APPROXIMATING PERFECT FAILURE DETECTORS IN ASYNCHRONOUS DISTRIBUTED SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Laura Susanne Sabel

January 1996

APPROXIMATING PERFECT FAILURE DETECTORS IN ASYNCHRONOUS DISTRIBUTED SYSTEMS

Laura Susanne Sabel, Ph.D.

Cornell University 1996

Systems that provide group membership services (e.g., Isis) can be used to solve many canonical problems in distributed systems, such as Consensus and Leader Election. However, solving these problems has been shown to require *failure detectors* of various strengths, all of which are impossible to implement in an asynchronous system. It has been shown that it is possible to solve Consensus in an asynchronous distributed system using a very weak failure detector. However, existing group membership systems do not appear to implement this type of failure detector. Furthermore, there are other problems that can be solved with group membership services, such as Leader Election, that are harder than Consensus and cannot be solved without a perfect failure detector. This leads to an apparent contradiction: how can provably impossible problems be solved in existing group membership systems?

In this dissertation, we investigate *approximations* to perfect failure detectors in order to obtain acceptable and practical solutions to otherwise-impossible problems such as Election. We argue that approximating a perfect failure detector yields approximate solutions to these problems, and define a notion of approximation that yields solutions that are useful in practice. We give a formal specification of an approximately-perfect failure detector, give two protocols that implement our approximation, and derive upper bounds on the number of tolerable failures for any such protocol. The approximately-perfect failure detector presented in this dissertation is similar to that used in Isis; this implies that the failure detector implemented in group membership services such as Isis are actually approximately-perfect failure detectors, and that the resulting solutions to Consensus, Election, and other problems are approximate solutions. Furthermore, the protocols that implement our specification are very simple and can be easily implemented. Such an implementation could be used in existing or future group membership services, or by other systems programmers for whom group membership services are not necessary but for whom failure detection would be useful.

This work was supported by an AT&T PhD Scholarship, and by grants from the Defense Advanced Research Projects Agency (DoD), IBM, and Siemens.

Biographical Sketch

The author received the BSE degree in Computer Science from Princeton University in 1989 and the MA degree from Cornell University in 1992.

Acknowledgements

The contents of this dissertation are based on joint work with my advisor, Professor Keith Marzullo. Over the last six years, he has given me ideas, wisdom, support, plants, the use of his home and his car, and lots of fine cooking. Words cannot express my gratitude, but here's a shot: Keith, thank you.

Rick Aaron lent his combinatorics expertise to the proof of Theorem 18. He also provided vast amounts of emotional support, and the occasional shove in the right direction, over the last three years. I couldn't have done it without him.

Several other people have contributed to this dissertation, in the form of ideas and feedback. Thanks to Ken Birman, Fred Schneider, and Sam Toueg.

I would also like to thank AT&T for three years of generous funding, the Department of Computer Science for lending me their virtually infinite resources for six years, and my family for accepting with grace and patience my vague descriptions of what I've been doing up here all this time.

Table of Contents

List of Figures

Chapter 1

Introduction

An asynchronous distributed system is a loosely-coupled collection of processors "cooperating" to solve a problem or execute a task, without sharing memory or clocks and with arbitrary message delivery times. Processes in any real system are subject to different types of failures, the simplest of which is a *crash* failure: when a process crashes, it simply stops executing instructions. Even when failures are restricted to crash failures, due to the asynchrony of message delivery it is not possible for a process to be certain that another process has failed. This makes cooperation among processes difficult. For example, it was shown in 1985 by Fischer, Lynch, and Paterson ([FLP85]) that the Consensus problem, in which a set of processes must agree on the same value of a variable, cannot be solved deterministically in an asynchronous distributed system if even one process can crash because of the inherent difficulty of distinguishing a crashed process from a slow process.

Many systems that provide *group membership* services have been developed (e.g., [BJ87,ADKM92,MPS91a]). These services allow processes to form groups and to maintain information about the membership of the groups. Such systems allow processes to cooperate more easily by providing abstractions, such message ordering and failure notification, that hide some of the asynchrony of the underlying system. Group membership services provide many useful properties to the application programmer. For example, such systems allow group members to agree on the order in which group members leave or join the group, and on the order in which messages are sent within the group. Group membership services can be used to solve many canonical problems in distributed systems, such as Consensus, Atomic Broadcast, and Leader Election. However, solving these problems has been shown to require *failure detectors* of various strengths ([CHT92,SM95]), all of which are impossible to implement in an asynchronous system. This leads to an apparent contradiction: how can provably impossible problems be solved in existing group membership systems? Unfortunately, it is not clear how group membership systems detect failures in order to solve these problems. Until recently the failure detection mechanisms of such systems have not been examined formally. The formal treatments that do exist focus on the ability of the failure detection mechanism to support group membership services, rather than the type of failure detector needed to solve problems such as Consensus and Election. Furthermore, there is no consensus as yet on the precise definition of "group membership"[1] ([ACBMT95]), and so examinations of the type of failure detector

[1] It would seem that achieving consensus among distributed systems researchers is also impossible.

needed to support group membership are by necessity applicable only to specific systems, and not to group membership as a general problem. One of the contributions of this dissertation is to resolve the above contradiction: we provide a formal specification of an implementable failure detector that allows approximate solutions to otherwise-impossible problems, and that can be used in existing group membership services.

The Consensus problem has been examined extensively in the literature. As stated above, it was shown in 1985 ([FLP85]) that Consensus cannot be solved deterministically in an asynchronous distributed system if even one process can crash. Since then, much work has been done on methods of circumventing the impossibility result, such as adding synchrony (e.g., bounded message delay or synchronized clocks) ([Cri88,DDS87]) or nondeterminism ([CD89]). However, it is difficult to ensure that a system always behaves synchronously in a distributed system; most systems exhibit asynchronous behavior at least some of the time due to varying workloads, rendering synchrony assumptions invalid.

In 1991 ([CT91]) it was shown that it is possible to solve Consensus using a failure detector. This work showed that Consensus can be solved using a very weak failure detector that makes an arbitrary number of mistakes. Implementing this failure detector requires some synchrony assumptions; however, the assumptions are likely to be met in many systems. Nevertheless, though a weak failure detector can be used to solve Consensus, there are very practical problems (e.g., Election) that are harder than Consensus and cannot be solved without a perfect failure detector. However, one can implement a failure detector that *approximates* a perfect failure detector; given such an approximation, one can achieve approximate solutions to problems such as Election. Clearly, it is desirable to define an approximation to a perfect failure detector that is "good enough" to yield useful solutions. Other contributions of this dissertation are to provide a definition of "approximate" that can be used in practice, and to give a specification of a failure detector that approximates a perfect failure detector.

We show in this dissertation that there are problems that are harder than Consensus, in that they cannot be solved with an imperfect failure detector: they require a perfect failure detector. We investigate approximations to perfect failure detectors in order to obtain acceptable and practical solutions to these problems. We argue that approximating a perfect failure detector yields approximate solutions to these problems, and define a notion of approximation that yields solutions that are useful in practice. We give a formal specification of an approximately-perfect failure detector, give two protocols that implement our approximation, and derive upper bounds on the number of tolerable failures for any such protocol. The approximately-perfect failure detector presented in this dissertation is similar to that used in Isis ([BJ87,RB91]); this implies that the failure detector implemented in group membership services such as Isis are actually approximately-perfect failure detectors, and that the resulting solutions to Consensus, Election, and other problems are approximate solutions. Furthermore, the protocols that implement our specification are very simple and can be easily implemented. Such an implementation could be used in existing or future group membership services, or by other systems programmers for whom group membership services are not necessary but for whom failure detection would be useful.

The dissertation is organized as follows. Chapter 2 describes the system model used throughout the dissertation, including notation and definitions. Chapter 3 discusses the Consensus and Election problems, and compares them in terms of the weakest failure detector that can be used to solve them. We show that Election is strictly harder than Consensus, and in fact requires a perfect failure detector. We discuss the Group Membership problem and the failure detectors that are commonly used to solve it, and outline other problems that require failure detectors. Chapter 4 reviews the well-known *fail-stop* failure model, which specifies a perfect failure detector. We show that this model is impossible to implement in an asynchronous system. We formally define *indistinguishability* of failure models, which is the notion of approximation that is used for the rest of the dissertation, and give three necessary conditions for any failure model to be indistinguishable from fail-stop. We give a specification of such a model, called *quasi fail-stop*, and prove that it is indistinguishable from fail-stop. We give two protocols that implement the quasi fail-stop model. We give bounds on fault tolerance for the implementation of any model indistinguishable from fail-stop. Chapter 5 examines stronger versions of the fail-stop model and briefly discusses whether these models can also be approximated. Chapter 6 discusses other research on failure detectors.

Chapter 2

Asynchronous Distributed Systems

We consider a distributed system consisting of a set of n processes $P = \{1, 2, \ldots, n\}$. A process fails by simply stopping execution (*crashing*), and a failed process does not recover. The system is asynchronous, meaning that the rate of execution of any process with respect to any other is unbounded and there are no physical clocks. Between any two processes i and j there exist two unidirectional FIFO channels: $C_{i,j}$ from i to j and $C_{j,i}$ from j to i.[1] Processes communicate only by sending and receiving messages over these channels. The channels are nonfaulty: they do not lose, generate, or garble messages. Message delivery time is unbounded. We assume for simplicity that channels have infinite buffers and that all messages m are unique (they can easily be made so by including in m its source and a sequence number). The state of a channel is the sequence of messages that have been sent

[1] In Chapter 4 we will redefine the order in which messages are delivered on these channels.

along the channel but not received along the channel.

A process is defined by a set of states, one of which is denoted the initial state. The state of a process i consists of the values of all internal variables of the process, plus the values of $2n + 1$ additional boolean variables that are defined as follows.

- $\mathbf{crash}_i$. This variable is initially false and can become true at any time. Once $\mathbf{crash}_i$ becomes true, the state of i does not change further. (This models the failure of i.)

- $\forall j \in P :$ $\mathbf{failed}_i(j)$. This variable is initially false for all values of j, and becomes true when i detects the crash of process j. Once $\mathbf{failed}_i(j)$ becomes true, it remains true forever.

- $\forall j \in P :$ $\mathbf{suspect}_i(j)$. This variable is initially false for all values of j, and becomes true when i *suspects* the crash of process j (e.g., due to a timeout at a lower level). We assume that a suspicion of a failure always precedes the detection of that failure, and that $\mathbf{suspect}_i(i)$ is always false. We assume unless stated otherwise that once $\mathbf{suspect}_i(j)$ becomes true, it remains true forever.

A *global state* of the system is a set of process and channel states. An *initial global state* is the global state in which each process state is an initial state and each channel state is the empty sequence.

An *event* e is an action that changes the global state of the system from Σ to Σ' such that Σ' differs from Σ in the local state of exactly one process i and the state of at most one channel incident on i. We say in this case that e is an event *of* i, and that e *changes* the state of i and/or the state of some channel incident on i. If an event e of process i changes the state of $C_{i,j}$ for some j, then we call e a *send event.* A send event changes the state of a channel by appending a message m to the sequence of messages on that channel. If event e of i changes the state of $C_{j,i}$ for some j, then we call e a *receive event.* A receive event changes the state of a channel by removing a message from the head of the sequence of messages on that channel.

An event e is uniquely determined by the process i whose state it changes, the state s of i in the global state in which e occurs, the state s' of i resulting from e, the states of the channels incident on i before e occurs, and the states of the channels incident on i after e occurs. Let $\mathcal{X}_{i,j}$ be the state of channel $C_{i,j}$ before e occurs. Let $\mathcal{X}_{i,*}$ be the n-tuple $\langle \mathcal{X}_{i,1}, \mathcal{X}_{i,2}, \ldots, \mathcal{X}_{i,n} \rangle$ and let $\mathcal{X}_{*,i}$ be the n-tuple $\langle \mathcal{X}_{1,i}, \mathcal{X}_{2,i}, \ldots, \mathcal{X}_{n,i} \rangle$. Similarly, let $\mathcal{X}'_{i,j}$ be the state of channel $C_{i,j}$ after e occurs, and let $\mathcal{X}'_{*,i}$ be the n-tuple $\langle \mathcal{X}'_{1,i}, \mathcal{X}'_{2,i}, \ldots, \mathcal{X}'_{n,i} \rangle$. Then, e is defined by the 7-tuple $\langle i, s, s', \mathcal{X}_{i,*}, \mathcal{X}'_{i,*}, \mathcal{X}_{*,i}, \mathcal{X}'_{*,i} \rangle$, such that:

- if $\mathcal{X}_{i,*} \neq \mathcal{X}'_{i,*}$ (e is a send event), then $\mathcal{X}_{*,i} = \mathcal{X}'_{*,i}$, there exists exactly one $j \neq i$ such that $\mathcal{X}_{i,j} \neq \mathcal{X}'_{i,j}$, and $\mathcal{X}'_{i,j} = (\mathcal{X}_{i,j} :: m)$ for some message m (where :: is the catenation operator).

- if $\mathcal{X}_{*,i} \neq \mathcal{X}'_{*,i}$ (e is a receive event), then $\mathcal{X}_{i,*} = \mathcal{X}'_{i,*}$, there exists exactly one $j \neq i$ such that $\mathcal{X}_{j,i} \neq \mathcal{X}'_{j,i}$, and $(m :: \mathcal{X}'_{j,i}) = \mathcal{X}_{j,i}$ for some message m.

Definition 1 *We say that* $e = \langle i, s, s', \mathcal{X}_{i,*}, \mathcal{X}'_{i,*}, \mathcal{X}_{*,i}, \mathcal{X}'_{*,i} \rangle$ can occur in *global state* Σ *if and only if:*

- *the state of process i in Σ is s,*

- *the states of the incoming channels incident on i in Σ are $\mathcal{X}_{i,*}$, and*

- *the states of the outgoing channels incident on i in Σ are $\mathcal{X}_{*,i}$.*

We will use the functional notation $e(\Sigma) = \Sigma'$ to denote that e occurs in global state Σ and results in global state Σ'.

Send, receive, crash, failure detection, and suspicion events are defined as follows. Let $e = \langle i, s, s', \mathcal{X}_{i,*}, \mathcal{X}'_{i,*}, \mathcal{X}_{*,i}, \mathcal{X}'_{*,i} \rangle$.

- If e is a send event of i and $\mathcal{X}'_{i,j} = (\mathcal{X}_{i,j} :: m)$ for some j, then e is denoted $\mathit{send}_i(j, m)$; that is, $\mathit{send}_i(j, m)$ denotes the event whereby process i sends the message m to process j.

- If e is a receive event of i and $(m :: \mathcal{X}'_{j,i}) = \mathcal{X}_{j,i}$ for some j, then e is denoted $\mathit{recv}_i(j, m)$; that is, $\mathit{recv}_i(j, m)$ denotes the event whereby process i receives the message m from process j.

- If $\mathbf{crash}_i$ is false in s and true in s', then e is denoted crash_i; that is, crash_i denotes the event whereby $\mathbf{crash}_i$ becomes true.

- If $\exists j:\ \mathbf{failed}_i(j)$ is false in s and true in s', then e is denoted $\mathit{failed}_i(j)$; that is, $\mathit{failed}_i(j)$ denotes the event whereby $\mathbf{failed}_i(j)$ becomes true.

- If $\exists j:\ \mathbf{suspect}_i(j)$ is false in s and true in s', then e is denoted $\mathit{suspect}_i(j)$; that is, $\mathit{suspect}_i(j)$ denotes the event whereby $\mathbf{suspect}_i(j)$ becomes true.

The events $\mathit{send}_i(j, m)$, $\mathit{recv}_i(j, m)$, crash_i, $\mathit{failed}_i(j)$, and $\mathit{suspect}_i(j)$ are atomic: at most one of the boolean variables **crash**, **failed**, and **suspect** or at most one channel state is changed by each event. For example, if $\mathbf{crash}_i$ is false in local state s of i when $send_i(j,m)$ occurs, then $\mathbf{crash}_i$ is false in the resulting state of i.

Definition 2 *A* run *is an infinite sequence of global states:* $r = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$, *where* Σ_0 *is an initial global state and there exists a sequence of events* $(e_0, e_1, e_2, \ldots)$ *such that for all* $i \geq 0$, $e_i(\Sigma_i) = \Sigma_{i+1}$. *For any positive integer* k, (r,k) *denotes* Σ_k *(the* $(k+1)^{st}$ *global state of* r*).*

Definition 3 *Given any run* $r = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$, *the* history *of* r, *denoted* $\mathcal{H}_r$, *is the sequence of events* $(e_0, e_1, e_2, \ldots)$ *such that for all* $i \geq 0$, $e_i(\Sigma_i) = \Sigma_{i+1}$.

Note that for any run r, $\mathcal{H}_r$ is uniquely determined. Furthermore, r can be constructed from a history $\mathcal{H}_r$ and the initial global state Σ_0.

Throughout this dissertation, we use the notation $\mathcal{H}_r = (\cdots e_i \cdots e_j \cdots e_k \cdots)$. This denotes that $\mathcal{H}_r$ is of the form $(x; e_i; y; e_j; z; e_k; w)$, where e_i, e_j, and e_k are events, x, y, and z are finite sequences of events, and w is an infinite sequence of events.

We specify properties of systems using predicate logic over global states and linear-time temporal logic over (infinite) suffixes of runs ([Pnu77]). We define the boolean predicates $\textsc{send}_i(j,m)$ and $\textsc{recv}_i(j,m)$ as follows.

- $\forall i,j,m$: $\textsc{send}_i(j,m)$ and $\textsc{recv}_i(j,m)$ are false in an initial global state.
- Let $e = send_i(j,m)$ and let Σ be a global state such that e can occur in Σ. Then $send_i(j,m)(\Sigma) \models \textsc{send}_i(j,m)$. That is, $\textsc{send}_i(j,m)$ is true in the global state resulting from $send_i(j,m)$ and in every global state thereafter.

- Let $e = recv_i(j,m)$ and let Σ be a global state such that e can occur in Σ. Then $recv_i(j,m)(\Sigma) \models \text{RECV}_i(j,m)$. That is, $\text{RECV}_i(j,m)$ is true in the global state resulting from $recv_i(j,m)$ and in every global state thereafter.

