Simulating Fail-Stop in Asynchronous Distributed Systems*

Laura Sabel**
Keith Marzullo

TR 94-1413
March 1994

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, and by grants from IBM and Siemens. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.
** This author is also supported by an AT&T PhD Scholarship.

Simulating Fail-Stop in Asynchronous Distributed Systems*

Laura S. Sabel[†]
Cornell University
Department of Computer Science

Keith Marzullo
University of California, San Diego
Department of Computer Science

10 March 1994

Abstract

The fail-stop failure model appears frequently in the distributed systems literature. However, in an asynchronous distributed system, the fail-stop model cannot be implemented. In particular, it is impossible to reliably detect crash failures in an asynchronous system.

In this paper, we show that it is possible to specify and implement a failure model that is indistinguishable from the fail-stop model from the point of view of any process within an asynchronous system. We give necessary conditions for a failure model to be indistinguishable from the fail-stop model, and derive lower bounds on the amount of process replication needed to implement such a failure model. We present a simple one-round protocol for implementing one such failure model, which we call *simulated fail-stop*.

1 Introduction

The fail-stop failure model appears frequently in the distributed systems literature. The fail-stop model makes two assumptions about the failure behavior of processes: processes fail only by permanently crashing, and when a process crashes, surviving processes will eventually detect that failure. The fail-stop model is appealing because it makes distributed algorithms easier to formulate: fail-stop failures are easy to tolerate.

For example, suppose that a set of processes $\{1, 2, ..., n\}$ wish to solve the *election problem*: at any point, no more than one process of the set can be the *leader*, and as long as all processes do not fail, it is always the case that there will eventually be a leader. Assuming a fail-stop failure model leads to a very simple solution. Each process maintains a local copy of the list $\langle 1, 2, ..., n\rangle$, and the first element of this list denotes the leader. When process i detects the failure of process j, i removes j froms its local copy of the list. When i finds itself the first element of its list, i knows that it is the leader. Since a process becomes the head of the list only when all lower-numbered processes have failed, there is no more than one leader at any time; and, as long as a process eventually detects the failure of the lower-numbered processes, it will eventually become the leader.

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593, and by grants from IBM and Siemens. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

†This author is also supported by an AT&T PhD Scholarship.

A serious limitation of assuming a fail-stop failure model is that it is often an unrealistic assumption. In particular, in an asynchronous distributed system (i.e., a system with no shared memory, arbitrary message delivery times, no global clock, and arbitrary process speeds), the fail-stop model cannot be implemented. This is because it is impossible to reliably detect crash failures in an asynchronous system (see Theorem 1).

On the other hand, there are systems (e.g., ISIS [BJ87]) that provide crash-failure detection without making synchrony assumptions. They do this by allowing failures to be detected erroneously, e.g., by using timeouts and gossip messages ([RB91]) to attain agreement among a set of processes that a process p has failed even though that process p may not have crashed. Hence, they provide a failure model that resembles fail-stop in some ways but is not strictly fail-stop.

In this paper, we present a failure model, called *simulated fail-stop*, that is internally indistinguishable from fail-stop, meaning that under this model, no process in the system can determine that it is not running in a system in which the fail-stop assumption holds. We give a set of conditions that are necessary in order for any model to be indistinguishable from fail-stop, and we prove that simulated fail-stop is indistinguishable from fail-stop. We give lower bounds on the number of processes needed for a one-round implementation of the simulated fail-stop model to tolerate t failures, and show that these bounds hold for any model that is indistinguishable from fail-stop. Finally, we show that the bounds are tight by giving a protocol that attains them.

The paper is organized as follows. Section 2 describes the system model used throughout the paper, including notation, definitions, and a formal logic used to describe system properties. Section 3 specifies the fail-stop and simulated fail-stop models, introduces the notion of indistinguishability of failure models, and proves that certain conditions are necessary and/or sufficient for a failure model to be indistinguishable from fail-stop. Section 4 gives lower bounds on the number of processes needed to tolerate t failures for one-round failure detection protocols implementing the simulated fail-stop model, and shows that these bounds hold for any model that is indistinguishable from fail-stop. Section 5 shows that these lower bounds are tight by presenting a protocol that meets them. Section 6 concludes the paper and discusses the work that remains to be done on this topic.

