V algorithm requires the more expensive dynamic storage allocation to maintain stacks.

- **Detection Delay** — The detection delay is defined to be "the length of the longest possible communication path along which control messages need to be sequentially passed after the basic computation has terminated but before the termination is
detected." The detection delay of our algorithm is $O(n)$. The worst-case detection delay of the $V$ algorithm is the same as that of the $CV$ algorithm, which, as pointed out in [61], could be as large as $M$.

- **Message Length** — The longest message used in our algorithm is $\text{SIGNAL}(c, S)$. The first parameter $c$ is an integer not exceeding $M$ and thus requires at most $\log M$ bits. The second parameter $S$ is a set of process identities that can be represented by $f$ integers, each of $\log n$ bits. The longest control message used in the $V$ algorithm is $\text{remove-entry}(p, q, t)$ (where $p$ and $q$ are process identities and $t$ is a time stamp), which needs $2\log n + \log M$ bits. For an extremely unreliable system in which more than two processes may fail during the computation, our algorithm has to use longer control messages than the $V$ algorithm uses.

- **Atomic sending** — Our algorithm uses no atomic sending. The $V$ algorithm requires that the underlying network be able to support $(k+1)$-message atomic sending, which allows a process to atomically send up to $k+1$ (possibly different) messages so as to ensure that each message transaction is correctly replicated at $k$ sites.

- **Network Services** — Aside from atomic sending, the network services required by both algorithms are comparable: reliable communication, fault-detecting, and either fault-flushing or return flush.

We have compared our algorithm with the $V$ algorithm and summarized their
Table 1: Performance Measures.

<table>
<thead>
<tr>
<th># Faults Tolerable</th>
<th>Overall Message Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td># Faults Tolerable</td>
<td>Base Cost</td>
</tr>
<tr>
<td>V Algorithm</td>
<td>$k (&lt; n)$</td>
</tr>
<tr>
<td>Ours</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Storage</th>
<th>Detection Delay</th>
<th>Message Length</th>
<th>Atomic Sending</th>
</tr>
</thead>
<tbody>
<tr>
<td>V Algorithm</td>
<td>$O(kM)$</td>
<td>$O(M)$</td>
<td>$2 \log n + \log M$</td>
</tr>
<tr>
<td>Ours</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$f \log n + \log M$</td>
</tr>
</tbody>
</table>

performance measures in Table 1. It is our view that a fault-tolerant algorithm should be able to tolerate any number of faults, should run as efficiently as the best non-fault-tolerant algorithm available if no process fails during the computation, and should incur only a reasonable amount of cost for each process failure that actually occurs. The above analyses indicate that our algorithm does have these nice properties.

5.7 Conclusion

We have proposed a fault-tolerant algorithm for termination detection that can tolerate any number of crash failures. This algorithm improves over existing ones in many aspects, including worst-case message complexity, storage overhead, and detection delay. Most important, it has two nice features: 1) No overhead is incurred for being fault-tolerant—if no process fails during the computation, the algorithm is comparable to the best existing non-fault-tolerant algorithm. 2) The cost for processing each fault occurrence is not high and is independent of both $M$, the number of
basic messages, and $k$, the maximum number of faulty processes the algorithm can
tolerate. These nice features, of course, do not come free: our algorithm requires that
every process knows the identifications of all processes in the basic system. We don’t
know whether it is possible to somehow relax this requirement without trading off
the above mentioned nice properties.

Our algorithm is based on the well-known Dijkstra-Scholten termination detector.
It achieves fault-tolerance not by message replication, but by appropriately setting
a barrier at the root of the system. Similar techniques might be applicable to other
termination detectors.
As described in Section 3.3 of Chapter III, the process status and the channel status play important roles when processes are subject to failure. In particular, we showed in the previous chapters: 1) that the fault-detecting service can provide to applications process status information sufficient to solve otherwise unsolvable problems such as the well-known consensus problem, and 2) that the fault-flushing service provides to the application information regarding channel status between a nonfaulty process and a faulty one sufficient to solve the termination detection problem.

In this chapter we discuss several other applications to demonstrate the usefulness of the proposed services in achieving fault-tolerance in distributed computing. First, the fail-detecting service is shown to make problems such as the process renaming problem easier to solve and to tremendously relax the original solvability constraint(s). The fail-flushing service is then shown to be critical to solving the backup process recovery problem in a totally asynchronous system.
6.1 Application of Service S1

Service S1 basically gives a process the capability of distinguishing a faulty process from a slow one, and thus helps in deciding when remedial procedures such as aborting or recovering actions should be taken. Problems that are unsolvable or difficult to solve in a totally asynchronous system due to lack of this capability can now be resolved by protocols that employ the S1 service to clear up the uncertainty about a (faulty) process status. The consensus problem [43] and the order preserving process renaming problem (OPR) [7] are two examples of such problems.