Both $\text{SEND}_i(j,m)$ and $\text{RECV}_i(j,m)$ are *stable* by definition ([CL85]): once such a predicate becomes true in a run, it remains true for the remainder of the run.

We define the boolean predicates CRASH_i, $\text{FAILED}_i(j)$, and $\text{SUSPECT}_i(j)$ as follows. Let Σ be a global state.

- $\forall i\colon\ \Sigma \models \text{CRASH}_i$ if and only if $\mathbf{crash}_i$ is true in Σ.

- $\forall i,j\colon\ \Sigma \models \text{FAILED}_i(j)$ if and only if $\mathbf{failed}_i(j)$ is true in Σ.

- $\forall i,j\colon\ \Sigma \models \text{SUSPECT}_i(j)$ if and only if $\mathbf{suspect}_i(j)$ is true in Σ.

Note that both CRASH_i and $\text{FAILED}_i(j)$ are stable by assumption: once the corresponding local variables become true in the local state of i, they remain true thereafter. $\text{SUSPECT}_i(j)$ is assumed to be stable unless stated otherwise. Also note that by assumption, $\forall i,j\colon\ (\Sigma \models \text{FAILED}_i(j)) \Rightarrow (\Sigma \models \text{SUSPECT}_i(j))$.

Definition 4 *Let $s = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$ be a suffix of a run,*[2] *let ϕ be a predicate, let k be a non-negative integer, and let $\mathcal{P}$ be a temporal logic formula.*

- $(s,k) \models \phi$ *if and only if* $\Sigma_k \models \phi$

- $(s,k) \models \Diamond\mathcal{P}$ *if and only if* $\exists k' \geq k\colon\ (s,k') \models \mathcal{P}$

- $(s,k) \models \Box\mathcal{P}$ *if and only if* $\forall k' \geq k\colon\ (s,k') \models \mathcal{P}$

[2] Note that Σ_0 need not be an initial global state in s.

Furthermore, we abbreviate $(s,0) \models \mathcal{P}$ *as* $s \models \mathcal{P}$.

We define the *failed-before* relation as follows.

Definition 5 *If* $r \models \Diamond\text{FAILED}_j(i)$ *for some run* r, *we say that* i failed before j *in* r.

Note that it is possible that both CRASH_i and CRASH_j hold in some global state of r yet neither i failed before j nor j failed before i in r. It is also possible that both i failed before j and j failed before i in some run, if failures can be detected erroneously.

We use a version of the *happens before* relation of [Lam78]. Given two events e_1 and e_2, define $e_1 \rightarrow e_2$ (read "e_1 happens before e_2") in some history $\mathcal{H}_r$ if one of the three following conditions holds:

1. e_1 and e_2 are of the same process, and either $e_1 = e_2$ or e_1 precedes e_2 in $\mathcal{H}_r$;

2. $e_1 = send_i(j,m)$ for some values of i, j, and m, and $e_2 = recv_j(i,m)$;

3. there exists an event e such that $e_1 \rightarrow e$ and $e \rightarrow e_2$ in $\mathcal{H}_r$.

The happens-before relation as defined here is the same as that given in [Lam78], except that our relation is reflexive for notational convenience. Note that for all $e_1 \neq e_2$, $e_1 \rightarrow e_2$ implies that e_1 precedes e_2 in $\mathcal{H}_r$. The converse does not hold, however.

Definition 6 *Let* r_i *be the sequence of states of process* i *in run* r, *with repeated states removed (i.e., so that adjacent states are distinct). If* x *and* y *are runs, then* *we say that run x is* isomorphic to *run y with respect to process i, denoted $x \equiv_i y$, if and only if $x_i = y_i$. In other words, $x \equiv_i y$ if and only if runs x and y are indistinguishable to process i. Similarly, let r_Q for $Q \subseteq P$ be the sequence of states of processes $i \in Q$ in r with repeated states removed. We say x is isomorphic to y with respect to Q, denoted $x \equiv_Q y$, if and only if $x_Q = y_Q$. (See [CM86] for a detailed discussion of the ramifications of indistinguishability of runs.)*

In Chapter 3, we use the following characterization of knowledge introduced in [CM86].

Definition 7 *A process i* knows *a predicate Φ in (r,k), written $(r,k) \models K_i\Phi$, if, for all prefixes (r',k') indistinguishable by i from (r,k), $(r',k') \models \Phi$.*

Definition 8 *A run r has a* process chain *$\langle i_1, i_2, \ldots i_m\rangle$ if there exist events $e_1, e_2, \ldots, e_m$ in r such that e_j is an event of process i_j and $e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_m$ in $\mathcal{H}_r$.*

Theorem 1 *If $(r,k) \models K_{i_1}K_{i_2}\ldots K_{i_m}\Phi$ and for $k' > k, (r,k') \models \neg K_{i_m}\Phi$ then there is a process chain $\langle i_1, i_2, \ldots i_m\rangle$ in r between k and k'.*

This theorem indicates that knowledge cannot be gained or lost without the exchange of messages, and is proven in [CM86].[3]

A *failure model* describes the manner in which the components of a system can fail. For our purposes, a failure model constrains how *crash* events and *failed*

[3] When a process detects a failure, it may gain knowledge about the state of the failed process. However, this knowledge is not explicitly gained by passing messages. Therefore, in runs in which failure detections occur, Theorem 1 may not hold.

events can occur with respect to each other. We give these constraints as a set of properties and define the failure model as the set of runs that satisfy these properties:

Definition 9 *A* failure model *is the set of runs that satisfy a given collection of properties relating* crash *and* failed *events.*

A failure model can also be described as a set of runs in which each process uses a *failure detector* to detect *crash* events, where the failure detector of each process obeys the constraints specified. Thus, a failure model can be seen as a specification of a failure detector.

We will refer to the failure detector hierarchy defined by Chandra and Toueg in [CT91]. The failure detectors in this hierarchy are specified using a range of assumptions about the *completeness* and *accuracy* of failure detections.[4] Note that Chandra and Toueg do not assume that failure detections are stable. These assumptions are:

Strong Completeness: $\forall r\colon\ r \models \forall i\colon\ \Box(\textsc{crash}_i \Rightarrow \forall j\colon\ \Diamond(\textsc{crash}_j \vee \Box\textsc{failed}_j(i)))$

(Eventually the failure of every process that crashes is permanently detected by every correct process).

Weak Completeness: $\forall r :\quad r \models (\forall i :\quad \Diamond\textsc{crash}_i) \vee (\forall i :\quad \Box[\textsc{crash}_i \Rightarrow \exists j : (\Box\neg\textsc{crash}_j \wedge \Diamond\Box\textsc{failed}_j(i))])$ (Either there is no correct process, or eventually the failure of every process that crashes is permanently detected by some correct process).

[4] The failure *suspicions* of [CT91] are equivalent to our failure *detections*.

Strong Accuracy: $\forall r\colon\ r \models \forall i,j\colon\ \Box(\text{FAILED}_i(j) \Rightarrow \text{CRASH}_j)$ (No process' failure is detected before that process crashes).

Weak Accuracy: $\forall r\colon\ r \models (\forall i\colon\ \Diamond\text{CRASH}_i) \vee \Box(\exists i\colon\ (\neg\text{CRASH}_i \wedge \forall j\colon\ \Box\neg\text{FAILED}_j(i)))$ (Either there is no correct process, or there is some correct process whose failure is never detected).

Eventual Strong Accuracy $\forall r\colon\ r \models \Diamond\forall i,j\colon\ \Box(\text{FAILED}_i(j) \Rightarrow \text{CRASH}_j)$ (There is a time after which there is *no* correct process whose failure is detected by any correct process).

Eventual Weak Accuracy $\forall r\colon\ r \models \Diamond[\Box(\exists i\colon\ (\neg\text{CRASH}_i \wedge \forall j\colon\ (\Box\neg\text{FAILED}_j(i))))]$ (There is a time after which the failure of *some* correct process is never detected by any correct process).

Chandra and Toueg define several failure detectors using these properties.

- A *Perfect Failure Detector* ($\mathcal{P}$) is a failure detector that generates runs satisfying both Strong Completeness and Strong Accuracy;
- A *Strong Failure Detector* ($\mathcal{S}$) generates runs that satisfy both Strong Completeness and Weak Accuracy;
- An *Eventually Strong Failure Detector* ($\Diamond\mathcal{S}$) generates runs that satisfy both Strong Completeness and Eventual Weak Accuracy;
- An *Eventually Weak Failure Detector* ($\Diamond\mathcal{W}$) generates runs that satisfy both Weak Completeness and Eventual Weak Accuracy.

In Section 3.2.3, we define the following additional property:

Very Weak Completeness: $\forall r: \ r \models (\forall i: \ \Diamond \text{CRASH}_i) \vee (\forall i: \ \Box[\text{CRASH}_i \Rightarrow \exists j: (\Box\neg\text{CRASH}_j \wedge \Diamond \text{FAILED}_j(i))])$ (Either there is no correct process, or eventually the failure of every process that crashes is detected by some correct process).

This property is equivalent to Weak Completeness if failure detections are stable.

Chapter 3

Problems Requiring Failure Detectors

3.1 Background

It was shown in 1985 that the Consensus problem cannot be solved deterministically in an asynchronous system if even a single crash failure can occur ([FLP85]). The intuition behind this widely-cited result is that in an asynchronous system, it is impossible for a process to distinguish between another process that has crashed and one that is merely very slow. The consequences of this result have been enormous, because most real distributed systems today can be characterized as asynchronous, and Consensus is an important problem to be solved if the system is to tolerate failures. As a result, Consensus has frequently been used as a yardstick of computability in asynchronous fault-tolerant distributed systems.

In this chapter, we show that there are other problems that cannot be solved in an asynchronous system, and for the same intuitive reason: it is impossible to distinguish a very slow processor from a crashed processor. However, these problems are harder than Consensus, in that there are contexts in which Consensus can be solved but these other problems cannot. More precisely, the weakest failure detector that is needed to solve these problems is a perfect failure detector, which is strictly stronger than the weakest failure detector that is needed to solve Consensus.

We use a formulation of the Election problem as the prototype for problems that are harder than Consensus. We review the Consensus problem and the weakest failure detector needed to solve Consensus. We then define the Election problem and give a proof that the weakest failure detector needed to solve Election is stronger than the weakest failure detector needed to solve Consensus. Finally, we discuss the Group Membership problem, and other problems that, like Election, are harder than Consensus.

3.2 Consensus Vs. Election

3.2.1 Consensus

The Consensus problem requires that all correct processes propose a value, and then reach an agreement on one of the proposed values. In [FLP85], it was shown that Consensus cannot be solved in an asynchronous system in which a single process may fail. The form of Consensus for which this result was shown is stated as follows.

Every process starts with an initial value in $\{0, 1\}$. A nonfaulty process decides on a value in $\{0, 1\}$ by entering an appropriate decision state. All nonfaulty processes that make a decision are required to choose the same value. Some process must eventually make a decision. The trivial solution in which a particular value is always chosen is ruled out by stipulating that both 0 and 1 are possible decision values.[1]

The proof of the impossibility of Consensus in [FLP85] assumes that it is impossible for a process to determine whether another process has crashed, or is just very slow. It is shown in [CT91] that Consensus can be solved using a *strong* failure detector ($\mathcal{S}$), which is the failure detector that satisfies Strong Completeness and Weak Accuracy. It is shown in [CHT92] that the weakest failure detector that can be used to solve Consensus is $\diamond\mathcal{W}$, which is the failure detector that satisfies Weak Completeness and Eventual Weak Accuracy. This result applies to problems that are equivalent to Consensus as well, such as the *Atomic Broadcast* problem ([Mul93,Cha93]).

3.2.2 Election

The Election problem is described as follows: At any time, at most one process considers itself the *leader*, and at any time, if there is no leader, a leader is eventually elected. More formally, let LEADER_i be a predicate that indicates that process

[1] The impossibility result that is shown for this form of Consensus also holds for stronger versions of the problem; e.g., the version in which every process is required to make a decision eventually.

i considers itself the leader. The Election Problem is specified by the following two properties:

- $\forall r\colon\ r \models \Box\,(\exists i\colon\ \text{LEADER}_i \Rightarrow \forall j\colon\ j \neq i : \neg\text{LEADER}_j)$
- $\forall r\colon\ r \models \Box\Diamond\ \exists i\colon\ \text{LEADER}_i$

This problem is harder than Consensus, because a perfect failure detector is required to solve it. We show this in the next section.

3.2.3 Election Is Harder Than Consensus

In this section, we show that Very Weak Completeness is necessary for solving Election, that Strong Accuracy or the equivalent is necessary for solving Election, and that the conjunction of Very Weak Completeness and Strong Accuracy is sufficient for solving Election. We then show that any failure detector that can be used to solve Election can be used to implement a perfect failure detector.

In principle, any set of properties can be used to define a failure detector. Thus, it is possible to define a failure detector that is sensitive to the identity of a process. For instance, one might define a failure detector that only includes runs in which a particular process i detects failures, or in which it is impossible for a specific process i to detect the failure of specific process j. In this dissertation, we consider such failure detectors unrealistic and so only consider failure detectors that do not rely on the identity of a process. We also assume that if Strong Accuracy is not guaranteed, then at least one erroneous failure detection is possible in a run.

We use the following notation. Let a *protocol* be a many-to-many mapping from a prefix of a run to a global state. If A and B are protocols, then $A+B$ is a protocol combining A and B: given a prefix of a run, one of the protocols is chosen nondeterministically and fairly to generate the next global state.

Theorem 2 *Very Weak Completeness is necessary to solve Election.*

Proof: Let F be a failure detection protocol that generates runs that do not satisfy Very Weak Completeness. We show that it is not possible to combine any protocol E with F such that $E+F$ is an Election protocol.

By assumption, there is a run r generated by F such that $r \models (\exists i\colon\ \Box\neg\text{CRASH}_i)\wedge(\exists i\colon\ [\Diamond\text{CRASH}_i \wedge \forall j\colon\ (\Diamond\text{CRASH}_j \vee \Box\neg\text{FAILED}_j(i))])$ (there is a correct process, and some process i crashes and its failure is never detected by any correct process). Assume for contradiction that there exists a protocol E that can be combined with F such that $E+F$ is an Election protocol. Let r be a run of this protocol such that process j is correct in r ($r \models \Box\neg\text{CRASH}_j$) and such that (r,k) is the first prefix for which some process i is the leader ($(r,k) \models \text{LEADER}_i$). Let i crash: $(r,k+1) \models \text{CRASH}_i$. From the assumption that Very Weak Completeness does not hold, it is possible that the failure of i will never be detected in r by any correct process. By the assumption that $E+F$ is an Election protocol, $(r,k+1) \models \Diamond\exists i\colon\ \text{LEADER}_i$. Let process ℓ become the leader in $(r,k+1+\delta)$: $(r,k+1+\delta) \models \text{LEADER}_\ell$.

Let r' be a run of $E+F$ such that $(r',k)=(r,k)$. We can extend (r',k) so that i does not crash, all messages from i are delayed until after $(r',k+1+\delta)$, and all other processes execute the same (nondeterministic and deterministic)[2]

[2] The only nondeterministic events are *crash* events (and possibly *failed* events, depending on

events through $(r', k+1+\delta)$ as in r. Then $(r', k+1+\delta) \models \text{LEADER}_i \wedge \text{LEADER}_\ell$, contradicting the assumption that $E + F$ is an Election protocol. □

We prove the following theorem under the assumption that at most one process may fail in a run. Clearly, if more processes may fail, then the result still holds.

Theorem 3 *Strong Accuracy or the equivalent is necessary to solve Election.*

Proof: We show that if a protocol can be used to solve Election, then either Strong Accuracy holds, or Strong Accuracy can be implemented.

Let F be a failure detection protocol that generates runs that do not satisfy Strong Accuracy. We show that it is not possible to combine any protocol E with F such that $E + F$ is an Election protocol, unless Strong Accuracy can be implemented. We assume that at most one process may fail in any run.

By assumption, there is a run generated by F such that $\exists i, j$: $\Diamond(\text{FAILED}_i(j) \wedge \neg\text{CRASH}_j)$ (the failure of some process is detected before that process crashes). Assume for contradiction that there exists a protocol E that can be combined with F such that $E + F$ is an Election protocol. Let r be a run of this protocol such that $(r, k) \models \text{LEADER}_i$ and $(r, k+1) \models \text{CRASH}_i$ for some process i. Because $E + F$ is an Election protocol, $\exists \ell$: $(r, k+1) \models \Diamond\text{LEADER}_\ell$. Let j become the next leader in $\Sigma_{k+\delta-1}$, where $\delta > 1$: $(r, k+\delta) \models \text{LEADER}_j$.

It must be the case that $\exists \ell$: $(r, k+\rho) \models \text{FAILED}_\ell(i)$ where $1 < \rho < \delta$. This is because process j cannot become the leader without knowing that there is no other leader – in particular, without learning that i is no longer the leader. Process j can only gain this information if there is a process chain in r from i to j initiated

the specification of F).

after i relinquished the leadership, or if the failure of i is detected by some process that then initiates a process chain. Since i crashes immediately after becoming the leader, the latter case must hold. Without loss of generality, let process x be the first process to detect the failure of i; that is, x executes $\mathit{failed}_x(i)$ in $\Sigma_{k+\rho-1}$, where $1 < \rho < \delta$. To summarize, $(r, k) \models \text{LEADER}_i$, $(r, k+1) \models \text{CRASH}_i$, $(r, k+\rho) \models \text{FAILED}_x(i)$, and $(r, \delta) \models \text{LEADER}_j$.

Consider a prefix $(r', k+\delta)$ of a run r' of $E+F$ that is identical to $(r, k+\delta)$, except that process i does not crash in r'. Clearly, this run does not satisfy the requirements of Election, because both i and j are leaders at $(r', k+\delta)$. Hence, r' cannot be generated by $E+F$. Because the only difference between $(r, k+\delta)$ and $(r', k+\delta)$ is the crash of i, the only properties of $E+F$ that would prohibit $(r', k+\delta)$ must be part of F. In particular, removing crash_i results in all detections of i's failure becoming erroneous (e.g., the detection $\mathit{failed}_x(i)$ is accurate in $(r, k+\delta)$ but erroneous in $(r', k+\delta)$), and so F must prohibit this particular set of erroneous failure detections.