2 System Model

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. 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 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 $n + 1$ additional boolean variables that are defined as follows:

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

- $\forall j \in P$: **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 **failed**$_i(j)$ becomes true, it remains true forever. Exactly when **failed**$_i(j)$ becomes true with respect to when **crash**$_j$ becomes true is discussed in this paper.

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 a function that maps global states to global states. An event e applied to a global state Σ yields a new global state Σ' that 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.

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 e 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.

We define events, runs, and predicates formally in Appendix A.1. Informally, send, receive, crash, and failure detection events are defined as follows:

- *send*$_i(j, m)$ denotes the event whereby process i sends the message m to process j.

- *recv*$_i(j, m)$ denotes the event whereby process i receives the message m from process j.

- *crash*$_i$ denotes the event whereby **crash**$_i$ becomes true.

- *failed*$_i(j)$ denotes the event whereby **failed**$_i(j)$ becomes true.

Definition 1 *A* run *of the system is an infinite sequence of global states of the system:* $r = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$, *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* $i \geq 0$, $\Sigma_{i+1} = e_i(\Sigma_i)$.

Definition 2 *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* $i \geq 0$, $\Sigma_{i+1} = e_i(\Sigma_i)$.

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 paper, 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 [Pne77]. We define the boolean predicates $\text{SEND}_i(j, m)$ and $\text{RECV}_i(j, m)$ as follows.

- $\forall i, j, m$: $\text{SEND}_i(j, m)$ and $\text{RECV}_i(j, m)$ are false in an initial global state.
- $send_i(j, m)(\Sigma) \models \text{SEND}_i(j, m)$. That is, $\text{SEND}_i(j, m)$ becomes true when $send_i(j, m)$ has occurred.
- $recv_i(j, m)(\Sigma) \models \text{RECV}_i(j, m)$. That is, $\text{RECV}_i(j, m)$ becomes true when $recv_i(j, m)$ has occurred.

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

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

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

Note that both CRASH_i and $\text{FAILED}_i(j)$ are stable by assumption: once these local variables become true in the local state of i, they remain true thereafter.

Let $s = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$ be a suffix of a run, let φ be a predicate, and let $\mathcal{P}$ be a temporal logic formula.

- $(s, k) \models \varphi$ iff $\Sigma_k \models \varphi$.
- $(s, k) \models \Diamond\mathcal{P}$ iff $\exists j \geq k$: $(s, j) \models \mathcal{P}$
- $(s, k) \models \Box\mathcal{P}$ iff $\forall j \geq k$: $(s, j) \models \mathcal{P}$

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

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

Definition 3 *If* $r \models \Diamond\text{FAILED}_j(i)$ *in 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 yet neither i failed before j nor j failed before i.

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. $e_1 = \mathit{send}_i(j, m)$ for some value of i, j, and m, and $e_2 = \mathit{recv}_j(i, m)$;

3. there exists an event e such that $e_1 \rightarrow e$ and $e \rightarrow e_2$.

The happens-before relation as defined here is the same as that given in [Lam78], except that our relation is reflexive. This is 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.

Let r be a run. Let r_i be the sequence of states of i in 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, r_Q for $Q \subseteq P$ is the sequence of states of processes $i \in Q$ in r with repeated states removed, and $x \equiv_Q y$ if and only if $x_Q = y_Q$. (See [CM86] for a detailed discussion of the ramifications of indistinguishability of runs.)

3 Specification of Failure Models

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* 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.

3.1 The Fail-Stop Failure Model

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 don't crash, and that there are no false detections of failure. These two conditions 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 conditions are:

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

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

We denote with **FS** the set of runs satisfying properties **FS1** and **FS2**.

Theorem 1 *In an asynchronous system in which crash failures are possible, properties* **FS1** *and* **FS2** *are impossible to implement.*

Proof: In [CT91], an algorithm is given for solving Consensus with a Strong Failure Detector. A Strong Failure Detector is shown to be strictly weaker than a Perfect Failure Detector, implying that a Perfect Failure Detector can also be used to solve Consensus. A solution to Consensus contradicts the result of [FLP85]; therefore, a Perfect Failure Detector cannot be constructed.

A Perfect Failure Detector is defined in [CT91] as a failure detector satisfying *Strong Completeness* and *Strong Accuracy*. These two properties are identical to **FS1** and **FS2**. Therefore, implementing **FS** is equivalent to implementing a Perfect Failure Detector, and is therefore impossible. □

3.2 Indistinguishable 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 events in both r and r'. We know that the two runs are isomorphic with respect to i 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 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 in P can determine that r is not in **FS**.