Though a single receiver now will not be blocked forever for an expected message coming from a faulty sender, there are two important issues that deserve some attention. First, the system as a whole does not have a global view about the set of faulty processes due to its asynchronous nature. For example, at time instant $t$, process $p_i$ may have known that $p_k$ had failed but process $p_j$ may have not. Second, a process may crash in the course of sending out a set of messages. As a result, some messages may not be sent out at all since sending out a set of messages is not an atomic action. Precautions have to be taken if an inconsistent view about such knowledge may contribute to an erroneous protocol state.

The problem nature of the order preserving process renaming problem (OPR) [7] is opposite to the consensus problem in the sense that a process has to decide on a value that upon which no one else will decide. However, they both require a
similar type of process coordination. The OPR problem is to find a renaming protocol such that the new names of all nonfaulty processes are uniquely chosen from a name space, \( \{1, 2, \ldots, N\} \), and their original naming order is preserved. Thus, if a process \( x \) chooses a new name \( \text{new}_x \), then for any nonfaulty processes \( i \) and \( j \), it is required that \( 1 \leq \text{new}_i \leq N \), \( 1 \leq \text{new}_j \leq N \) and \( \text{new}_i < \text{new}_j \) iff \( i < j \). It is assumed that each process initially has a unique name taken from an unbounded ordered domain.

When faults are not detectable, and we have \( n \) processes and need to be able to tolerate up to \( k \) faults, the size of the name space is shown to be \( N \geq 2^k(n - k + 1) - 1 \), where \( n \geq 2k \) [7]. The value of \( N \) is exponentially expanded relative to \( k \), and the set of new names may become far more sparse than the original name set. Though it is theoretically interesting to see that the OPR problem is solvable in a totally asynchronous system subject to failures, the characteristics required of a solution to this problem make it difficult to apply practically.

With service S1 provided, the simple protocol mentioned in [7] for the situation that decision is revocable, with modification, could be used to solve this problem with the size of name space \( N \) equal to the number of processes, \( n \), regardless of the number of failures that may occur. The protocol is summarized below. Initially, everyone has the smallest new identifier, say 1, and then announces its new proposed identity to every other process. Whenever a process \( i \) receives a proposed new identity \( \text{new}_j \geq \text{new}_i \) from \( j < i \), \( i \) updates its proposed new name to be \( \text{new}_j + 1 \) and then
otherwise, a receiver may block forever. (Note that another synchronization step may be necessary in order to allow every nonfaulty process to use new name consistently.)

6.2 Application of Service S2

Though service S1 is sufficient for problems such as consensus and OPR to be solvable in $S^k$, $0 \leq k \leq n$, by itself it is not sufficient for some other problems such as termination detection [38], or backup process recovery Problem [56] to be solvable. The main reason is due to the uncertainty incurred by the in-transit messages coming from a faulty process. In other words, a receiver has no idea when all these messages will arrive, though it may have been notified the failure of the sender. As mentioned earlier, these in-transit messages may carry not only consistent, but also productive results that are essential to the computation. Simply to discard these messages is just a waste of system resources. Service S2 thus becomes important to problems which require knowing that a channel from a faulty process is free of messages.

The utilization of service S2 for the termination detection problem has been explained in Chapter V. In the following, we show that the fault-flushing service is also essential to the backup process recovery problem (or fault-tolerant processes problem). As the fault-detecting service alone is not sufficient for the algorithm of fault-tolerant processes in [56] to correctly recover due to the unpredictable message delay of an asynchronous system, the fail-flushing service becomes imperative for the
algorithm in order for the backup process to obtain the most up-to-date knowledge of a faulty process.

We summarized the fault-tolerant processes approach [56] below. The model assumed is FIFO; that is, messages will be received in the order they were sent. Based on a message-logging scheme, a primary process will send a checkpoint to its backup process whenever it intends to send out a message to others. The checkpoint contains a list of messages received since the last checkpoint. The backup process maintains all the checkpoints received in order, so it knows what sequence of messages the primary process has received and what actions it has to simulate in order to make sure it can restart from a consistent position at which the primary fails.

When a primary process fails, its backup process at the other site will restart by recovering from all the checkpoints sent by the primary process. Let's take a look at an example of this scheme which will help us to understand why the fail-flushing service is critical to this algorithm.

In Figure 9, a primary process becomes faulty after sending out two message checkpoints. However, after its backup process receives the first message checkpoint, the backup process detects the failure. The backup process does not know how many message checkpoints are on the way. But as message delay is unpredictable, a backup process cannot wait forever. So, it decides to take recovery action. Unfortunately, there is a message checkpoint on the way that is not known to the backup process.
makes the name known to the public.

It is obvious that this protocol will use at most $n$ names. To make this protocol useful, however, an enhancement feature on how to finalize a decision should be carefully deliberated. In other words, the difficult part of making the above protocol work is letting each process recognize when the proposed name is stabilized and can be used in practice.