If x is the only process that has detected the failure of i in $(r, k+\delta)$ (and $(r', k+\delta)$), then r' must satisfy F, since if F were to prohibit r' then F would ensure Strong Accuracy. Therefore, more than one process must have detected i's failure in $(r, k+\delta)$, and F must prohibit at least some of these processes from making erroneous detections. However, if F were to prohibit such a set of erroneous detections, then the remaining processes could guarantee that the detection of i's failure is accurate in r as follows. Let the processes that have detected i's failure be $S \subset P$. Each process in S sends a message to all of the remaining processes indicating that i has failed. All of the remaining processes eventually receive these messages because, by assumption, there is at most one failure. Since F prohibits at least some of the processes in S from making erroneous detections of the failure of i, all of the remaining processes eventually know that i has indeed crashed. Therefore, either r' exists, contradicting the assumption that $E + F$ is an Election protocol, or Strong Accuracy can be implemented given F. □

Theorem 4 *Strong Accuracy and Very Weak Completeness are sufficient to solve Election.*

Proof: The Election problem can be solved using the following algorithm:

- Each process has a unique ID number which are known by all processes *a priori*.
- The leader is initially the process with the lowest ID number.
- If a process detects a failure, it broadcasts this information to all other processes. Upon receiving such a message, the receiver detects the failure.
- When a process detects the failure of all processes with lower ID numbers, then that process becomes the leader.

The proof that this protocol satisfies Election is straightforward. □

Theorems 2, 3, and 4 together show that a failure detector that satisfies Very Weak Completeness and Strong Accuracy is the weakest failure detector necessary to solve Election. However, these two properties together are strong enough to implement a perfect failure detector, as shown in the following theorem.

Theorem 5 *If a failure detector is sufficient to solve Election, then that failure detector can be used to implement a perfect failure detector.*

Proof: It is shown in ([CT91]) that a failure detector satisfying Strong Accuracy and Weak Completeness can be used to implement a perfect failure detector. Given Strong Accuracy and Very Weak Completeness it is easy to implement Weak Completeness by requiring suspicions to be stable; i.e., once $\text{SUSPECT}_i(j)$ becomes true, it remains true. Therefore, a failure detector that satisfies Strong Accuracy and Very Weak Completeness can be used to implement a perfect failure detector. The theorem then follows from Theorems 2 and 3. □

This theorem implies that the conjunction of Very Weak Completeness and Strong Accuracy is equivalent to a perfect failure detector; hence, a perfect failure detector is the weakest failure detector that is sufficient to solve Election.

3.3 Group Membership

The Group Membership problem (GMP) is the basis of such distributed systems as Isis ([BJ87]), Consul ([MPS91a]), and Transis ([ADKM92]). These systems provide services that allow processes to form cooperative groups, and to exchange messages within and between groups in a consistent way. Though formal specifications of GMP have been attempted recently (e.g., [MBRS94,JRF93,HS93,ADKM93,RB91, MSMA91]; but see [ACBMT95]), there is not yet general agreement on the definition of GMP. Typically, each member of a process group maintains a *view* of the group membership. When the group membership changes, due to a process joining, leaving voluntarily, or crashing, all group members must eventually update their views; furthermore, all group members must eventually share the same view. Thus, any solution to GMP requires a failure detector.

Currently, systems that solve GMP implement various types of imperfect failure detectors, but do not formalize the properties of these implementations. For instance, the Isis system ([BJ87,RB91]) allows failures to be detected erroneously, and only requires that all correct processes in a group eventually agree on the order in which incorrect group members fail. The Consul system ([MPS91a]) imposes the additional requirement that correct processes in a group agree on the last message sent by a failed process. However, the properties of the failure detector are generally not specified in isolation; instead, the properties are implicitly assumed or are implied by the properties of the system in general. For example, in [RB91] it is shown that in order for system views to be unique, any process that initiates a view change for its group must receive messages from a majority of group members. This implies that if a majority of processes fail, the uniqueness property cannot be guaranteed.

In [HS93] and [Ric95b,Ric95c] some properties provided by membership services are isolated and specified. Some of these properties are harder than Consensus, and some are not. For instance, [HS93] shows that total ordering of messages and agreement on the last message sent by a failed process are both properties that are provided by typical systems. These properties cannot be implemented without a perfect failure detector, and are therefore harder than Consensus. In [Ric95b] Ricciardi shows that the *Uniformity* property, which requires that either all correct processes take a given action or none do, can be implemented in an asynchronous system as long as a majority of processes do not crash; therefore, Uniformity is not as hard as Consensus. In general, it appears that a basic membership service can be provided with a weak failure detector, but that many desirable properties require stronger failure detectors.[3] In the next chapter, we present a formal specification of a failure detector that satisfies the requirements of group membership systems such as Isis. Furthermore, we show that the failure detector is indistinguishable from a perfect failure detector to all processes.

3.4 Other Problems

Given the results of the previous section, it is clear that any problem whose specification implies Election can be solved only with a failure detector at least as strong as a perfect failure detector. For example, the asynchronous version of the Primary Backup problem ([BMST92]) requires that there is no more than one primary server at any time and that there is always eventually a primary server, and so Primary Backup implies Election. It is easy to implement Primary Backup using a perfect failure detector, and so a perfect failure detector is the weakest failure detector that can be used to implement Primary Backup.

There may also be problems that do not resemble Election but require a perfect failure detector. The Terminating Reliable Broadcast problem (TRB) ([Mul93,

[3]Chandra, Hadzilacos, and Toueg show in [CHT95] that even a very weak specification of the Group Membership problem cannot be solved in an asynchronous system. However, their specification does not allow two processes in a group to hold different views temporarily before agreeing on a new view.

CT94]) appears to be one example. In this problem, if a correct process sends a message, then that message is eventually received by all other correct processes; if a faulty process sends a message, then all correct processes eventually receive the same message. In [Mul93] it is shown that in an asynchronous system, TRB is strictly harder than Consensus. In [CT94], it is claimed (without proof) that the weakest failure detector for solving TRB is a perfect failure detector. We do not know whether Election is equivalent to TRB.

Generalized Repeated Coordination refers to problems that require repeated instances of some kind of agreement (e.g., repeated Consensus, repeated Reliable Broadcast). These can be shown to require a perfect failure detector ([Ric95a]). The Election problem is an example of such a problem: whenever a new leader is elected, the remaining non-leader processes implicitly "agree" that none of them is the leader.

In [SS94], it is shown that *Primary Partition Virtually-Synchronous Communication* (PP-VSC) cannot be implemented with $\diamond\mathcal{W}$, the weakest failure detector that can implement Consensus. This problem arises in systems that use group membership protocols and resembles the Terminating Reliable Broadcast problem mentioned above. [SS94] does not establish the weakest failure detector for solving PP-VSC, but it does show that a failure detector weaker than a perfect failure detector is strong enough to solve the problem. Hence, PP-VSC appears to be harder than Consensus and easier than Election.

There are problems that are harder than Election as well. In Chapter 5 we define failure detectors that are stronger than perfect (for example, a failure detector that guarantees that the failure of a process is detected only after all messages that it has sent have been received by the detecting process). This failure detector is required by some problems, including the non-blocking version of the asynchronous Primary Backup problem ([BMST92]). It may be possible to define a hierarchy of problems ranked according to the type of failure detector that they require.

3.5 Summary

In this chapter, we have shown that there are simple and practical problems that are harder than Consensus in that they require a perfect failure detector to be solvable in an asynchronous system. This motivates our examination of *approximations* of perfect failure detectors. With an approximation, one can construct approximate solutions to problems such as Election. In many cases, an approximation is sufficient.

In the next chapter, we will give a formal definition of approximation, and argue that using this approximation leads to useful solutions to problems that require a perfect failure detector.

Chapter 4

Approximating Fail-Stop

4.1 The Fail-Stop Failure Model and Perfect Failure Detectors

The *fail-stop* model has been studied since 1983, when the term “fail-stop” was introduced in the context of shared-memory distributed systems ([SS83]). A fail-stop processor was defined as a processor of which the only visible effects of a failure are: (1) the processor stops executing; (2) the processor’s internal state and the contents of its volatile memory are lost (though the contents of stable storage are not lost); (3) other nonfaulty processors can detect the failure. This definition has since been modified for use in asynchronous message-passing systems. “Fail-stop” is now commonly used to refer to the assumptions that only crash failures can occur and that all crash failures are eventually detected by correct processes. The fail-stop model is appealing because it makes distributed algorithms easier to formulate: fail-stop failures are easy to tolerate. However, as implied by [FLP85] and shown this section, the fail-stop model cannot be implemented.

The minimal set of fail-stop assumptions found in the literature is that in any infinite run of the system, a process's failure is eventually detected by all processes that do not crash, and that there are no erroneous detections of failure. These two properties specify the failure model defined in [Sch84]. Hence, we adopt this as the definition of the fail-stop failure model. Formally, the two fail-stop properties are:

FS1: $\forall r, i:\ r \models \Box(\text{CRASH}_i \Rightarrow \forall j:\ \Diamond(\text{CRASH}_j \vee \text{FAILED}_j(i)))$

FS2: $\forall r, i, j:\ r \models \Box(\text{FAILED}_i(j) \Rightarrow \text{CRASH}_j)$

We denote with **FS** the set of runs satisfying properties **FS1** and **FS2**. This definition corresponds to processes using the *Perfect Failure Detector* of [CT91] (denoted $\mathcal{P}$): $\mathcal{P}$ is defined in [CT91] as the failure detector satisfying Strong Completeness and Strong Accuracy, which are equivalent to **FS1** and **FS2**.

The fail-stop model is very useful for solving otherwise-impossible problems in asynchronous distributed systems. In particular, the failure detector $\mathcal{P}$ is sufficient for solving Consensus ([CT91,CHT92]), and is necessary for solving Election ([SM95]). This implies that implementing such a failure detector is impossible in asynchronous systems.

Theorem 6 *In an asynchronous system in which crash failures are possible, it is impossible to implement the failure detector $\mathcal{P}$.*

Proof: In [CT91], an algorithm is given for solving Consensus with a Strong Failure Detector (denoted $\mathcal{S}$). $\mathcal{S}$ is shown to be strictly weaker than $\mathcal{P}$, implying that $\mathcal{P}$

can also be used to solve Consensus. A solution to Consensus contradicts the result of [FLP85]; therefore, $\mathcal{P}$ cannot be constructed. □

4.2 Indistinguishability

Though the fail-stop model is impossible to implement, its usefulness in solving fundamental problems leads us to explore the possibility that weaker approximations of the fail-stop model might also be useful. To this end, we define *indistinguishability* of failure models.

A process determines which event to execute based on its state and the messages that it has received. A run r is isomorphic to a run r' with respect to a process i if i executes the same sequence of events in both r and r'. Therefore, $r \equiv_i r'$ if i starts in the same initial state in both runs, receives the same messages in the same order in both runs, and makes the same nondeterministic choices (if any) in both runs. Consider a run r of a system. If r is not in **FS** but is isomorphic with respect to i to a run r' in **FS**, then the events that i executes are the same as if it were running in a system satisfying the fail-stop assumptions. Hence, if $r \equiv_P r'$, then no process can determine that r is not in **FS**.

Suppose that we define a failure model $\mathcal{M}$ such that all runs in $\mathcal{M}$ are isomorphic to runs in **FS**. If $\mathcal{M}$ were implemented in an asynchronous distributed system, then at any time, the state of each process in the system would be consistent with the fail-stop model; that is, the individual processes would behave as if a perfect failure detector were implemented. No process would be able to distinguish a system in which $\mathcal{M}$ were implemented from one in which **FS** were implemented. This leads us to the following definition.

Definition 10 *A failure model* $\mathcal{M}$ *is* indistinguishable *from the fail-stop model if for any run* $r \in \mathcal{M}$*, there exists a run* $r' \in \mathbf{FS}$ *such that* $r \equiv_P r'$ *(that is,* r *is indistinguishable from* r' *to every process).*

A failure model that is indistinguishable from fail-stop retains many of fail-stop's useful properties. Consider the Election problem. Assuming a fail-stop failure model leads to a very simple Election protocol, described in the proof of Theorem 4 in Section 3.2.3 and summarized as follows. Each process has a unique ID number. The process with the lowest ID number is the leader. When i has detected the failure of all processes with lower ID numbers, i becomes the leader.

If a run of this protocol is in a failure model $\mathcal{M}$ that is indistinguishable from but not identical to **FS**, then there may be more than one leader in some global state (i.e., if **FS2** does not hold), but no process will be able to determine this. Internally the execution is the same as if there were only one leader at a time, because every process takes the same actions that it might take if the run were in **FS**. Therefore, any safety property of the Election protocol that can be determined by an individual process and that holds under under **FS** will hold under $\mathcal{M}$ as well.

As another example, consider the Consensus problem. A simple Consensus protocol that assumes **FS** is as follows. Use an Election protocol to guarantee that there is eventually a single leader at any time. The leader broadcasts its initial value to all other processes, waits for acknowledgements from all other processes that have not failed, and then broadcasts a "commit" message. Upon receiving the commit message, the other processes decide on the leader's value. If the leader fails and the process that becomes the new leader has not received a "commit" message, the new leader re-starts the protocol, using the original leader's value if it received it.

If this protocol is run using a model that is indistinguishable from but weaker than fail-stop, then a new leader may take over before the previous leader has failed. However, such a situation will not be noticeable to the other processes, because it is possible under the fail-stop model that a new leader would take over even after the original leader had broadcast the "commit" message. Thus, the correctness of the protocol is not compromised.

These examples show that a failure model that is indistinguishable from fail-stop is an approximation to fail-stop that can be used in practice. Furthermore, such an approximation is sufficiently weaker than fail-stop in that it can be implemented in practice, as will be shown in later sections.

We can slightly weaken the definition of indistinguishability and still retain the usefulness of indistinguishability as a measure of approximation. Suppose that the failure model $\mathcal{M}$ is implemented by adding a "failure detector layer" underneath the "application layer" of the system; i.e., processes execute instructions that cause failures to be detected according to the specification of $\mathcal{M}$, but these instructions are independent of any application program that the processes might be running, so that programs written to use the output of the failure model would not have knowledge of the implementation of the failure model. If the sequence of events that are executed by a process in the application layer is isomorphic to a sequence that could be executed given a perfect failure detector, then that process will not be able to distinguish the run from one in the fail-stop model. Thus, we can abstract away the instructions that are used only to implement the failure model. In particular, we can treat application events and messages separately from events and messages within the failure detector layer. This leads to a modification of the definition of indistinguishability.

Definition 11 *A failure model $\mathcal{M}$ is* indistinguishable *from the fail-stop model if for any run $r \in \mathcal{M}$, there exists a run $r' \in \mathbf{FS}$ such that $r \equiv_P r'$ with respect to application events.*

For the remainder of this dissertation, we use the above definition of indistinguishability as our notion of failure model approximation:

Definition 12 *An* approximate fail-stop *model is a failure model that is indistinguishable from the fail-stop model. An* approximately perfect *failure detector is a failure detector that is specified by an approximate fail-stop model, where* indistinguishable *is defined as in Definition 11.*

We can redefine the happens-before relation so that only application messages affect the relation. Let $\rightsquigarrow$ denote the happens-before relation with respect to application messages only (i.e., ignoring messages used to implement the failure model). Given two events e_1 and e_2, define $e_1 \rightsquigarrow e_2$ in some history $\mathcal{H}_r$ if one of the following three conditions holds:

1. e_1 and e_2 are of the same process, and either $e_1 = e_2$ or e_1 precedes e_2 in $\mathcal{H}_r$;

2. $e_1 = send_i(j, m)$ and $e_2 = recv_j(i, m)$ for some values of i, j, and m, where m is an application message;

3. there exists an event e such that $e_1 \rightsquigarrow e$ and $e \rightsquigarrow e_2$ in $\mathcal{H}_r$.

4.3 Necessary Conditions

Recall that the reason that fail-stop cannot be implemented in an asynchronous system is because the crash of a process cannot be reliably detected. A failure model $\mathcal{M}$ that can be implemented and is indistinguishable from **FS** must be weaker than **FS**. However, it cannot be too weak; intuitively, a process i must not be able to determine that some process j executes an event after i detects that j has crashed. Furthermore, if process i detects the failure of j then j must crash at some point, and observed process crashes must be seen to occur in some total order. This intuition leads to the following three conditions. We show in Theorem 10 that these conditions are necessary for indistinguishability from **FS**.

Condition 7 *For all runs r, if $r \models \Diamond \text{FAILED}_i(j)$, then $r \models \Diamond \text{CRASH}_j$. That is, if i observes the failure of j in some run, then j fails in that run.*

Condition 8 *The failed-before relation must be acyclic. That is, for all runs r and for all k, there cannot exist processes $x_1, x_2, \ldots, x_k$ such that $r \models \Diamond[\text{FAILED}_{x_1}(x_2) \wedge \text{FAILED}_{x_2}(x_3) \wedge \cdots \wedge \text{FAILED}_{x_{k-1}}(x_k) \wedge \text{FAILED}_{x_k}(x_1)]$. We will refer to this condition as the* Acyclicity *condition.*

Condition 9 *Let* $\rightsquigarrow$ *denote the happens-before relation with respect to application messages only (i.e., ignoring messages used to implement the failure model). For all runs* r*, there cannot be an event* e_j *of process* j *such that* $\mathrm{failed}_i(j) \rightsquigarrow e_j$ *in* $\mathcal{H}_r$*.*

Theorem 10 *If failure model* $\mathcal{M}$ *is indistinguishable from* **FS***, then all runs of* $\mathcal{M}$ *satisfy Conditions 7–9.*

Proof:

Condition 7 In order for two runs to be isomorphic, their histories must contain the same events. For every run r in **FS**, $\mathit{failed}_i(j) \in \mathcal{H}_r \Rightarrow \mathit{crash}_j \in \mathcal{H}_r$. Therefore, the same must be true of every run in $\mathcal{M}$.

Condition 8 (Acyclicity) For contradiction, suppose that there is some run r of $\mathcal{M}$ such that r does not satisfy Condition 8. We show that there is no run r' satisfying **FS** that is isomorphic to r with respect to P.