Definition 4 *A failure model M is* indistinguishable *from the fail-stop model if for any run $r \in M$, there exists a run $r' \in$* **FS** *such that $r \equiv_P r'$ (that is, r is indistinguishable from r' to every process in P).*

Consider the election protocol described in Section 1. If a run of this protocol is in a failure model M that is indistinguishable from, but not identical to **FS**, then there may be more than one leader in some global state, but no process will be able to determine this. Thus, internally the execution is the same as if there were only one leader at a time.

Recall that the reason that **FS** can not be implemented in an asynchronous system is because the crash of a process cannot be reliably detected. A failure model M that can be implemented and is indistinguishable from **FS** must be weaker than **FS**. However, it cannot be too weak; at the very least, 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 a process detects the failure of i then i must crash at some point, and process crashes must have been able to occur in some total order. Hence, the following three conditions are necessary for indistinguishability from **FS**.

Condition 1 *For all runs r, if $r \models \Diamond\text{FAILED}_i(j)$, then $r \models \Diamond\text{CRASH}_j$.*

Condition 2 *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 $r \models \text{FAILED}_{x_1}(x_2) \wedge \text{FAILED}_{x_2}(x_3) \wedge \cdots \wedge \text{FAILED}_{x_{k-1}}(x_k) \wedge \text{FAILED}_{x_k}(x_1)$.*

Condition 3 *For all runs r, there cannot be an event e of process j such that $\text{failed}_i(j) \rightarrow e$ in $\mathcal{H}_r$.*

Theorem 2 *If failure model M is indistinguishable from* **FS**, *then all runs of M satisfy Conditions 1–3.*

Proof:

Condition 1 In order for two runs to be isomorphic, their histories must contain the same events. For every run r that satisfies **FS**, $\textit{failed}_i(j) \in \mathcal{H}_r \Rightarrow \textit{crash}_j \in \mathcal{H}_r$. Therefore, the same must be true of every run that satisfies M. □

Condition 2 For contradiction, suppose that there is some run r of M such that r does not satisfy Condition 2. 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 2, then there is some set of processes $\{x_0, x_1, \ldots, x_k\}$ such that $\mathcal{H}_r = (\cdots \textit{failed}_{x_0}(x_1) \cdots \textit{failed}_{x_1}(x_2) \cdots\cdots\cdots \textit{failed}_{x_{k-1}}(x_k) \cdots \textit{failed}_{x_k}(x_0) \cdots)$. For any run r' satisfying **FS**, $\mathcal{H}_{r'}$ must contain $\textit{crash}_{x_i}$ for all $0 \leq i \leq k$. Furthermore, $\textit{crash}_{x_i}$ must occur before $\textit{failed}_{x_{i\ominus 1}}(x_i)$ and $\textit{failed}_{x_i}(x_{i\oplus 1})$ must occur before $\textit{crash}_{x_i}$ where $\ominus$ and $\oplus$ are $-$ and $+$ modulo $k+1$ respectively. By transitivity, this leads to circular constraints on $\mathcal{H}_{r'}$: $\textit{crash}_{x_0}$ must occur before $\textit{failed}_{x_k}(x_0)$, which must occur before $\textit{crash}_{x_k}$, which must occur before $\textit{failed}_{x_{k-1}}(x_k), \ldots, \textit{crash}_{x_1}$ must occur before $\textit{failed}_{x_0}(x_1)$, which must occur before $\textit{crash}_{x_0}$. It is impossible to satisfy all of these ordering constraints in a valid run. Therefore, there is no run r' isomorphic to r that satisfies **FS**. □

Condition 3 For contradiction, suppose that there is some run r of M such that r does not satisfy Condition 3. We will show that there is no run r' satisfying **FS** that is isomorphic to r with respect to P.