The above protocol is modified based on the concept of stable vector proposed in [7] as follows. Instead of just carrying its own proposed name, it carries all proposed names known to it. Let each process allocate locations for a set of $n$ elements. Each element consists of a pair of information for one process: the original name and the proposed name. Initially, each process sets its own entry with the proposed name to be the smallest name allowed, and announces the set to all other processes. Whenever a process receives a set with an element that has larger proposed name, it updates its own set. In particular, if an element has a smaller original name but a larger proposed name, then the process updates its proposed name to be the proposed name of the element plus one. After updating its own set, the process announces the new set to others again. At the time when it receives information from every nonfaulty process known to it in which the proposed name that the other process knew is the same as the one it currently proposes, the name it currently proposes for itself becomes its new process identity. The above step is possible only when the $S1$ service is facilitated;
Clearly, the backup process does not restart from the place where the primary has announced to others (the last send contains the state that the primary process was in). Hence, the recovery action definitely violates system consistency due to a later arrival of a message checkpoint. Though the availability of service S2 is not explicitly assumed in [56], from the counter example, it is not hard to see that service S2 is vital to this problem if system consistency is to be maintained.

6.3 Conclusion

Two other applications were discussed to demonstrate the usefulness of the proposed services in achieving fault-tolerance in distributed computing. The fail-detecting service is shown to make problems such as the process renaming problem easier to solve and to tremendously relax the original solvability constraint(s), and the fail-flushing service is shown to be critical to solving the backup process recovery problem in a totally asynchronous system.
CHAPTER VII
Conclusions and Future Work

In this dissertation, we have identified and proposed two system services, fault-detecting and fault-flushing, to be provided in distributed computing environments to facilitate the design of fault-tolerant distributed applications. We showed that the ability to detect faulty processes plays an important role in both consensus and termination detection problems. This feature is sufficient for the consensus problem to be solvable. While fault detectability is necessary for termination detection to be 1-solvable, it alone is not sufficient. Augmented with the fault-flushing service, the termination detection problem is shown to be solvable.

The fault-tolerant termination detection algorithm constructed based upon the proposed services improves over existing algorithms in many aspects, including worst-case message complexity, storage overhead, and detection delay. Most important, it has two nice features: 1) No overhead is incurred for being fault-tolerant—if no process fails during the computation, the algorithm is comparable to the best existing non-fault-tolerant algorithm. 2) The cost for processing each fault occurrence is not high and is independent of both $M$, the number of basic messages, and $k$, the maximum
number of faulty processes the algorithm can tolerate. These nice features, of course, do not come free: our algorithm requires that every process knows the identifications of all processes in the basic system. It is uncertain to us whether it is possible to somehow relax this requirement without trading off the above mentioned nice properties.

We also discussed why it is impossible to realize these services as distributed applications running above the network. They should be achieved by the underlying network, where the synchronous nature presented by the lower layers or extra hardware facility are made available.

It has been suggested in the literature that, in a totally asynchronous system, as consensus is impossible in the presence of faulty processes, so is termination detection. We confirmed this belief by showing that termination detection is indeed harder than consensus.

In addition to the consensus problem and the termination detection problem, the usage of these proposed services are also demonstrated by several other applications that are important in distributed computing. The fail-detecting service is shown to make problems such as the process renaming problem easier to solve and to tremendously relax the original solvability constraint(s). The fail-flushing service is shown to be critical to solving the backup process recovery problem in a totally asynchronous system.
In summary, this thesis research focuses on the study of problems concerning fault-tolerant distributed systems. In particular, the followings are accomplished:

1. Two system services, fault-detecting and fault-flushing, are identified and proposed to facilitate the design of fault-tolerant distributed algorithms.

2. The fault-detecting service is shown to be sufficient for solving the consensus problem in a totally asynchronous application environment.

3. The fault-detecting service alone is shown to be insufficient for solving the termination detection problem in a totally asynchronous application environment.

4. The problem of consensus is reduced to the problem of termination detection. Thus, the termination detection problem is indeed shown to be harder than the consensus problem.

5. A fault-tolerant termination detection algorithm is constructed. It will tolerate any number of crash failures, run as efficiently as its non-fault-tolerant predecessor if no process actually fails during the computation, and otherwise incur only a small amount of cost for each actual failure.

6. The fail-detecting service is shown to make problems such as the process renaming problem easier to solve and to tremendously relax the original solvability constraint(s). The fail-flushing service is then shown to be critical to solving the backup process recovery problem in a totally asynchronous system.
7.1 Future Work

Though the two proposed services are shown to be useful for many distributed applications, we are not sure whether they are sufficient for deriving an asynchronous problem solution from any and every synchronous solution to a distributed system problem. Intuitively it seems that they are sufficient, but we do not have any proof support.

Our fault-tolerant termination detection algorithm is based on the well-known Dijkstra-Scholten termination detector. It achieves fault-tolerance not by message replication, but by appropriately setting a barrier at the root of the system. Hopefully, similar techniques will be applicable to other termination detectors.

The proposed services are shown to be sufficient for making the termination detection problem fault-tolerant. It is also interesting to see whether the proposed services are necessary for solving it.

Recently, a stronger version of the consensus problem, called agreement consensus, has been shown to be harder than the regular consensus problem [31]. It would be interesting to investigate this problem's relationship with termination detection, as well as its relationship with services S1 and S2.

The implementation of the proposed services is only sketched in the dissertation. It deserves more attention and discussion to come up with an efficient and reliable realization.
BIBLIOGRAPHY