If r does not satisfy Condition 8, then there is some set of processes $\{x_0, x_1, \ldots, x_k\}$ such that $\mathcal{H}_r$ contains the events $\mathit{failed}_{x_0}(x_1)$, $\mathit{failed}_{x_1}(x_2)$, $\ldots$, $\mathit{failed}_{x_{k-1}}(x_k)$, $\mathit{failed}_{x_k}(x_0)$ in some order. For any isomorphic run r' in **FS**, $\mathcal{H}_{r'}$ must contain crash_{x_i} for all $0 \leq i \leq k$. Furthermore, crash_{x_i} must occur before $\mathit{failed}_{x_{i\ominus 1}}(x_i)$ and $\mathit{failed}_{x_i}(x_{i\oplus 1})$ must occur before crash_{x_i} in $\mathcal{H}_{r'}$, where $\ominus$ and $\oplus$ are $-$ and $+$ modulo $k+1$ respectively. By transitivity, this leads to circular constraints on $\mathcal{H}_{r'}$: crash_{x_0} must occur before $\mathit{failed}_{x_k}(x_0)$, which must occur before crash_{x_k}, which must occur before $\mathit{failed}_{x_{k-1}}(x_k)$, $\cdots$, crash_{x_1} must occur before $\mathit{failed}_{x_0}(x_1)$, which must occur before crash_{x_0}.

It is impossible to satisfy all of these ordering constraints in a valid run. Therefore, there is no run r' isomorphic to r that is in **FS**.

Condition 9 For contradiction, suppose that there is some run r in $\mathcal{M}$ such that r does not satisfy Condition 9. We will show that there is no run r' in **FS** that is isomorphic to r with respect to P.

If r does not satisfy Condition 9, then $\mathcal{H}_r = (\cdots \mathit{failed}_i(j) \cdots \mathit{send}_i(k, m_k) \cdots \mathit{recv}_j(\ell, m_j) \cdots e_j \cdots)$, where messages m_j, m_k are application messages and $\mathit{send}_i(k, m_k) \rightsquigarrow \mathit{recv}_j(\ell, m_j)$. For any r' isomorphic to r, $\mathcal{H}_{r'}$ must maintain the order of $\mathit{failed}_i(j)$, $\mathit{send}_i(k, m_k)$, and $\mathit{recv}_j(\ell, m_j)$. However, for r' to be in **FS**, crash_j must occur before $\mathit{failed}_i(j)$ in $\mathcal{H}_{r'}$. This means that in $\mathcal{H}_{r'}$, crash_j must occur before $\mathit{recv}_j(\ell, m_j)$, which contradicts the definition of crash_j. Therefore, there is no run r' in **FS** that is isomorphic to r.

$\square$

We have shown that Conditions 7, 8, and 9 are necessary for a failure model to be indistinguishable from fail-stop. However, these conditions are not sufficient, as shown in the following theorem.

Theorem 11 *There exists a run r that satisfies Conditions 7–9 such that* $\neg\exists r' \colon (r' \equiv_P r) \wedge (r' \in \mathbf{FS})$.

Proof: Let r be the following run:

$\mathit{failed}_j(i)$; $\mathit{send}_j(k, m_k)$; $\mathit{recv}_k(j, m_k)$; crash_k; $\mathit{failed}_\ell(k)$; $\mathit{send}_\ell(i, m_i)$; $\mathit{recv}_i(\ell, m_i)$; $\mathit{crash}_i \cdots$

For any r' isomorphic to r, we have the following ordering constraints on $\mathcal{H}_{r'}$:

- $failed_j(i) \rightsquigarrow send_j(k, m_k) \rightsquigarrow recv_k(j, m_k) \rightsquigarrow crash_k$
- $failed_\ell(k) \rightsquigarrow send_\ell(i, m_i) \rightsquigarrow recv_i(\ell, m_i) \rightsquigarrow crash_i$
- $crash_i$ must occur before $failed_j(i)$
- $crash_k$ must occur before $failed_\ell(k)$

It is impossible to satisfy all of these ordering constraints in a valid run. Therefore, there is no run r' isomorphic to r that satisfies **FS**. $\square$

Theorem 11 implies that a failure model $\mathcal{M}$ that satisfies Conditions 7–9 may not be indistinguishable from **FS**. In the next section, we give a set of conditions that are sufficient, though not all are necessary.

4.4 The Quasi Fail-Stop Model

We give four properties that specify a model that is indistinguishable from fail-stop. We call this model the *quasi fail-stop* model (**qFS**).

To construct properties for **qFS**, we weaken one of the properties of the fail-stop model. Weakening **FS1** yields a model in which some failures may be undetected. Under such a model, it could be impossible for a system to make progress. Therefore, we follow [CT91,CHT92,RB91] and weaken **FS2**. This yields a model in which nonexistent failures may be detected.

FS1 is a liveness property. It can be trivially implemented by having each process automatically detect the failure of every other process. Such an implementation would not be useful in practice. In a real system, **FS1** would be implemented using timeouts: each process would periodically send a message to every other process. If process i were not to receive a message from process j within some predetermined length of time, then i would (perhaps erroneously) detect the failure of j. The usefulness of this method depends upon the amount of synchrony present in the system. We assume for the remainder of this dissertation that there is some mechanism provided by the underlying system to implement **FS1**.

We replace **FS2** with the following four properties.

$$\textbf{qFS2a}: \forall r, i, j: \; r \models \Box(\text{FAILED}_i(j) \Rightarrow \Diamond\text{CRASH}_j)$$

This property states that if process i detects that process j has crashed, then eventually j will crash even if i's detection was erroneous. This property implies Condition 7: if $\mathit{failed}_i(j)$ occurs in $\mathcal{H}_r$, then crash_j occurs in $\mathcal{H}_r$.

qFS2b : The failed-before relation is acyclic in all runs.

This is Condition 8.

$$\textbf{qFS2c}: \forall r, i: \; r \models \Box\neg\text{FAILED}_i(i)$$

This property states that a process never detects its own failure. That is, $\mathit{failed}_i(i)$ does not occur in $\mathcal{H}_r$.

qFS2d : $\forall r, i, j, k, m$ where m is an application message: $r \models$

$$\Box[\text{FAILED}_i(j) \wedge \neg\text{SEND}_i(k,m) \Rightarrow \Box((\text{SEND}_i(k,m) \wedge \text{RECV}_k(i,m)) \Rightarrow \text{FAILED}_k(j))]$$

This property states that once i detects the failure of j, then any subsequent application messages sent by i to any process k will not be received until k has also detected the failure of j. That is, if $send_i(k, m)$ occurs after $failed_i(j)$ in $\mathcal{H}_r$, then $failed_k(j)$ occurs before $recv_k(i, m)$ in $\mathcal{H}_r$.

Properties **qFS2c** and **qFS2d** together imply Condition 9, as shown in the following lemma.

Lemma 12 *If* **qFS2c** *and* **qFS2d** *hold in run r, then there cannot be an event e_j of process j such that* $\text{failed}_i(j) \rightsquigarrow e_j$ *in $\mathcal{H}_r$.*

Proof: Consider any run r. If $i = j$, then the lemma is trivially true, because from **qFS2c**, $failed_i(i)$ does not appear in $\mathcal{H}_r$. Assume that $i \neq j$. For contradiction, let e_j be an event of j such that $failed_i(j) \rightsquigarrow e_j$ in $\mathcal{H}_r$. Since $failed_i(j)$ and e_j are of different processes, from the definition of the $\rightsquigarrow$ relation there is a sequence of events $failed_i(j) \rightsquigarrow send_i(k_1, m_{k_1}) \rightsquigarrow recv_{k_1}(i, m_{k_1}) \rightsquigarrow send_{k_1}(k_2, m_{k_2}) \rightsquigarrow \cdots \rightsquigarrow recv_j(k_t, m_{k_{t+1}}) \rightsquigarrow e_j$, where each m_{k_i} is an application message. From **qFS2d**, each process in this chain, including j, must have detected the failure of j by the time it receives its message. Therefore, $failed_j(j)$ must occur in $\mathcal{H}_r$, which contradicts **qFS2c**. $\square$

The **qFS** properties are summarized in Figure 4.1.

Theorem 13 *The quasi fail-stop model* (**qFS**) *is indistinguishable from the fail-stop model* (**FS**).

In order to prove that for any run r that satisfies **FS1** and **qFS2a-d** there is an isomorphic run r' that satisfies **FS1** and **FS2**, we first determine the conditions under which an event in a history $\mathcal{H}_r$ can be moved to yield a history $\mathcal{H}_{r'}$ such that $r \equiv_P r'$.

qFS1: **FS1**

qFS2a: $r \models \Box(\text{FAILED}_i(j) \Rightarrow \Diamond\text{CRASH}_j)$

qFS2b: The failed-before relation is acyclic.

qFS2c: $r \models \Box\neg\text{FAILED}_i(i)$

qFS2d: $r \models \Box[\text{FAILED}_i(j) \wedge \neg\text{SEND}_i(k,m) \Rightarrow$

$\Box((\text{SEND}_i(k,m) \wedge \text{RECV}_k(i,m)) \Rightarrow \text{FAILED}_k(j))]$

Figure 4.1: Quasi Fail-Stop Properties

Lemma 14 *Given $\mathcal{H}_r = (\ldots e_t, \ldots, e_u \ldots)$ corresponding to run r, event e_u can be moved before e_t to yield $\mathcal{H}_{r'} = (\ldots e_u, e_t, \ldots)$ such that $r \equiv_P r'$ if and only if $\neg(e_t \rightarrow e_u)$ in $\mathcal{H}_r$.*

Proof: Consider $\mathcal{H}_r = (\ldots e_t, e_{t+1}, e_{t+2} \ldots)$ corresponding to run $r = (\ldots, \Sigma_t, \Sigma_{t+1}, \Sigma_{t+2}, \ldots)$. Suppose that e_t and e_{t+1} are of the same process i. Since e_t changes the state of i, the state of i is not the same in Σ_t as in Σ_{t+1}. Therefore, e_{t+1} cannot occur in Σ_t.

Now suppose that e_t and e_{t+1} are of two different processes i and j, respectively. The state of j in Σ_t is the same as that in Σ_{t+1}, because e_t does not change the state of j. Therefore, if e_{t+1} is not a receive event, then e_{t+1} can occur in Σ_t. If e_{t+1} is a receive event, and changes the state of any incoming channel other than $C_{i,j}$, then e_{t+1} can occur in Σ_t, because the states of all other incoming channels must be the same in Σ_t and Σ_{t+1}. However, if $e_{t+1} = recv_j(i,m)$ and $e_t = send_i(j,m)$, then e_{t+1} cannot occur in Σ_t, because the message m is not part of $\mathcal{X}_{i,j}$ in Σ_t.

In summary, e_{t+1} cannot occur in Σ_t if and only if

- e_t and e_{t+1} are of the same process, or
- $e_t = send_i(j, m)$ and $e_{t+1} = recv_j(i, m)$.

In other words, e_{t+1} cannot occur in Σ_t if and only if $(e_t \rightarrow e_{t+1})$.

Assume that e_{t+1} can occur in Σ_t, and let $\Sigma'_{t+1} = e_{t+1}(\Sigma_t)$. It can be shown by a similar argument that e_{t+1} cannot change the state of i, $\mathcal{X}_{i,j}$, or $\mathcal{X}_{j,i}$ in such a way as to violate the preconditions for e_t, so e_t can always occur in Σ'_{t+1}. Furthermore, $e_t(e_{t+1}(\Sigma_t)) = e_{t+1}(e_t(\Sigma_t))$. Therefore, $r' = (\ldots \Sigma_t, \Sigma'_{t+1}, \Sigma_{t+2}, \ldots)$ is a valid run, where $\mathcal{H}_{r'} = (\ldots e_{t+1}, e_t, e_{t+2} \ldots)$.

Consider $r_{\{i,j\}}$ and $r'_{\{i,j\}}$. (Recall that repeated states are removed in these sequences – see Definition 6 in Chapter 2.) From the construction of r', $r_i = r'_i$ and $r_j = r'_j$. Since e_t and e_{t+1} do not change the states of processes other than i and j, $r_k = r'_k$ for all processes $k \notin \{i, j\}$. Therefore, $r \equiv_P r'$.

We have shown that if $\neg(e_t \rightarrow e_{t+1})$ in $\mathcal{H}_r$, then e_{t+1} can be moved before e_t to yield $\mathcal{H}_{r'}$ such that $r' \equiv_P r$. It can also be shown that for any two events e_t and e_u in $\mathcal{H}_r$ such that $t < u$ and $\neg(e_t \rightarrow e_u)$, e_u can occur in Σ_t, e_t can occur in $e_u(\Sigma_t)$, and $e_u(e_t(\Sigma_t)) = e_t(e_u(\Sigma_t))$. Therefore, e_u can be moved to directly before e_t to yield $\mathcal{H}_{r'}$ such that $r \equiv_P r'$. $\square_{Lemma}$

We can now prove the theorem.

Proof of Theorem 13: Let r be a run in **qFS**. We must show that there is a run $r' \in$ **FS** such that $r \equiv_P r'$. If r satisfies **FS2** then the theorem trivially holds, so we assume that r violates **FS2**. Then, there exists at least one pair of processes i and j such that $r \models \Diamond(\text{FAILED}_j(i) \wedge \neg\text{CRASH}_i)$. For each such pair, by **qFS2a**, $r \models \Diamond\text{CRASH}_i$. Therefore, $\mathcal{H}_r$ is of the form $(\cdots \mathit{failed}_j(i) \cdots \mathit{crash}_i \cdots)$.

Definition 13 *A pair of processes* (i, j) *is* **bad** *in* $\mathcal{H}_r$ *if* $\mathcal{H}_r = (\cdots \text{ failed}_j(i) \cdots \text{ crash}_i \cdots)$. *Otherwise,* (i, j) *is* **good** *in* $\mathcal{H}_r$*.*

We prove the theorem by induction on the number of bad process pairs in $\mathcal{H}_r$.

Base case Assume that there is only one bad pair in $\mathcal{H}_r$. Let $\mathcal{H}_r = (x; \mathit{failed}_j(i); y; \mathit{crash}_i; z)$ where x, y, and z are sequences of events. Let k be the number of events in y. We construct by induction on k a run r' isomorphic to r such that $\mathcal{H}_{r'} = (x'; \mathit{crash}_i; \mathit{failed}_j(i); y'; z)$ where x' and y' are sequences of events.

> **Base Case** (Inner Induction) Assume $k = 0$. Then $\mathcal{H}_r = (x; \mathit{failed}_j(i); \mathit{crash}_i; z)$. Since crash_i and $\mathit{failed}_j(i)$ are of different processes, by Lemma 14 they can be swapped to yield $\mathcal{H}_{r'} = (x; \mathit{crash}_i; \mathit{failed}_j(i); z)$ such that $r' \equiv_P r$. Clearly, r' satisfies **FS2**.
>
> **Induction case** (Inner Induction) Assume that the theorem holds for all histories in which $k \leq \ell - 1$, and assume that $k = \ell$. $\mathcal{H}_r = (x; \mathit{failed}_j(i); e_1; e_2; \cdots; e_l; \mathit{crash}_i; z)$. By Lemma 12 we know that $\neg(\mathit{failed}_j(i) \rightarrow \mathit{crash}_i)$. Let e_u be the first event of $(e_1; \cdots; \mathit{crash}_i)$ such that $\neg(\mathit{failed}_j(i) \rightarrow e_u)$. Since e_u is the first such event and $\rightarrow$ is transitive, $\forall x : \ 1 \leq x < u : \neg(e_x \rightarrow e_u)$. Let $Q \subset P$ be the set of processes such that $\mathit{failed}_j(i), e_1, \ldots, e_{u-1}$ are events of processes in

Q. Then e_u must be an event of a process in $\overline{Q}$. Therefore, there is a history $\mathcal{H}_{r''} = (x; e_u; \mathit{failed}_j(i); e_1; e_2; \cdots; e_{u-1}; e_{u+1}; \cdots; e_l; \mathit{crash}_i)$ such that $r'' \equiv_P r$. By the induction hypothesis there is a history $\mathcal{H}_{r'}$ of the desired form such that $r' \equiv_P r''$, and hence $r' \equiv_P r$. $\Box_{Inner\ Induction}$

Induction case Assume that there are k bad pairs in $\mathcal{H}_r$, one of which is (x, y). We will show that we can use the same inductive construction presented in the Base Case to yield a history $\mathcal{H}_{r'}$ with strictly fewer bad pairs such that $r' \equiv_P r$, so that the Inductive Hypothesis applies to $\mathcal{H}_{r'}$.

Overview: Given a bad pair (x, y), consider another pair of processes (a, b). Using a case analysis on all possible placements of $\mathit{failed}_b(a)$ and crash_a in $\mathcal{H}_r$ with respect to $\mathit{failed}_y(x)$ and crash_x, we show that using the earlier inductive construction, we can "fix" (x, y) — i.e., construct a history $\mathcal{H}_{r'}$ in which (x, y) is good — such that:

- if (a, b) is bad in $\mathcal{H}_r$, then (a, b) is either good or bad in $\mathcal{H}_{r'}$;
- if (a, b) is good in $\mathcal{H}_r$, then (a, b) is either still good in $\mathcal{H}_{r'}$, or is bad in $\mathcal{H}_{r'}$ but can be fixed without making (x, y) bad again by using a finite number of applications of the same inductive construction.

There are twelve possible placements of $\mathit{failed}_b(a)$ and crash_a with respect to $\mathit{failed}_y(x)$ and crash_x. In each case, we consider the effect on (a, b) of applying the inductive construction of the Base Case to (x, y).