If r does not satisfy Condition 3, then $\mathcal{H}_r = (\cdots \textit{failed}_i(j) \cdots \textit{send}_i(k, m_k) \cdots \textit{recv}_j(\ell, m_j) \cdots e_j \cdots)$, where $\textit{send}_i(k, m_k) \rightarrow \textit{recv}_j(\ell, m_j)$. For any r' isomorphic to r, $\mathcal{H}_{r'}$ must maintain the order of $\textit{failed}_i(j)$, $\textit{send}_i(k, m_k)$, and $\textit{recv}_j(\ell, m_j)$ in order to satisfy the happens-before relation. However, for r' to satisfy **FS**, $\textit{crash}_j$ must occur before $\textit{failed}_i(j)$ in $\mathcal{H}_{r'}$. This means that in $\mathcal{H}_{r'}$, $\textit{crash}_j$ must occur before $\textit{recv}_j(\ell, m_j)$, which contradicts the definition of $\textit{crash}_j$. Therefore, there is no run r' isomorphic to r that satisfies **FS**. □

We have shown that Conditions 1, 2, and 3 are necessary for a failure model to be indistinguishable from fail-stop. However, these conditions are not sufficient.

Theorem 3 *There exists a run r that satisfies Conditions 1–3 such that $\neg\exists r' : r' \equiv_P r \wedge r' \in$* **FS**.

Proof: Let r be the following run:

$$\mathit{failed}_y(x); \mathit{send}_y(a, m_a); \mathit{recv}_a(y, m_a); \mathit{crash}_a; \mathit{failed}_b(a); \mathit{send}_b(x, m_x); \mathit{recv}_x(b, m_x); \mathit{crash}_x \cdots$$

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

- $\mathit{failed}_y(x) \to \mathit{send}_y(a, m_a) \to \mathit{recv}_a(y, m_a) \to \mathit{crash}_a$
- $\mathit{failed}_b(a) \to \mathit{send}_b(x, m_x) \to \mathit{recv}_x(b, m_x) \to \mathit{crash}_x$
- crash_x must occur before $\mathit{failed}_y(x)$
- crash_a must occur before $\mathit{failed}_b(a)$

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**. □

Theorem 3 implies that a failure model M that satisfies Conditions 1–3 may not be indistinguishable from **FS**. In the next section, we give a set of conditions that are sufficient, though not all are necessary.

3.3 Simulated Fail-Stop

We give four properties that comprise a model that is indistinguishable from fail-stop. We call this model the *simulated fail-stop* model (**sFS**).

To construct conditions for the **sFS** model, we weaken one of the conditions 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. In a real system, it would be 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. We assume for the remainder of this paper that there is some mechanism provided by the underlying system to implement **FS1**.

We replace **FS2** with the following four conditions:

sFS2a: $\forall r, i, j\colon\ r \models \Box(\text{FAILED}_i(j) \Rightarrow \Diamond\text{CRASH}_j)$

This condition states that if process i detects that process j has crashed, then eventually j will crash even if i's detection was erroneous. In conjunction with **FS1**, this condition implies Condition 1: if $\mathit{failed}_i(j)$ occurs in $\mathcal{H}_r$, then crash_j occurs in $\mathcal{H}_r$.

sFS2b : The failed-before relation is always acyclic.

sFS1:	**FS1**
sFS2a:	$r \models \Box(\text{FAILED}_i(j) \Rightarrow \Diamond\text{CRASH}_j)$
sFS2b:	The failed-before relation is acyclic.
sFS2c:	$r \models \Box\neg\text{FAILED}_i(i)$
sFS2d:	$r \models \Box[\text{FAILED}_i(j) \wedge \neg\text{SEND}_i(k, m) \Rightarrow$ $\Box((\text{SEND}_i(k, m) \wedge \text{RECV}_k(i, m)) \Rightarrow \text{FAILED}_k(j))]$

Figure 1: Simulated Fail-Stop Conditions

This is Condition 2.

sFS2c: $\forall r, i$: $r \models \Box\neg\text{FAILED}_i(i)$

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

sFS2d : $\forall r, i, j, k$: $r \models \Box[\text{FAILED}_i(j) \wedge \neg\text{SEND}_i(k, m) \Rightarrow$
$\Box((\text{SEND}_i(k, m) \wedge \text{RECV}_k(i, m)) \Rightarrow \text{FAILED}_k(j))]$

This condition states that once i detects the failure of j, then any subsequent 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 **sFS2c** and **sFS2d** together imply Condition 3, as shown in the following lemma.

Lemma 4 *If* **sFS2c** *and* **sFS2d** *hold in a run r, then there cannot be an event e of process j such that* failed$_i(j) \rightarrow e$ *in $\mathcal{H}_r$.*

Proof: Consider any run r. If $i = j$, then the lemma is trivially true, because from **sFS2c**, *failed*$_i(i)$ does not appear in $\mathcal{H}_r$. Assume that $i \neq j$. For contradiction, let e be an event of j such that *failed*$_i(j) \rightarrow e$ in $\mathcal{H}_r$. Since *failed*$_i(j)$ and e are of different processes, from the definition of the happens-before relation there is a sequence of events *failed*$_i(j) \rightarrow$ *send*$_i(k_1, m_{k_1}) \rightarrow$ *recv*$_{k_1}(i, m_{k_1}) \rightarrow$ *send*$_{k_1}(k_2, m_{k_2}) \rightarrow \cdots \rightarrow$ *recv*$_j(k_t, m_{k_{t+1}}) \rightarrow e$. From **sFS2d**, 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 **sFS2c**. □

The **sFS** conditions are summarized in Figure 1.

Theorem 5 *The simulated fail-stop model* (**sFS**) *is indistinguishable from the fail-stop model* (**FS**).

The full proof of this theorem is given in Appendix A.2. An outline of the proof is given below.

Consider a run r that satisfies **FS1** and **sFS2a-d** but 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 **sFS2a**, $r \models \Diamond\text{CRASH}_i$. Therefore, $\mathcal{H}_r = (\cdots \textit{failed}_j(i) \cdots \textit{crash}_i \cdots)$. It can be shown that an event e can be moved within $\mathcal{H}_r$, resulting in $\mathcal{H}_{r'}$ such that $r' \equiv_P r$, as long as the happens-before relation is maintained in $\mathcal{H}_{r'}$. We show in Appendix A.2 that $\neg(\textit{failed}_j(i) \rightarrow \textit{crash}_i)$, and that therefore, $\textit{crash}_i$ and all events e between $\textit{failed}_j(i)$ and $\textit{crash}_i$ in $\mathcal{H}_r$ such that $e \rightarrow \textit{crash}_i$ can be moved to precede $\textit{failed}_j(i)$ in $\mathcal{H}_{r'}$. Thus, if r satisfies **sFS2a-d**, then the events in $\mathcal{H}_r$ can be rearranged so that $\textit{crash}_i$ precedes $\textit{failed}_j(i)$ for all i, j in $\mathcal{H}_{r'}$.

4 Lower Bounds

The simulated fail-stop properties (**FS1, sFS2a-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 **sFS**.

A *one-round protocol* for detecting a failure is one in which each process i exchanges one round of messages with other processes before executing $\textit{failed}_i(j)$. Any protocol simpler than a one-round protocol would allow at least one process to unilaterally detect the failure of some other process. Such a protocol, however, would limit which processes another process could detect as faulty. For example, suppose that process i can unilaterally decide that process j has failed. Process i can execute $\textit{failed}_i(j)$ concurrently with any event of process j, and so process j can never execute $\textit{failed}_j(i)$. Hence, we will consider the class of one-round protocols in order to determine message and replication complexity.

We say that a process i initiates a failure detection protocol when it "suspects" the failure of another process j (e.g., due to a timeout at a lower level). In the first half of the round, process i sends a message to all other processes; in the second half of the round, processes send an acknowledgement message to i. We call the first message $\text{SUSP}_{i,j}$ and the acknowledgement message $\text{ACK.SUSP}_{i,j}$. Upon completion of the failure detection protocol, i will execute either $\textit{crash}_i$ or $\textit{failed}_i(j)$ for some $j \neq i$.

A one-round protocol that implements **sFS** must avoid cycles in the failed-before relation since all runs in **sFS** satisfy **sFS2b**. Implementing **sFS2b** requires that in any run there is at least one process that participates in all failure detections. To see why this is so, consider the problem of avoiding cycles involving exactly two processes. Suppose that process a suspects the failure of process b. Before a can execute $\textit{failed}_a(b)$, the failure detection protocol must ensure that $\textit{failed}_b(a)$ has not been executed and that $\textit{failed}_b(a)$ will not be executed in the future.

The failure detection protocol cannot require a to communicate with b directly, because b may have indeed crashed. Therefore, the protocol must require a to receive information from, and distribute information to, other processes. In particular, a must receive information from enough other processes to be sure that $failed_b(a)$ has not been executed, and a must distribute information to enough other processes to be sure that if $failed_a(b)$ is executed, then $failed_b(a)$ will not be executed in the future.