1. $\cdots \quad \mathit{crash}_a \quad \cdots \quad \mathit{failed}_b(a) \quad \cdots \quad \underline{\mathit{failed}_y(x)} \quad \cdots \quad \underline{\mathit{crash}_x} \quad \cdots$

2. $\cdots$ $\mathit{failed}_b(a)$ $\cdots$ crash_a $\cdots$ $\underline{\mathit{failed}_y(x)}$ $\cdots$ $\underline{\mathit{crash}_x}$ $\cdots$

3. $\cdots$ $\underline{\mathit{failed}_y(x)}$ $\cdots$ $\underline{\mathit{crash}_x}$ $\cdots$ crash_a $\cdots$ $\mathit{failed}_b(a)$ $\cdots$

4. $\cdots$ $\underline{\mathit{failed}_y(x)}$ $\cdots$ $\underline{\mathit{crash}_x}$ $\cdots$ $\mathit{failed}_b(a)$ $\cdots$ crash_a $\cdots$

5. $\cdots$ $\mathit{failed}_b(a)$ $\cdots$ $\underline{\mathit{failed}_y(x)}$ $\cdots$ $\underline{\mathit{crash}_x}$ $\cdots$ crash_a $\cdots$

6. $\cdots$ crash_a $\cdots$ $\underline{\mathit{failed}_y(x)}$ $\cdots$ $\underline{\mathit{crash}_x}$ $\cdots$ $\mathit{failed}_b(a)$ $\cdots$

   Since only events that occur between $\mathit{failed}_y(x)$ and crash_x are moved in the inductive construction, (a, b) is independent of (x, y) in these six cases, in that fixing (x, y) has no effect on the goodness of (a, b). Thus, (x, y) becomes good and (a, b) is unchanged.

7. $\cdots$ $\mathit{failed}_b(a)$ $\cdots$ $\underline{\mathit{failed}_y(x)}$ $\cdots$ crash_a $\cdots$ $\underline{\mathit{crash}_x}$ $\cdots$

   In this case, the history $\mathcal{H}_{r'}$ resulting from an application of the construction of the base case has one of two forms, depending on whether or not $\mathit{failed}_y(x) \rightarrow \mathit{crash}_a$:

   - $\mathcal{H}_{r'} = (\cdots \mathit{failed}_b(a) \cdots \mathit{crash}_a \cdots \underline{\mathit{crash}_x}; \underline{\mathit{failed}_y(x)} \cdots)$
   - $\mathcal{H}_{r'} = (\cdots \mathit{failed}_b(a) \cdots \underline{\mathit{crash}_x}; \underline{\mathit{failed}_y(x)} \cdots \mathit{crash}_a \cdots)$

   In either case, (x, y) is now good and (a, b) remains bad.

8. $\cdots$ $\underline{\mathit{failed}_y(x)}$ $\cdots$ crash_a $\cdots$ $\underline{\mathit{crash}_x}$ $\cdots$ $\mathit{failed}_b(a)$ $\cdots$

   In this case, the history $\mathcal{H}_{r'}$ resulting from an application of the construction of the base case has one of two forms:

- $\mathcal{H}_{r'} = (\cdots \mathit{crash}_a \cdots \underline{\mathit{crash}_x};\underline{\mathit{failed}_y(x)} \cdots \mathit{failed}_b(a) \cdots)$
- $\mathcal{H}_{r'} = (\cdots \underline{\mathit{crash}_x};\underline{\mathit{failed}_y(x)} \cdots \mathit{crash}_a \cdots \mathit{failed}_b(a) \cdots)$

In either case, (x, y) is now good and (a, b) remains good.

9. $\cdots \quad \mathit{crash}_a \quad \cdots \quad \underline{\mathit{failed}_y(x)} \quad \cdots \quad \mathit{failed}_b(a) \quad \cdots \quad \underline{\mathit{crash}_x} \quad \cdots$

   In this case, the history $\mathcal{H}_{r'}$ resulting from an application of the construction of the base case has one of two forms:

   - $\mathcal{H}_{r'} = (\cdots \mathit{crash}_a \cdots \mathit{failed}_b(a) \cdots \underline{\mathit{crash}_x};\underline{\mathit{failed}_y(x)} \cdots)$
   - $\mathcal{H}_{r'} = (\cdots \mathit{crash}_a \cdots \underline{\mathit{crash}_x};\underline{\mathit{failed}_y(x)} \cdots \mathit{failed}_b(a) \cdots)$

   In either case, (x, y) is now good and (a, b) remains good.

10. $\cdots \quad \underline{\mathit{failed}_y(x)} \quad \cdots \quad \mathit{failed}_b(a) \quad \cdots \quad \underline{\mathit{crash}_x} \quad \cdots \quad \mathit{crash}_a \quad \cdots$

    In this case, the history $\mathcal{H}_{r'}$ resulting from an application of the construction of the base case has one of two forms:

    - $\mathcal{H}_{r'} = (\cdots \mathit{failed}_b(a) \cdots \underline{\mathit{crash}_x};\underline{\mathit{failed}_y(x)} \cdots \mathit{crash}_a \cdots)$
    - $\mathcal{H}_{r'} = (\cdots \underline{\mathit{crash}_x};\underline{\mathit{failed}_y(x)} \cdots \mathit{failed}_b(a) \cdots \mathit{crash}_a \cdots)$

    In either case, (x, y) is now good and (a, b) remains bad.

11. $\cdots \quad \underline{\mathit{failed}_y(x)} \quad \cdots \quad \mathit{failed}_b(a) \quad \cdots \quad \mathit{crash}_a \quad \cdots \quad \underline{\mathit{crash}_x} \quad \cdots$

    In this case, the history $\mathcal{H}_{r'}$ resulting from an application of the construction of the base case has one of four forms:

    - $\mathcal{H}_{r'} = (\cdots \mathit{failed}_b(a) \cdots \mathit{crash}_a \cdots \underline{\mathit{crash}_x};\underline{\mathit{failed}_y(x)} \cdots)$

- $\mathcal{H}_{r'} = (\cdots \mathit{failed}_b(a) \cdots \underline{\mathit{crash}_x}; \underline{\mathit{failed}_y(x)} \cdots \mathit{crash}_a \cdots)$
- $\mathcal{H}_{r'} = (\cdots \underline{\mathit{crash}_x}; \underline{\mathit{failed}_y(x)} \cdots \mathit{failed}_b(a) \cdots \mathit{crash}_a \cdots)$
- $\mathcal{H}_{r'} = (\cdots \mathit{crash}_a \cdots \underline{\mathit{crash}_x}; \underline{\mathit{failed}_y(x)} \cdots \mathit{failed}_b(a) \cdots)$

In the first three cases, (x, y) is now good and (a, b) remains bad; in the fourth case, (x, y) is now good and (a, b) is now good, thus reducing the number of bad pairs by two instead of one.

12. $\cdots \quad \underline{\mathit{failed}_y(x)} \quad \cdots \quad \mathit{crash}_a \quad \cdots \quad \mathit{failed}_b(a) \cdots \quad \underline{\mathit{crash}_x} \quad \cdots$

In this case, the history $\mathcal{H}_{r'}$ resulting from an application of the construction of the base case has one of four forms:

- $\mathcal{H}_{r'} = (\cdots \underline{\mathit{crash}_x}; \underline{\mathit{failed}_y(x)} \cdots \mathit{crash}_a \cdots \mathit{failed}_b(a) \cdots)$
- $\mathcal{H}_{r'} = (\cdots \mathit{crash}_a \cdots \mathit{failed}_b(a) \cdots \underline{\mathit{crash}_x}; \underline{\mathit{failed}_y(x)} \cdots)$
- $\mathcal{H}_{r'} = (\cdots \mathit{crash}_a \cdots \underline{\mathit{crash}_x}; \underline{\mathit{failed}_y(x)} \cdots \mathit{failed}_b(a) \cdots)$
- $\mathcal{H}_{r'} = (\cdots \mathit{failed}_b(a) \cdots \underline{\mathit{crash}_x}; \underline{\mathit{failed}_y(x)} \cdots \mathit{crash}_a \cdots)$

In the first three cases, (x, y) is now good and (a, b) remains good. However, in the fourth case, (x, y) is now good, but (a, b) is now bad. Thus, the number of bad pairs may not be reduced. Furthermore, for each pair (i, j) such that $\mathit{failed}_j(i)$ and crash_i appear in $\mathcal{H}_r$ in the same order with respect to $\mathit{failed}_y(x)$ and crash_x as $\mathit{failed}_b(a)$ and crash_a, there can be one more bad pair in $\mathcal{H}_{r'}$ than there is in $\mathcal{H}_r$. However, we can construct a history $\mathcal{H}_{r''}$ from $\mathcal{H}_{r'}$ in the same manner in which $\mathcal{H}_{r'}$ was constructed from $\mathcal{H}_r$, such that (a, b) is good in $\mathcal{H}_{r''}$ and (x, y) remains good in $\mathcal{H}_{r''}$ as follows.

We have $\mathcal{H}_{r'} = (\cdots \mathit{failed}_b(a) \cdots \underline{\mathit{crash}_x; \mathit{failed}_y(x)} \cdots \mathit{crash}_a \cdots)$. Recall that in the construction of $\mathcal{H}_{r'}$ from $\mathcal{H}_r$, an event e between crash_x and $\mathit{failed}_y(x)$ was moved if and only if $\neg(\mathit{failed}_y(x) \to e)$. Therefore, since $\mathit{failed}_b(a)$ was moved in the construction of $\mathcal{H}_{r'}$ and crash_a was not, it must be the case that in both $\mathcal{H}_r$ and $\mathcal{H}_{r'}$

$$\neg(\mathit{failed}_y(x) \to \mathit{failed}_b(a)) \wedge (\mathit{failed}_y(x) \to \mathit{crash}_a) \tag{4.1}$$

As shown in the case analysis, there are four possible results of applying the inductive construction to $\mathcal{H}_{r'}$. Either of the first three possibilities yields a history $\mathcal{H}_{r''}$ in which (a,b) is good and (x,y) remains good. We claim that the fourth possibility cannot occur.

Proof of claim: Suppose, for contradiction, that $\mathcal{H}_{r''} = (\cdots \mathit{failed}_y(x) \cdots \underline{\mathit{crash}_a; \mathit{failed}_b(a)} \cdots \mathit{crash}_x \cdots)$. Then by the earlier argument it must be the case that in $\mathcal{H}_{r'}$ and $\mathcal{H}_{r''}$

$$\neg(\mathit{failed}_b(a) \to \mathit{failed}_y(x)) \wedge (\mathit{failed}_b(a) \to \mathit{crash}_x) \tag{4.2}$$

$(\mathit{failed}_y(x) \to \mathit{crash}_a)$ in $\mathcal{H}_{r'}$ implies that $\mathit{failed}_a(x)$ occurs in $\mathcal{H}_{r'}$, from **qFS2d** and the definition of happens-before. Similarly, $(\mathit{failed}_b(a) \to \mathit{crash}_x)$ implies that $\mathit{failed}_x(a)$ occurs in $\mathcal{H}_{r'}$. Thus, Equations 4.1 and 4.2 imply that in $\mathcal{H}_{r'}$ both $\mathit{failed}_a(x)$ and $\mathit{failed}_x(a)$ occur in $\mathcal{H}_{r'}$, which contradicts **qFS2b**. Therefore, $\mathcal{H}_{r''}$ cannot have the assumed form, so both (a,b) and (x,y) must be good in $\mathcal{H}_{r''}$. $\square_{\mathit{Claim}}$

> Thus, if fixing (x, y) in $\mathcal{H}_r$ results in t new pairs (a_i, b_i) that are bad in $\mathcal{H}_{r'}$, then we can fix all of these pairs in t applications of the inductive construction. (Note that the t bad pairs do not interfere with each other: since all of them are bad, they all fall under one of the first 11 cases. Therefore, fixing one pair (a_i, b_i) either fixes another pair (a_j, b_j) or does not affect (a_j, b_j).)

Thus, the number of bad pairs in $\mathcal{H}_r$ can be reduced by at least one in some finite number of applications of the inductive construction given in the base case. Furthermore, this number is bounded by n.

Therefore, we can construct a history $\mathcal{H}_{r'}$ with fewer than k bad pairs such that $r' \equiv_P r$. From the Induction Hypothesis, there is a run r'' that satisfies **FS2** such that $r' \equiv_P r''$; therefore, $r \equiv_P r''$. □

4.5 The Complexity of Implementing Approximate Fail-Stop

The quasi fail-stop properties (**FS1**, **qFS2a-d**) put restrictions on the way in which failures are detected. Implementing these properties requires that processes follow a protocol for detecting failures. In this section, we give lower bounds on message complexity and replication for failure detection protocols implementing approximate fail-stop models.

In the following discussion, we assume that a process i initiates a failure detection protocol when it suspects the failure of another process j. Upon completion of the failure detection protocol, i will execute either $crash_i$ or $failed_i(j)$ for some $j \neq i$.[1]

So far, we have assumed that failure detectors and failure models are specified in a purely asynchronous environment in which crash failures can occur spontaneously. However, if assumptions can be made about the number of failures that may occur, it is possible to implement an approximate fail-stop model more efficiently. Thus, we present complexity results for a range of assumptions about the environment.

Theorem 15 *Let f be the maximum number of processes that may fail in a run. If $f > \frac{n}{2}$ then Acyclicity (Condition 8) is not implementable for accuracy assumptions weaker than Weak Accuracy (i.e., that there is a correct process whose failure is never detected).*

Proof: Assume that $f > \frac{n}{2}$ and that the failure model does not guarantee Weak Accuracy, i.e., the failure of any process may be detected. Partition the processes into two disjoint sets A and B such that $|A| = \lfloor \frac{n}{2} \rfloor$ and $|B| = \lceil \frac{n}{2} \rceil$. We construct runs r_1, r_2, and r_3 as follows:

- In r_1, all processes in set A have crashed in $(r_1, \lfloor \frac{n}{2} \rfloor)$.[2] By completeness, every process in set B will eventually detect all crashes in A.

[1] We do not make any assumptions about the mechanism used to suspect failures; in most systems, a timeout mechanism is used. However, if a correct process suspects a failure, that failure will always be detected eventually. As a result, the suspected process must eventually crash. This implies that a suspicion mechanism that generates many erroneous suspicions will cause the failure detection mechanism to generate many erroneous detections. We discuss ways to avoid this in Section 4.6.1.

[2] At most one event may occur at each step of a run, so $(r, \lfloor \frac{n}{2} \rfloor)$ is the shortest prefix of r in which all processes in A may have crashed.

- In r_2, all processes in set B have crashed in $(r_2, \lceil \frac{n}{2} \rceil)$. By completeness, every process in set A will eventually detect all crashes in B.

- In r_3, messages between sets A and B are delayed indefinitely.

$(r_1, \lfloor \frac{n}{2} \rfloor) \equiv_B (r_3, \lfloor \frac{n}{2} \rfloor)$ and $(r_2, \lceil \frac{n}{2} \rceil) \equiv_A (r_3, \lceil \frac{n}{2} \rceil)$. Therefore, in r_3 every process in B eventually detects all crashes in A, and vice versa. (Messages between A and B can be received after this point.) The failed-before relation is cyclic in r_3. $\square$

Theorem 15 holds for any type of protocol implementing any failure model indistinguishable from fail-stop. In the rest of the dissertation, we consider specific types of protocols and give complexity results for them. We assume that for all processes i and j, process i executes $suspect_i(j)$ before executing $failed_i(j)$; that is, a process does not initiate a protocol to detect a failure unless the failure was first suspected.

Definition 14 *A* quorum-based protocol *is a protocol in which a process does not execute* failed(j) *until it knows that at least* $\lfloor \frac{n}{2} + 1 \rfloor$ *processes have executed* suspect(j).

Theorem 16 *Any protocol that implements an approximate fail-stop model in an asynchronous environment in which crash failures may occur is a quorum-based protocol.*

Proof: For contradiction, assume that there is a protocol that allows any process i to execute $failed_i(j)$ after learning that fewer than $\lfloor \frac{n}{2} + 1 \rfloor$ processes suspect j. Let $K \subset P$ be a set of processes such that $|K| < \lfloor \frac{n}{2} + 1 \rfloor$, and let $K' \subset P - K$

be a set of processes such that $|K'| < \lfloor \frac{n}{2} + 1 \rfloor$. Let processes i and j be such that $i \in K$ and $j \in K'$. Let t_1, t_2, t be integers such that $t > \max(t_1, t_2)$. We construct runs r_1, r_2, and r_3 as follows.

- In r_1, process i has executed $\mathit{failed}_i(j)$ in (r_1, t_1) after receiving messages from all other processes in set K indicating suspicion of j, though j has not crashed. Messages from processes in $P - K$ to processes in K are delayed until after (r_1, t).

- In r_2, process j has executed $\mathit{failed}_j(i)$ in (r_2, t_2) after receiving messages from all other processes in set K' indicating suspicion of i, though i has not crashed. Messages from processes in $P - K'$ to processes in K' are delayed until after (r_2, t).

- In r_3, messages between sets K and K' are delayed until after (r_3, t). Processes in K and K' execute the same events through t_1 and t_2, respectively, as they do in r_1 and r_2, respectively.

In (r_3, t), both $\mathit{failed}_i(j)$ and $\mathit{failed}_j(i)$ have been executed, violating Acyclicity and therefore the assumption that the protocol implements an approximate fail-stop model. $\square$

From Theorem 16, it follows that a process must receive messages from at least $\lfloor \frac{n}{2} \rfloor$ other processes before it can detect a failure.

Definition 15 *The* quorum set Q_{ij} *of* $\mathrm{failed}_i(j)$ *is the set of processes k from which i has received messages indicating that* $\mathrm{suspect}_k(j)$ *has been executed, plus i itself if* $\mathrm{suspect}_i(j)$ *has been executed.*

A protocol that implements any approximate fail-stop model must satisfy Acyclicity, and so must avoid cycles in the failed-before relation. This can be achieved by requiring that in any run there is at least one process that participates in all failure detections. We call this property the *Witness property* ($\mathcal{W}$), because it states that the quorum sets for all failure detections must have at least one process (the *witness*) in common. The Witness property can be stated formally as follows:

$$\bigcap_{\forall i,j\colon \text{FAILED}_i(j)} Q_{ij} \neq \emptyset \qquad (\mathcal{W})$$

That is, there is some process w that is in the quorum set of all failure detections. Note that this is a stronger condition than what is necessary, for example, in the update of replicated variables [Gif79] in which only each pair of quorum sets must intersect.

4.5.1 One-Round Protocols

The simplest type of protocol for failure detection is one in which a process sends messages to other processes and then receives responses before detecting a failure.

Definition 16 *A* one-round protocol *is a protocol in which a process executes* $\text{failed}_i(j)$ *as soon as it has received a message from some number of processes* (ENOUGH).

For this class of protocols, the Witness property is necessary as well as sufficient to guarantee Acyclicity, and therefore to implement an approximate fail-stop model.

Theorem 17 *For any run r generated by a one-round protocol,* $(r \models \Box \mathit{Acyclicity}) \Rightarrow (r \models \Box \mathcal{W})$.

The full proof of the theorem is given below. To see why the theorem holds, consider the problem of avoiding cycles involving exactly two processes. Suppose that process a suspects the failure of process b. Before a can execute $\mathit{failed}_a(b)$, the failure detection protocol must ensure that $\mathit{failed}_b(a)$ has not been executed and that $\mathit{failed}_b(a)$ will not be executed in the future. The set Q_{ab} must be large enough to ensure that b, after hearing from Q_{ba}, will not execute $\mathit{failed}_b(a)$. In particular, the sets Q_{ab} and Q_{ba} must have a non-null intersection. This argument generalizes for cycles involving more than two processes.

Proof: We will show that $(\exists r\colon\ r \models \Diamond \neg \mathcal{W}) \Rightarrow (\exists r\colon\ r \models \neg \Diamond$ Acyclicity). To do this, we first assume that $\mathcal{W}$ does not hold in some state of r, i.e., that it is possible for k failures to be detected such that the quorum sets for those detections have an empty intersection. We then show that using this assumption, a run can be constructed in which there is a k–cycle in the failed-before relation.

We divide the n processes in P into k sets $S_0, \ldots, S_{k-1}$ such that for $0 \leq i \leq k-1$, $i \in S_i$; that is, processes 0 through $k-1$ are in sets S_0 through S_{k-1}, and the rest of the processes are distributed arbitrarily among S_0 through S_{k-1}. We use $\oplus$ and $\ominus$ to denote addition and subtraction modulo k.

Consider the following scenario. For all $i : 0 \leq i \leq k-1$:

1. Process i suspects the failure of process $i \oplus 1$, and sends the message $\textsc{susp}_{i,i\oplus 1}$ to all processes in P. The messages sent to the processes in set $S_{i\oplus 1}$ are delayed indefinitely.

2. As a result of Step 1, process i receives a message $\textsc{susp}_{j\ominus 1,j}$ from process $j \ominus 1$ for all $j \neq i, 0 \leq j \leq k-1$. Thus, process i does not learn that another process has suspected it of having crashed.

3. Upon receiving $\textsc{susp}_{j\ominus 1,j}$, process i suspects the failure of process j, and sends $\textsc{susp}_{i,j}$ to all processes in P. The messages sent to the processes in set $S_{i\oplus 1}$ are delayed behind the previous messages (recall that interprocess channels are FIFO with respect to $\textsc{susp}$ messages).

Process i has now received $\textsc{susp}_{k,i\oplus 1}$ messages from all processes k in $\bigcup_{j\neq i\ominus 1} S_j$. Let $Q_{i,i\oplus 1} = \bigcup_{j\neq i\ominus 1} S_j$ for all $i : 0 \leq i \leq k-1$. No process in S_i is in $Q_{i\ominus 1,i}$; in other words, for every process i in P, there is some quorum set of which i is not a member. Therefore, $\bigcap_{i=0}^{k-1} Q_{i,i\oplus 1} = \emptyset$. Furthermore, by definition of Q_{ij}, every process i has received enough $\textsc{susp}$ messages to execute $\mathit{failed}_i(i \oplus 1)$. We have $\mathit{failed}_0(1), \ldots, \mathit{failed}_{(k-2)}(k-1), \mathit{failed}_{(k-1)}(0)$ in $\mathcal{H}_r$, which represents a k-cycle in the failed-before relation. $\square$

Let f be the maximum number of crashes in any run. The necessity of the Witness property places a constraint on f as a function of n and on the number of messages for which a process must wait before detecting a failure.

The simplest way to ensure that $\mathcal{W}$ holds in any protocol is to require a process to wait for responses from every other process, except for those that are suspected to have failed, before detecting a failure. If the Weak Accuracy assumption holds (i.e., there is always at least one correct process whose failure is never detected), then this process will be a witness to every failure detection that is executed. Thus, the only requirement on the number of failures is $f < n$. However, if n is large and f is small, then each failure detection requires a process to wait for many messages, which in practice could take a long time.

An alternative implementation is to require a process to wait for a fixed, predetermined number of responses before detecting a failure. This approach reduces the size of the quorum for which a process must wait, but it places a stronger restriction on the number of failures that can occur.

Theorem 18 *If the size of the quorum set is a fixed and equal size for each failure detection, then to guarantee that $r \models \Box\mathcal{W}$ for all runs r when f failures are possible, the size of each quorum set must be strictly greater than $n(\frac{f-1}{f})$.*

Proof: We assume that in any run, no more than f failures will occur. Therefore, the largest possible cycle in a run satisfying approximate fail-stop involves f processes. We must guarantee that any f quorum sets $Q_1 \cdots Q_f$ have a nonempty intersection.

Let the size of a quorum be x. Let $y = n - x$. Suppose $y = \lceil \frac{n}{f} \rceil$. Then there is a set of f quorum sets such that $\forall i: \ \exists j: \ i \notin Q_j$. In particular, let $Q_1 = P - \{1, 2, \ldots, y\}$, $Q_2 = P - \{y+1, y+2, \ldots, 2y\}$, $\cdots$, $Q_f = P - \{n - y + 1, n - y + 2, \ldots, n\}$. By construction, each process is not a member of at least one quorum. Therefore, $\bigcap_{i=1}^{f} Q_i = \emptyset$. Clearly, such a set of quorum sets can also be constructed if $y > \lceil \frac{n}{f} \rceil$. Therefore, we must have $y < \lceil \frac{n}{f} \rceil$. From this,

$$\begin{aligned} x = n - y \quad &\Rightarrow \quad x > n - \lceil \frac{n}{f} \rceil \\ &\Rightarrow \quad x > \lfloor \frac{nf - n}{f} \rfloor \end{aligned}$$

$$\Rightarrow \quad x > \lfloor \frac{n(f-1)}{f} \rfloor$$

Therefore, the size of a quorum must be an integer strictly greater than $n(\frac{f-1}{f})$.

$\Box$

Corollary 19 *If the minimum quorum size is used in a protocol for failure detection, then it must be the case that* $n > f^2$.

Proof: The size of the quorum is equal to the number of messages that process i must receive before executing $\mathit{failed}_i(j)$ (plus 1 for i itself). In order for the protocol to make progress, at least this many processes must remain alive. Therefore, we have

$$\begin{aligned} n - f > \lfloor n(\tfrac{f-1}{f}) \rfloor \quad &\Rightarrow \quad n - f > \lfloor n - \tfrac{n}{f} \rfloor > n - \lfloor \tfrac{n}{f} \rfloor \\ &\Rightarrow \quad f < \lfloor \tfrac{n}{f} \rfloor \\ &\Rightarrow \quad f^2 < f \lfloor \tfrac{n}{f} \rfloor < n \\ &\Rightarrow \quad f^2 < n \end{aligned}$$

$\Box$

We now give a one-round protocol that implements **qFS** if $f^2 < n$, proving that the above lower bounds are tight. This protocol requires a relaxation of the FIFO property of channels. Let $\textsc{susp}_{i,j}$ be a message indicating that process i suspects the failure of process j. We require that all application messages are delivered in FIFO order and that all $\textsc{susp}$ messages are delivered in FIFO order. Furthermore, we impose the following constraint on message delivery order:

> Let m be an application message and let $\textsc{susp}$ be a suspicion message. For all processes i and j, if i sends $\textsc{susp}$ to j before sending m to j, then j receives $\textsc{susp}$ before m.

We do not require the converse.

To implement **qFS**, process i executes as follows.

- When process i first suspects the failure of process j, i sends the message $\textsc{susp}_{i,j}$ to all processes (including itself). Process i waits for $\textsc{susp}_{k,j}$ messages from other processes k and takes no other action except for sending SUSP messages until it completes the protocol or crashes. Note that once process i begins the protocol, it does not receive application messages until after the protocol is complete (though it may receive SUSP messages). This behavior is permitted by the message ordering constraints.

- When process i has received $\textsc{susp}_{k,j}$ messages from more than $n(\frac{f-1}{f})$ processes (including itself), i executes $failed_i(j)$.

- When process x receives a $\textsc{susp}_{k,x}$ message, indicating that process x is suspected by some process k, x executes $crash_x$.

- When process x receives a $\textsc{susp}_{k,y}$ message and x does not suspect the failure of y, x suspects the failure of y and executes the first step of the protocol.

We argue that this protocol implements each of the quasi fail-stop properties.

qFS2a: Process i does not execute $failed_i(j)$ until it has sent $\textsc{susp}_{i,j}$ to all other processes, including j. Since channels are nonfaulty, j will eventually receive such a message, which will cause j to crash.

qFS2b: We use the notation $\textsc{send}_i(S, m)$ as shorthand for $(\forall p \in S\colon\ \textsc{send}_i(p, m))$.

Lemma 20 *Given the protocol of Section 4.5.1, then* $[r \models \exists S = \{1, 2, \ldots, k\}\colon (\text{FAILED}_1(2) \wedge \text{FAILED}_2(3) \wedge \cdots \wedge \text{FAILED}_{k-1}(k))] \Rightarrow [\exists q\colon\ (\text{send}_q(S,$ *"k failed"*$) \rightarrow \text{send}_q(S,$ *"k − 1 failed"*$) \rightarrow \cdots \rightarrow \text{send}_q(S,$ *"2 failed"*$))$ *in* $\mathcal{H}_r]$.

Proof: From the protocol and Theorem 18, the quorum size is sufficient to ensure $\mathcal{W}$. Therefore, $r \models \exists q\colon\ \forall i, j \in S\colon\ \text{FAILED}_i(j) \Rightarrow \text{RECV}_i(q,$ "j failed"$)$ $\Rightarrow \text{SEND}_q(S,$ "j failed"$)$. We prove the lemma by induction on k.

Base case For $k = 2$, the proof is trivial. Let $k = 3$. $S = \{1, 2, 3\}$, $r \models \text{FAILED}_1(2) \wedge \text{FAILED}_2(3)$, and $r \models \text{SEND}_q(S,$ "2 failed"$) \wedge \text{SEND}_q(S,$ "3 failed"$)$. Assume for contradiction that $send_q(S,$ "2 failed"$) \rightarrow send_q(S,$ "3 failed"$)$ in $\mathcal{H}_r$. Then, because channels are FIFO, $recv_2(q,$ "2 failed"$) \rightarrow recv_2(q,$ "3 failed"$)$ in $\mathcal{H}_r$. By the protocol, $crash_2 \rightarrow failed_2(3)$ in $\mathcal{H}_r$, so $r \models \neg\text{FAILED}_2(3)$. Therefore, it must be the case that $send_q(S,$ "3 failed"$) \rightarrow send_q(S,$ "2 failed"$)$.

Induction case Assume that the lemma is true for $k = l - 1$. For $k = l$, we have $\text{FAILED}_1(2) \wedge \text{FAILED}_2(3) \wedge \cdots \wedge \text{FAILED}_{l-1}(l)$. By the induction hypothesis, $send_q(S,$ "$l - 1$ failed"$) \rightarrow \cdots \rightarrow send_q(S,$ "2 failed"$)$ in $\mathcal{H}_r$. Assume for contradiction that $send_q(S,$ "$l - 1$ failed"$) \rightarrow send_q(S,$ "l failed"$)$ in $\mathcal{H}_r$. Then, as in the base case, $recv_{l-1}(q,$ "$l - 1$ failed"$) \rightarrow recv_{l-1}(q,$ "l failed"$)$, so $crash_{l-1} \rightarrow failed_{l-1}(l)$ in $\mathcal{H}_r$ and $r \models \neg\text{FAILED}_{l-1}(l)$. Therefore, $send_q(S,$ "l failed"$) \rightarrow send_q(S,$ "$l - 1$ failed"$)$ in $\mathcal{H}_r$. $\square_{Lemma}$

The quorum size for each failure detection is sufficient to guarantee $\mathcal{W}$. Assume for contradiction that the failed-before relation is not acyclic. Then $r \models \exists S = \{1, \ldots, k\}: \ \text{FAILED}_1(2) \wedge \cdots \wedge \text{FAILED}_{k-1}(k) \wedge \text{FAILED}_k(1)$. By Lemma 20, $\exists q\colon$ $send_q(S,$ "1 failed"$) \rightarrow send_q(S,$ "k failed"$) \rightarrow \cdots \rightarrow send_q(S,$ "2 failed"$)$ in $\mathcal{H}_r$. Thus, $recv_1(q,$ "1 failed"$) \rightarrow recv_1(q,$ "2 failed"$)$ in $\mathcal{H}_r$, $crash_1 \rightarrow failed_1(2)$ in $\mathcal{H}_r$, and $r \models \neg\text{FAILED}_1(2)$.

qFS2c: Process i cannot execute $suspect_i(i)$ without receiving at least one $\text{SUSP}_{k,i}$ message. (Recall that a process does not spontaneously suspect its own failure.) Upon receiving such a message, i crashes. Therefore, $suspect_i(i)$ is never executed.

qFS2d: Any message m sent by i to k after $failed_i(j)$ is executed must be received after the message $\text{SUSP}_{i,j}$. Upon receiving $\text{SUSP}_{i,j}$ from i, process k suspects the failure of j and initiates the failure detection protocol. Process k does not receive m until either $crash_k$ or $failed_k(j)$ is executed. Therefore, message m is not received by k unless $failed_k(j)$ has been executed.

4.5.2 Multi-Round Protocols

In order to avoid the lower bounds implied by the necessity of the Witness property, we must consider protocols that are more complex than one-round protocols. If we allow processes to exchange additional rounds of messages with other processes before detecting a failure, then it is possible to guarantee Acyclicity without the overhead of guaranteeing that $\mathcal{W}$ holds.

We present a protocol that implements an approximate fail-stop model using multiple rounds for each failure detection. Each process i has an array $status_i[1..n]$ which is used to keep track of the current status of the detection of the failure of process j. The array is initialized to $[\perp, \perp, \ldots, \perp]$. The possible values of $status_i[j]$ are:

1. $\perp$, indicating that $\textsc{suspect}_i(j)$ is false;

2. "pending", indicating that $\textsc{suspect}_i(j)$ is true but fewer than $\lfloor \frac{n}{2} \rfloor$ $\textsc{susp}_{k,j}$ messages have been received;

3. "ready", indicating that $\textsc{suspect}_i(j)$ is true and $\lfloor \frac{n}{2} \rfloor$ $\textsc{susp}_{k,j}$ messages have been received, but $failed_i(j)$ cannot yet be executed;

4. "done", indicating that $\textsc{failed}_i(j)$ is true.

We assume the same message ordering constraints as in the previous section. Furthermore, we again require that once process i begins the protocol, it does not receive application messages until after the protocol is complete (though it may receive SUSP messages).

Process i executes as follows.

- When $suspect_i(j)$ is executed, i immediately sends $\textsc{susp}_{i,j}$ to all other processes and sets $status_i[j]$ to "pending".

- When $\textsc{susp}_{k,i}$ is received from some process k, $crash_i$ is executed immediately.

- When $\textsc{susp}_{k,j}$ is received for $j \neq i$, if $\textsc{suspect}_i(j)$ is false then $suspect_i(j)$ is executed.

- When $\lfloor \frac{n}{2} \rfloor$ $\textsc{susp}_{k,j}$ messages have been received, i sets $status_i[j]$ to "ready" and continues as follows.
  - if there is a process $k \neq j$ such that $status_i[k]$ = "pending", continue waiting for $\textsc{susp}_{x,k}$ messages;
  - otherwise, for all processes k such that $status_i[k]$ = "ready", execute $failed_i(k)$, set $status_i[k]$ to "done", and end the protocol.

This protocol is similar to the one-round protocol. In fact, if i receives enough $\textsc{susp}_{k,j}$ messages before suspecting any other processes, the protocol ends in a single round, and the quorum set of $failed_i(j)$ is smaller than in the one-round protocol. If another process k is suspected during this round, however, i initiates another round to detect failure of k, and does not detect either failure until both quorum sets are complete. In this case, both suspected processes are marked "pending" in the *status* array, and each is marked "ready" when its quorum is obtained. The detections can take place only when both are "ready". Of course, more than two processes can be "pending" or "ready" at the same time. No more than a minority of the processes can be suspected at one time, or the quorum will not be achievable (from Definition 14 and Theorem 16, this constraint cannot be avoided). Thus, the protocol can take up to $\lceil \frac{n}{2} \rceil$ - 1 rounds.

Note that the quorum size in this protocol is only $\lfloor \frac{n}{2} \rfloor + 1$ (including i). This means that any two quorums will have a non-null intersection, but it is possible that any three or more quorums will not have a process in common. Therefore, the protocol does not guarantee the Witness property.

We will show that the above protocol guarantees **qFS2a-d**.

qFS2a: If process j has crashed before $failed_i(j)$ is executed, then this property is satisfied trivially. Suppose that this is not the case. $failed_i(j)$ is never executed unless $suspect_i(j)$ has been executed. When $suspect_i(j)$ is executed, $\text{SUSP}_{i,j}$ is sent to all other processes, including j. Therefore, process j will eventually receive this message, after which it will crash.

qFS2b: In Section 5.2 we give a protocol for implementing *quasi transitive fail-stop*, which is identical to **qFS** except that the failed-before relation is required to be transitive. The protocol is the same as the multi-round protocol given above, except that all "ready" failure detections are executed atomically, so that either all are executed are none is. A proof that the resulting protocol guarantees that the failed-before relation is transitive is given in Section 5.2.

Clearly, if the failed-before relation is transitive, then it is also acyclic. In the multi-round protocol given above, "ready" failure detections are not executed atomically. However, once $status_i[j]$ is "ready", either $failed_i(j)$ is executed or process i crashes. Therefore, if some sequence of *failed* events is executed in a run of the multi-round **qFS** protocol, then there is a run of the quasi transitive fail-stop protocol in which the same sequence of detection events is executed. Since the failed-before relation is acyclic in all runs of the transitive protocol, the same must be true of runs of the multi-round **qFS** protocol.

qFS2c: In order for a process to execute $failed_i(i)$ in this protocol, the process would first have to receive $\text{SUSP}_{k,i}$ from some process k. This would cause the process to crash before executing $\mathit{failed}_i(i)$.

qFS2d: Any message m sent by i to k after $\mathit{failed}_i(j)$ is executed must be received after the message $\text{SUSP}_{i,j}$. According to the protocol, when k receives $\text{SUSP}_{i,j}$ it executes $\mathit{suspect}_k(j)$ and starts the protocol, and does not receive application messages until the protocol is complete. Therefore, $\text{RECV}_k(i, m) \Rightarrow \text{FAILED}_k(j)$.

The multi-round protocol does not guarantee the Witness property, yet it implements **qFS**. Thus, $\mathcal{W}$ is not necessary to guarantee Acyclicity for protocols that are require more than one-round.

As mentioned earlier, no more than a minority of failures can be tolerated by this protocol – if a majority of processes fail, then it becomes impossible for a process to collect a quorum for any failure detection (see Definition 14 and Theorem 16). This gives us the restriction that $f < \frac{n}{2}$. This bound on the number of failures that can be tolerated is a significant improvement over the $f < \sqrt{n}$ bound that holds for one-round protocols. However, in the average case, the one-round protocol will require less time from initial suspicion of a failure to its detection. Since the multi-round protocol requires that a failure detection be postponed until a quorum has been obtained for all processes suspected concurrently, the multi-round protocol intuitively requires $O(n)$ times longer to execute. (Of course, due to the asynchrony of message transmission, a single round could potentially take longer than $O(n)$ rounds.) This implies that there is a tradeoff between fault tolerance and message complexity for protocols that implement approximate fail-stop models.

4.5.3 Improving Fault Tolerance

The bounds on the number of tolerable failures for both types of protocol arise from the requirement that the failed-before relation be acyclic. Without this requirement, a smaller quorum would be sufficient. In fact, it can be shown that properties **qFS2a**, **qFS2c**, and **qFS2d** can be implemented with a quorum size of zero, because of the reliability of channels. Thus, if a weaker approximation of fail-stop were sufficient for solving some problem, then a much cheaper failure detection protocol would be possible (though of course the resulting failure model would not be indistinguishable from fail-stop), and up to $n-1$ failures could be tolerated.

However, most problems requiring a failure detector do depend on failure detections being acyclic. As an example of sensitivity to Acyclicity, consider the problem of determining the last process to fail when recovering from a total failure ([Ske85]). Solving this problem requires that processes record information about the failures that they detect (that is, their view of the failed-before relation). As processes recover, they combine their information until enough processes have recovered to determine which was alive the longest before the total failure. If cyclic failure detection is possible, then the problem is not solvable. For example, suppose $P = \{1, 2\}$, process 1 falsely detects 2's failure, and then process 1 crashes. Process 2 detects 1's failure, proceeds with its work, and finally crashes. If process

1 were to then recover, it would conclude that it was the last to fail. In general, if cyclic detection is possible then the only possible recovery is to always wait for all crashed processes to recover.

Consider the one-round protocol of Section 4.5.1. Suppose that instead of detecting a failure after receiving $n(\frac{f-1}{f})$ messages, a process waits for messages from all processes that it has not suspected. In this case, as long as there is one process that is never suspected by any other process (i.e., Weak Accuracy holds), the Witness property will be guaranteed. This is because that process will receive and respond to all suspicion messages, and will therefore be a member of every quorum. If we assume that such a process exists, then the protocol will be correct as long as $f < n$. This bound is more reasonable than $f < \sqrt{n}$, or even $f < \frac{n}{2}$ as required by the multi-round protocol.

The same modification can be made to the multi-round protocol: instead of waiting for messages from a majority of processes for each concurrent detection before detecting a failure, a process can wait for messages from all processes that have not been suspected. However, since this would guarantee the Witness property, there would be no need for multiple rounds: a failure could be detected as soon as messages are received from all unsuspected processes.

In practice, waiting for messages from more processes causes a protocol to run longer. Thus, if it is known that the number of failures will be low, it is preferable to detect a failure as soon as possible. However, if the necessary bounds on the number of acceptable failures cannot be expected to hold, but Weak Accuracy can be guaranteed, then the longer, more lenient version of the protocol is preferable.

4.6 Future Work

4.6.1 De-Stabilizing Suspicions and Detections

So far, we have assumed that suspicions and failure detections are stable, so that a process cannot decide that a process is correct after it has been suspected. Because of this assumption, the number of suspicions is equal to the number of eventual failures in the system. Therefore, any bound on the number of tolerable failures also imposes a bound on the number of tolerable suspicions. However, we cannot control the number of suspicions that occur in a run, because they are assumed to come from a lower level of the system. This means that if the suspicion mechanism is unreliable, the number of failures will become too large for any further failures to be detected. In this case, a process that finds itself unable to obtain a quorum for a failure detection due to too many suspicions must either crash itself or wait forever.

Some systems (e.g., Isis) employ conservative suspicion mechanisms to avoid this problem in the average case. In such systems, the number of erroneous suspicions is assumed to be low, so that most failures are due to spontaneous process crashes and not to the behavior of the failure detection protocol itself. Another way to avoid the problem is to allow a process to *repent* a suspicion if it receives a later message from the suspected process. This method is used in [CT91].

If repentance were permitted, then the protocols of the previous sections could be used under a wider array of circumstances. For example, if the suspicion mechanism were not reliable, repentance of suspicions would allow a process to make progress where before it could not: instead of crashing or waiting forever, a process that had suspected too many processes would need only wait until some of the suspicions were repented. Furthermore, allowing processes to repent failure detections would allow the use of the protocols in systems in which crashed processes may recover. In this case, once a process recovered, all correct processes that had detected its failure could simply "change their minds" about the failure detection. Without repentance, the recovering process would have to assume a new identity (as in Isis) in order to avoid contradicting the detections of its failure.

The results of this chapter can be extended to allow repentance. We can follow [HS93] and [MMSA95] and assign incarnation numbers to processes. When a process crashes and recovers, it can assume a new incarnation number that is unique over the lifetime of the process. Thus, instead of suspecting or detecting the failure of process i, a process can suspect or detect the failure of $i.k$, where $i.k$ is incarnation k of process i. We can then refer to events such as $crash_{i.k}$, $failed_{i.k}(j.\ell)$, and so on.

The results presented in this dissertation do not change if this extension is used, and the protocols should work with only slight modifications. However, the bounds on f as a proportion of n would then hold for permanent failures only. This would allow the protocols to be used in environments where communication delays vary too widely to prevent false suspicions.

4.6.2 Comparison to the Failure Detectors of Chandra and Toueg

We discuss the weak failure detector hierarchy of Chandra and Toueg in Chapters 2 and 6. In Chapter 6 we suggest that one can use weak failure detectors as specifications of different assumptions about the nature of suspicions, and then ask the question: what is the weakest failure "suspector" that can be used to implement approximate fail-stop? We have not yet answered this question. However, we show in Section 4.5 (Theorem 15) that if a majority of processes can fail in a run, then approximate fail-stop cannot be solved with a failure suspector weaker than the Strong failure suspector ($\mathcal{S}$), which guarantees Weak Accuracy and Strong Completeness. We also show in Section 4.5.1 that implementing approximate fail-stop with a one-round protocol requires the Witness property, which requires Weak Accuracy, and in Section 4.5.3 we show that fault tolerance can be improved for multi-round protocols if Weak Accuracy holds.

Because of the necessity of Weak Accuracy, we conjecture that if repentance is allowed, the weakest failure suspector that can be used to implement approximate fail-stop is $\Diamond\mathcal{S}$, which satisfies Strong Completeness and Eventual Weak Accuracy. This failure suspector is shown in [CT95] to be equivalent to the $\Diamond\mathcal{W}$ failure suspector. If repentance is not allowed, then $\Diamond\mathcal{W}$ and $\Diamond\mathcal{S}$ are equivalent to $\mathcal{S}$, and so $\mathcal{S}$ would be the weakest failure suspector for implementing approximate fail-stop.

Chapter 5

Approximating Other Fail-Stop Failure Models

In this chapter, we present a hierarchy of fail-stop models. The fail-stop model defined in Section 4.1 is the weakest model in this hierarchy. The models are shown in Figure 5.1. An arrow is drawn from model A to model B if B is strictly stronger than A.

5.1 Transitive Fail-Stop

Transitive fail-stop is an extension of fail-stop in which the failed-before relation is transitive. Formally, the transitive fail-stop properties are:

TFS1: **FS1**

TFS2: **FS2**

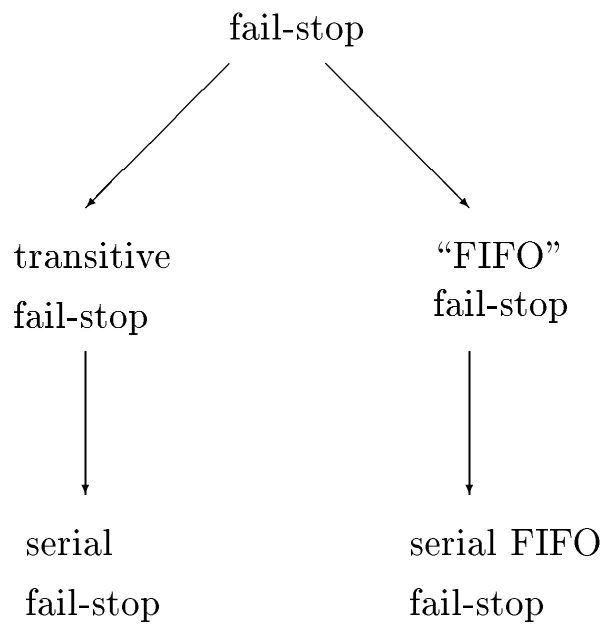

Figure 5.1: A Hierarchy of Fail-Stop Models

TFS3: $\forall r, i, j, k: \; r \models \Box(\text{FAILED}_i(j) \wedge \text{FAILED}_j(k) \Rightarrow \text{FAILED}_i(k))$

We will refer to **TFS3** as Transitivity.

The transitive fail-stop model is used in [Ske85], in which it is shown that if the failed-before relation is transitive, then determining the last process to fail after a total failure can be done more efficiently.

Since the transitive fail-stop model is stronger than the fail-stop model, it cannot be implemented in an asynchronous system. However, Transitivity can be implemented if **FS1** and **FS2** are given.

We give a protocol that implements Transitivity on top of the fail-stop model. The protocol is similar to the multi-round protocol of Section 4.5.2. The protocol assumes that process i executes the event $failed_i(j)$ if and only if $crash_j$ has occurred. We retain the notation $failed_i(j)$ to indicate that i has detected j's failure in the underlying (fail-stop) system, and we use $tfailed_i(j)$ to indicate that i detects j's failure in the transitive fail-stop protocol.

Process i behaves as follows:

when $failed_i(j)$ **do**

  $\forall x\colon\ \neg failed_i(x) : send_i(x, \text{"}f\ i\ j\text{"})$;

  when $\forall x\colon\ (\text{FAILED}_i(x) \vee \text{RECV}_i(x, \text{"}f\ x\ j\text{"}))$ **do**

    if $\exists k\colon\ failed_i(k) \rightarrow failed_i(j)$ **then**

      wait until $tfailed_i(k)$;

    $tfailed_i(j)$;

To show that this protocol implements Transitivity, we use the notation $\text{TFAILED}_i(j)$ to represent the predicate that is true in a global state if and only if $tfailed_i(j)$ has been executed. Let r be a run generated by the protocol. Assume that $\text{TFAILED}_i(j) \wedge \text{TFAILED}_j(k)$ in some global state Σ of $\mathcal{H}_r$. We will prove that $\text{TFAILED}_i(k)$ holds in Σ.

$\text{TFAILED}_i(j) \Rightarrow \text{FAILED}_i(j)$; therefore, j failed before i in $\mathcal{H}_r$. Thus, CRASH_i does not hold when $tfailed_j(k)$ is executed. Therefore, from the specification of the protocol, j must have received the message "$f\ i\ k$" from i before executing $tfailed_j(k)$. Since i does not send such a message until it executes $failed_i(k)$, it must be the case that $failed_i(k) \rightarrow send_i(j, \text{"}f\ i\ k\text{"})$. Furthermore, i does not send this message until it has executed $failed_i(j)$. Therefore, $failed_i(k) \rightarrow failed_i(j)$

in $\mathcal{H}_r$. From the **if** statement in the protocol, i will not execute *tfailed*$_i(j)$ until *tfailed*$_i(k)$ has been executed. Thus, $\text{TFAILED}_i(j) \Rightarrow \text{TFAILED}_i(k)$. □

5.2 Quasi Transitive Fail-Stop

Replacing **TFS2** with the **qFS** conditions (**qFS2a-d**) yields the *quasi transitive fail-stop model* (**qTFS**). This model is clearly indistinguishable from **TFS**.

qTFS can be implemented using the multi-round protocol given in Section 4.5.2, with the added requirement that all "ready" detections must be executed simultaneously. The proof of correctness is the same, except for the proof that **qFS2b** (Acyclicity) holds; **qFS2b** follows from the fact that Transitivity holds, as shown in the following theorem and corollary.

Theorem 21 *The multiround protocol given in Section 4.5.2 implements Transitivity, if all concurrent detections are executed simultaneously.*

Proof: We must show that if $\text{FAILED}_x(y)$ and $\text{FAILED}_y(z)$ are both true in some state, then $\text{FAILED}_x(z)$ is true in the same state.

Suppose for contradiction that there is a state Σ in some run r such that $\Sigma \models \text{FAILED}_x(y) \wedge \text{FAILED}_y(z) \wedge \neg\text{FAILED}_x(z)$. Because Q_{xy} and Q_{yz} are majorities, there is some process w such that $w \in Q_{xy} \cap Q_{yz}$. Either $\text{RECV}_y(w, \text{SUSP}_{w,z}) \wedge \text{RECV}_x(w, \text{SUSP}_{w,y})$, or $w = x$, or $w = y$. There are four cases to consider.

1. *send*$_w(\text{SUSP}_{wz}) \rightarrow$ *send*$_w(\text{SUSP}_{wy})$ in $\mathcal{H}_r$. Since channels are FIFO with respect to SUSP messages, process x must have received SUSP_{wz} before it

executed $failed_x(y)$. Therefore, x executed $suspect_x(z)$ before $failed_x(y)$ and did not execute $failed_x(y)$ until $failed_x(z)$ could also be executed. Therefore, $\text{FAILED}_x(y) \Rightarrow \text{FAILED}_x(z)$, contradicting the assumption that $\neg\text{FAILED}_x(z)$ is true in Σ.

2. $send_w(\text{SUSP}_{wy}) \rightarrow send_w(\text{SUSP}_{wz})$ in $\mathcal{H}_r$. Since channels are FIFO with respect to SUSP messages, process y must have received SUSP_{wy} before it executed $failed_y(z)$. Upon receiving this message, process y would have crashed. This contradicts the assumption that $\text{FAILED}_y(z)$ is true in Σ.

3. $w = y$. This implies $y \in Q_{xy}$. This is not possible, because y would have to execute $suspect_y(y)$ before sending $\text{SUSP}_{y,y}$ to x, and so would crash itself before such a message could be sent.

4. $w = x$. This implies that $suspect_x(z)$ is in $\mathcal{H}_r$. This must occur either before or after $suspect_x(y)$ in $\mathcal{H}_r$. If x executes $suspect_x(z)$ before $suspect_x(y)$, then $failed_x(z)$ must be executed no later than $failed_x(y)$, contradicting the assumption that $\Sigma \models \neg\text{FAILED}_x(z)$. If x executes $suspect_x(y)$ before $suspect_x(z)$, then since messages are FIFO with respect to SUSP messages, y must receive $\text{SUSP}_{x,y}$ before $\text{SUSP}_{x,z}$, and so y would crash before it could execute $failed_y(z)$. □

Corollary 22 *The multiround protocol given in Section 4.5.2 implements Acyclicity, if all concurrent detections are executed simultaneously.*

Proof: Assume for contradiction that $\exists p_0, p_1, \ldots, p_k$: $\text{FAILED}_{p_0}(p_1) \wedge \text{FAILED}_{p_1}(p_2) \wedge \cdots \wedge \text{FAILED}_{p_k}(p_0)$ is true in some state. $\text{FAILED}_{p_0}(p_1) \wedge \text{FAILED}_{p_1}(p_2) \Rightarrow \text{FAILED}_{p_0}(p_2)$

in Σ; $\text{FAILED}_{p_0}(p_2) \wedge \text{FAILED}_{p_2}(p_3) \Rightarrow \text{FAILED}_{p_0}(p_3)$ in Σ, ..., $\text{FAILED}_{p_0}(p_k) \wedge \text{FAILED}_{p_k}(p_0) \Rightarrow \text{FAILED}_{p_0}(p_0)$ in Σ, which is not possible by Theorem 21. □

5.2.1 Lower Bounds

We show in this section that implementing Transitivity requires the ability of a process to detect more than one failure simultaneously. We also show that Transitivity cannot be implemented with a simpler (one-round) protocol.

Theorem 23 *The requirement that failure detections can be executed simultaneously is necessary for implementing Transitivity with the multiround protocol.*

Proof: We will construct a run in which it is impossible for some processes to determine the order in which two failures must be detected in order to guarantee Transitivity. These processes must therefore detect both failures simultaneously to ensure that Transitivity holds.

Number each of the processes with a unique integer from 1 through n. Processes 1 and 2 concurrently suspect each other's failures (i.e., $suspect_1(2)$ and $suspect_2(1)$ are both executed, but neither $suspect_1(2) \rightarrow suspect_2(1)$ nor $suspect_2(1) \rightarrow suspect_1(2)$). Processes 3 and 4 crash shortly thereafter. Eventually, both 1 and 2 must also crash. Let I be the set of processes with odd numbers greater than 4, and let J be the set of processes with even numbers greater than 4. There are $\lceil \frac{n}{2} \rceil - 2$ processes in I and $\lfloor \frac{n}{2} \rfloor - 2$ processes in J. Let all processes in I receive (and respond to) $\text{SUSP}_{1,2}$ before $\text{SUSP}_{2,1}$, and let all processes in J receive (and respond to) $\text{SUSP}_{2,1}$ before $\text{SUSP}_{1,2}$.

Recall that a process must receive $\lfloor \frac{n}{2} \rfloor$ messages before executing a *failed* event. Processes 3 and 4 may have received and responded to $\text{SUSP}_{1,2}$ and/or $\text{SUSP}_{2,1}$ before crashing. If processes 3 and 4 received $\text{SUSP}_{1,2}$ before $\text{SUSP}_{2,1}$, then process 1 could have received $\lceil \frac{n}{2} \rceil \geq \lfloor \frac{n}{2} \rfloor$ $\text{SUSP}_{k,2}$ messages and executed $\mathit{failed}_1(2)$ before crashing. Similarly, if processes 3 and 4 received $\text{SUSP}_{2,1}$ before $\text{SUSP}_{1,2}$, then process 2 could have received $\lfloor \frac{n}{2} \rfloor$ $\text{SUSP}_{k,1}$ messages and executed $\mathit{failed}_2(1)$ before crashing. However, processes 3 and 4 may have crashed before learning about either 1's or 2's suspicions. In this case, neither $\mathit{failed}_1(2)$ nor $\mathit{failed}_2(1)$ can have been executed.

Consider the surviving processes in $I \cup J$. Each $p \in I \cup J$ must execute $\mathit{failed}_p(1)$ and $\mathit{failed}_p(2)$ eventually. If $\mathit{failed}_1(2)$ has been executed, then each $p \in I \cup J$ must execute $\mathit{failed}_p(2)$ before $\mathit{failed}_p(1)$ to guarantee Transitivity; if $\mathit{failed}_2(1)$ has been executed, then each p must execute $\mathit{failed}_p(1)$ before $\mathit{failed}_p(2)$. If neither $\mathit{failed}_1(2)$ nor $\mathit{failed}_2(1)$ has been executed, then each p can execute the detections in either order. The processes in $I \cup J$ cannot distinguish between these three cases in bounded time. Therefore, each $p \in I \cup J$ must execute $\mathit{failed}_p(1)$ and $\mathit{failed}_p(2)$ simultaneously in order to guarantee Transitivity. □

Theorem 24 *Quasi transitive fail-stop cannot be implemented with a one-round protocol.*

Proof: Assume for contradiction that processes use a one-round protocol to detect failures, and that the protocol generates runs in **qFS**. We will give a scenario in which Transitivity is violated.

Suppose that processes i and j concurrently suspect each other's failures ($\mathit{suspect}_i(j)$ and $\mathit{suspect}_j(i)$ are both executed, but neither $\mathit{suspect}_i(j) \rightarrow \mathit{suspect}_j(i)$ nor $\mathit{suspect}_j(i) \rightarrow \mathit{suspect}_i(j)$). Process i sends $\textsc{susp}_{i,j}$ to all other processes, and process j sends $\textsc{susp}_{j,i}$ to all other processes. Because there will be some process w that is in both $Q_{i,j}$ and $Q_{j,i}$ (recall that the Witness property is necessary for one-round protocols to implement **qFS**), either $\mathit{failed}_i(j)$ or $\mathit{failed}_j(i)$ may be executed, but not both: either process j will receive $\textsc{susp}_{w,j}$ before $\textsc{susp}_{w,i}$ and thus crash before executing $\mathit{failed}_j(i)$, or process i will receive $\textsc{susp}_{w,i}$ before $\textsc{susp}_{w,j}$ and crash before executing $\mathit{failed}_i(j)$.

Consider a correct process k that receives both $\textsc{susp}_{i,j}$ and $\textsc{susp}_{j,i}$. Process k will execute both $\mathit{suspect}_k(i)$ and $\mathit{suspect}_k(j)$, and, if enough processes remain correct, will eventually receive enough $\textsc{susp}$ messages to execute both $\mathit{failed}_k(i)$ and $\mathit{failed}_k(j)$. If $\mathit{failed}_i(j)$ has been executed, then k must execute $\mathit{failed}_k(j)$ before $\mathit{failed}_k(i)$ by Transitivity; if $\mathit{failed}_j(i)$ has been executed, then k must execute $\mathit{failed}_k(i)$ before $\mathit{failed}_k(j)$. However, k cannot determine whether $\mathit{failed}_i(j)$ or $\mathit{failed}_j(i)$ has been executed without receiving further messages. Therefore, Transitivity cannot be guaranteed in one round. $\square$

5.3 Serial Fail-Stop

Serial fail-stop is an extension of fail-stop in which failures are detected in the same total order by all processes. Formally, the serial fail-stop properties are:

SFS1: **FS1**

SFS2: **FS2**

SFS3: $\forall r, i, j, x, y$: $(\mathit{failed}_i(x) \to \mathit{failed}_i(y) \text{ in } \mathcal{H}_r) \Rightarrow$

$$[r \models \text{FAILED}_j(y) \Rightarrow (\mathit{failed}_j(x) \to \mathit{failed}_j(y) \text{ in } \mathcal{H}_r)]$$

This is the model that is used in many group-membership systems ([RB91, ADKM92,MPS91a]). In such systems, processes that are part of the same group must share the same view of the group at all times. Therefore, the order in which group members fail must appear the same to all processes.

The serial fail-stop model is stronger than the transitive fail-stop model, and therefore cannot be implemented in an asynchronous system. However, it can be implemented if transitive fail-stop is given: the processes need only run a Consensus algorithm to agree on the ordering for detected failures.

5.4 Approximating Serial Fail-Stop

Given an approximate transitive fail-stop implementation such as **qTFS**, one can implement an Atomic Broadcast protocol.[1] Thus, serial fail-stop can be approximated as follows. Before executing $\mathit{failed}_i(j)$, process i sends the message "j failed" using an atomic broadcast. Upon receiving an atomic broadcast of the form "k failed", process i executes $\mathit{failed}_i(k)$. By the definition of atomic broadcast, every such message will be received in the same order by all processes; therefore, all failures will be detected in the same order by all processes.

[1] This is not surprising; **qTFS** can be implemented on top of the $\Diamond\mathcal{W}$ failure suspector of [CT93], which is known to be strong enough for solving the equivalent problem of Consensus. (See Section 4.6.2).

5.5 FIFO Fail-Stop

The FIFO fail-stop model assumes that the failure of process i is not detected by process j until j has received all messages sent to it by i. Formally, the FIFO fail-stop properties are:

FFS1: **FS1**

FFS2: **FS1**

FFS3: $\forall r, i, j, m\colon\ r \models \Box[\text{SEND}_i(j, m) \Rightarrow \neg(\text{FAILED}_j(i) \wedge \neg\text{RECV}_j(i, m))]$

This model facilitates the solution of many problems, because detecting a failure implies that the final state of the failed process is known. For instance, the Primary-Backup problem can be solved with no blocking in this model. Furthermore, this model is simulated by several membership services, including Consul and Isis, as discussed in the next section.

5.6 Approximating FIFO Fail-Stop

FIFO fail-stop cannot be approximated in the same way as the other models presented: there is no model that is indistinguishable from FIFO fail-stop, unless communication is allowed to block.

One can approximate FIFO fail-stop by having processes agree on the last message sent by a failed process. Subsequent messages received from a failed process can then be ignored. This method generates runs that are distinguishable from

FIFO fail-stop runs, but it is sufficient for many purposes; indeed, it is provided by some group membership services, such as Isis [BJ87,RB91] and Consul [MPS91a].

Alternatively, one can alter the definition of *send* and *recv* events by requiring processes to acknowledge all messages when they are received, and requiring *send* events to block until the acknowledgement is received by the sender. Under this definition, SEND does not become true until an acknowledgement is received by the sender, instead of when the message is sent. If a process i executes $\mathit{failed}_i(j)$ and then receives a message from j, i can simply fail to send an acknowledgement to j, thereby preventing the sending of the message from being completed. As a result, all messages m for which $\text{SEND}_j(i, m)$ is true must have been received by i before $\mathit{failed}_i(j)$ was executed, thus satisfying **FFS3**. This method can be considered a "simulation" of FIFO fail-stop. However, it has some undesirable properties – for instance, a message is not "sent" until after it is received. Furthermore, requiring an acknowledgement means that executing a *send* event takes an arbitrary amount of time, during which the sending process makes no progress.

Each of the above approximations can be useful. However, neither approximation is sufficient to solve the non-blocking Primary Backup problem.

Chapter 6

Related Work

6.1 Weak Failure Detectors

Chandra and Toueg ([Cha93,CT95]) have examined failure detectors from the point of view of solving the Consensus problem. They define a hierarchy of failure detectors by varying the Accuracy and Completeness properties (see Chapter 2), and show that the weakest failure detector in this hierarchy is the weakest that can be used to solve Consensus.

The weakest failure detector in the failure model hierarchy presented in Chapter 5 is the fail-stop model, which is equivalent to $\mathcal{P}$, the strongest failure detector in the hierarchy of [Cha93]. This does not imply that our hierarchy is an extension of the hierarchy of [Cha93]: the hierarchies are not generally comparable. This is because Chandra and Toueg assume that the suspicion mechanism is highly unreliable, and therefore that suspicions can be repented. However, if the work in this dissertation is extended to allow repentance (e.g., by adding incarnation numbers to processes, as described in Section 4.6.1), then the failure models in the Chandra/Toueg hierarchy can be used in conjunction with our work. One can use the weak failure detectors as specifications of different assumptions about the nature of suspicions, and then ask the question: what is the weakest failure "suspector" that can be used to implement approximate fail-stop? We have done preliminary work towards answering this question, but the results are as yet incomplete. For instance, we know that if repentance is disallowed, $\mathcal{S}$ is the weakest failure detector that can be used to approximate fail-stop if $f > \frac{n}{2}$ (see Theorem 15 in Section 4.5). If weaker suspectors are assumed, then repentance is necessary to avoid the possibility of a total failure.

6.2 Group Membership Systems

Distributed systems providing group membership services have become common in recent years (e.g., [BJ87,ADKM92,MPS91a]). These systems allow processes to form groups and to keep track of the membership of groups in a consistent way. When the group membership changes, due to a process joining, leaving voluntarily, or crashing, all group members must eventually update their views; furthermore, all group members must eventually share the same view. Thus, any solution to GMP requires a failure detector.

Many papers have been written in the last few years that attempt to formalize systems that provide membership services. In [HS93,HS95], the problem of group membership is broken down into the various properties that can be provided by membership services, and several existing systems are characterized according to these properties. Most of the properties deal with the manner in which an application is notified of process failures and recoveries. For example, a system is said to be *accurate* if a failure is only reported to the application when the site has actually failed, and *live* if every failure is eventually detected. The liveness property corresponds to the completeness property used in this dissertation and in [Cha93,CT95]. Also, the property "total ordering of membership messages" corresponds to the serial fail-stop model presented in Section 5.3: all failure (and recovery) notifications must be delivered to all applications in the same order. Implementing this property requires a perfect failure detector. The "agreement on last message" property requires that all applications agree on the last message sent by a failed process, and that this message be delivered before the failure notification. This property can be considered an approximation of the FIFO fail-stop model (see Section 5.6). This property is useful in many systems; however, as noted in Section 5.6, there are problems that require the FIFO fail-stop model for which the agreement on last message property is not sufficient (e.g., non-blocking Primary Backup).

[MBRS94], [Ric95b], and [Ric95c] isolate the problem of *Uniform Coordination* as a basis for most membership services. In this problem, there are certain actions that must either be taken by all correct processes (i.e., "uniformly") or by none at all. Processes may be *exempted* from taking the action if they are suspected of crashing, as long as enough processes remain unexempted. It is shown that this problem is not as hard as Consensus, and can be solved in an asynchronous system as long as there is always a majority of correct processes. In our quasi fail-stop model, failure detections are uniform actions: if a failure is detected, then the failure will eventually occur, and every correct process is required to detect actual failures. The method given in [Ric95b] for solving Uniform Coordination is very similar to our quasi fail-stop model: it is only necessary to coordinate processes that are not exempted, and exempted processes are treated as though they have crashed, so that all runs of the system are indistinguishable from runs in which all exempted processes actually crash. It is not surprising that this work takes a similar approach to ours, since both efforts arose from attempts to understand and formalize the properties guaranteed by the Isis system.

Isis ([BJ87]) provides group membership services in asynchronous distributed systems. It provides several broadcast primitives that impose varying degrees of ordering on messages exchanged between and within groups, including an ABCAST primitive that allows processes to atomically broadcast messages: all messages sent using ABCAST will be delivered at all correct processes in the same total order. Since the Atomic Broadcast problem is equivalent to Consensus ([Mul93,Cha93]), it would appear that Isis violates the impossibility result of [FLP85]. However, a closer examination of Isis reveals that it uses an imperfect failure detector ([RB91,Ric95b]). This failure detector is very similar to the quasi fail-stop model presented in Chapter 4 of this dissertation. In particular, if a process in a group is suspected of failing, that process is *shunned*, even if it has not actually crashed. The remaining processes in the group continue to make decisions; the shunned process is excluded from the group.

Isis provides *virtual synchrony*: all changes to the membership of a group are seen in the same total order by all remaining members of the group. This requires the serial fail-stop model defined in Chapter 5 (Section 5.4). Furthermore, virtual synchrony allows processes to agree on the last message received from a process that has left the group: messages sent by such a process are either received by all other processes before the process leaves the group, or after. This property is equivalent to the approximation of the FIFO fail-stop model presented in Chapter 5 (Section 5.6).

The Consul membership system ([MPS91b]) is based on the Psync multicast mechanism. For efficiency, Psync maintains a partial order of messages rather than a total order. As a result, different processes in a group can see different orderings of a set of failures, though all processes are guaranteed to see the same set of failures eventually. However, Consul does guarantee that processes agree on the last message sent by a failed process. This implies that Consul also approximates a perfect failure detector.

Chandra, Hadzilacos, and Toueg show in [CHT95] that even a very weak specification of the Group Membership problem cannot be solved in an asynchronous system. However, their specification does not allow two processes in a group to hold different views temporarily before agreeing on a new view. Most existing systems do allow a period of transition between membership changes during which processes in a group may hold different views before agreement is reached. Therefore, the result of [CHT95] does not contradict previous work on this topic.

Bibliography

[ACBMT95] Emmanuelle Anceaume, Bernadette Charron-Bost, Pascale Minet, and Sam Toueg. On the formal specifications of group membership services. Technical Report TR95-1534, Department of Computer Science, Cornell University, August 1995.

[ADKM92] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 76–84, July 1992.

[ADKM93] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms for multicast communication groups. In *Proceedings of the IEEE 13th International Conference for Distributed Computing Systems*, pages 551–560, May 1993.

[BJ87] Kenneth Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh Annual ACM Symposium on Operating System Principles*, pages 123–138. ACM, 1987.

[BMST92] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-backup protocols: lower bounds and optimal implementations. In *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications*. IFIP 10.4, September 1992.

[CD89] Benny Chor and Cynthia Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, 5:443–497, 1989.

[Cha93] Tushar Deepak Chandra. *Unreliable Failure Detectors for Asynchronous Distributed Systems*. PhD thesis, Cornell University, 1993.

Available as Technical Report TR93-1377, Department of Computer Science, Cornell University, August 1993.

[CHT92] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*. ACM, August 1992.

[CHT95] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. Impossibility of group membership in asynchronous systems. Technical Report TR95-1533, Department of Computer Science, Cornell University, August 1995.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM86] K. M. Chandy and Jayadev Misra. How processes learn. *Distributed Computing*, 1(1):42–50, 1986.

[Cri88] Flaviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. Technical Report IBM Research Report RJ 5964 (59426), IBM, March 1988.

[CT91] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*. ACM, August 1991.

[CT93] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. Technical Report TR93-1374, Department of Computer Science, Cornell University, August 1993.

[CT94] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. Technical Report TR94-1458, Department of Computer Science, Cornell University, October 1994.

[CT95] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report TR95-1535, Department of Computer Science, Cornell University, 1995.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34:77–97, January 1987.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Symposium on Operating Systems Principles*, pages 150–162. ACM SIGOPS, December 1979.

[HS93] Matti A. Hiltunen and Richard D. Schlichting. Understanding membership. Technical Report TR95-07, Department of Computer Science, University of Arizona, May 1993.

[HS95] Matti A. Hiltunen and Richard D. Schlichting. Properties of membership services. In *Proceedings of the Second International Symposium on Autonomous Decentralized Systems*, April 1995.

[JRF93] Farnam Jahanian, Ragunathan Rajkumar, and Sameh Fakhouri. Group membership protocols: Specification, design and implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, 1993.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[MBRS94] Dalia Malki, Ken Birman, Aleta Ricciardi, and André Schiper. Uniform actions in asynchronous distributed systems. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 274–283. ACM, August 1994.

[MMSA95] Louise E. Moser, P. M. Melliar-Smith, and Vivek Agrawala. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):459–473, May 1995.

[MPS91a] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report TR 91-32, Department of Computer Science, University of Arizona, July 1991.

[MPS91b] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. A membership protocol based on partial order. In *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, February 1991.

[MSMA91] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *Proceedings of the IEEE 11th International Conference for Distributed Computing Systems*, pages 480–488, May 1991.

[Mul93] Sape Mullender, editor. *Distributed Systems*, chapter 5. ACM Press frontier series. Addison-Wesley, second edition, 1993.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium of Foundations of Computer Science*. ACM, November 1977.

[RB91] Aleta Ricciardi and Kenneth Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*. ACM, August 1991.

[Ric95a] Aleta Ricciardi, 1995. Personal communication.

[Ric95b] Aleta Ricciardi. Dissecting distributed coordination. In *Proceedings of the 9th Workshop on Distributed Algorithms (WDAG)*, September 1995.

[Ric95c] Aleta Ricciardi. Sequential distributed coordination. Technical Report TR-PDS-1995-003, Department of Electrical and Computer Engineering, University of Texas at Austin, May 1995.

[Sch84] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[Ske85] Dale Skeen. Determining the last process to fail. *ACM Transactions on Computer Systems*, 3(1):15–30, February 1985.

[SM95] Laura Sabel and Keith Marzullo. Election vs. consensus in asynchronous distributed systems. Technical Report TR95-1488, Department of Computer Science, Cornell University, February 1995.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[SS94] André Schiper and Alain Sandoz. Primary Partition "Virtually-Synchronous Communication" harder than consensus. In *Proceedings of the 8th Workshop on Distributed Algorithms*, 1994.