The relevant information that a must disseminate is that a suspects the failure of b. In order for a to know that this information has been received by other processes, it must receive messages from other processes acknowledging that the failure of b is suspected.

Definition 5 *The* quorum set Q_{ij} *of* failed$_i(j)$ *is the set of processes from which i has received acknowledgement messages relating to its suspicion of j's crash. Formally,* $Q_{ij} = \{k \in P : \text{SEND}_i(k, \text{SUSP}_{i,j}) \land \text{RECV}_i(k, \text{ACK.SUSP}_{i,j})\}$.

The set Q_{ab} must be large enough to ensure that b, after hearing from Q_{ba}, will not execute $failed_b(a)$. In particular, the sets Q_{ab} and Q_{ba} must have a non-null intersection.

We call this property the *Witness Property* ($\mathcal{W}$), because the quorum sets for any two failure detections must have at least one process (the *witness*) in common. It can be shown that the same property must hold in order to avoid cycles of any size. The Witness Property can be stated formally as follows:

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

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.

Theorem 6 $(\forall r\colon\ r \models \Box\mathbf{sFS2b}) \Rightarrow (\forall r\colon\ r \models \Box\mathcal{W})$.

It was argued above that $(r \models \Box\mathbf{sFS2b}) \Rightarrow (r \models \Box\mathcal{W})$ if only cycles of size two are possible. The full proof of the theorem is given in Appendix A.3.

Since **sFS2b** (Condition 2) is necessary for indistinguishability from **FS** (see Section 3.2), Theorem 6 implies that $\mathcal{W}$ is necessary for any one-round protocol that implements a failure model indistinguishable from **FS**. Let t be the maximum number of crashes in any run, including those that arise from erroneous suspicions. The necessity of the Witness Property places a constraint on t as a function of n and on the number of messages that a process must wait for before detecting a failure.

The simplest way to ensure that $\mathcal{W}$ holds in a one-round 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 there is always at least one process that never fails, nor is suspected of failing, then this process will be a witness to every failure detection that is executed. This implementation only requires that $t < n$. However, if n is large and t 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 7 *If the size of the quorum set is a fixed and equal size for each failure detection, then to guarantee that* $r \models \Box\mathcal{W}$ *when* t *failures are possible, the size of each quorum set must be strictly greater than* $n(\frac{t-1}{t})$.

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

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

$$
\begin{aligned}
x = n - y \quad &\Rightarrow \quad x > n - \lceil \frac{n}{t} \rceil \\
&\Rightarrow \quad x > \lfloor \frac{nt - n}{t} \rfloor \\
&\Rightarrow \quad x > \lfloor \frac{n(t-1)}{t} \rfloor
\end{aligned}
$$

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

Corollary 8 *If the minimum quorum size is used in a one-round protocol for failure detection, then it must be the case that* $n \geq t^2$.

Proof: In a one-round protocol, the size of the quorum is equal to the number of ACK.SUSP$_{i,j}$ messages that process i must receive before executing *failed*$_i(j)$. Since i is in its own quorum, i must wait for $\lfloor n(\frac{t-1}{t}) \rfloor$ messages before detecting j's failure. In order for the one-round protocol to make progress, at least this many other processes must remain alive. Therefore, we have

$$
\begin{aligned}
n - t \geq \lfloor n(\frac{t-1}{t}) \rfloor \quad &\Rightarrow \quad n - t \geq \lfloor n - \frac{n}{t} \rfloor \geq n - \lfloor \frac{n}{t} \rfloor \\
&\Rightarrow \quad t \leq \lfloor \frac{n}{t} \rfloor \\
&\Rightarrow \quad t^2 \leq t \lfloor \frac{n}{t} \rfloor \leq n \\
&\Rightarrow \quad t^2 \leq n
\end{aligned}
$$

□

5 Upper Bounds

We give a simple one-round protocol that implements **sFS2a-d**. We assume that a failure can be suspected spontaneously (e.g., due to a timeout), but that no more than t failures are suspected in any run. In this protocol, $\text{SUSP}_{i,j} = \text{ACK.SUSP}_{i,j} =$ "j failed".

- When process i suspects the failure of process j, i sends the message "j failed" to all processes (including itself). Process i waits for messages of the form "j failed" from other processes and takes no other action except for acknowledging "x failed" messages until it completes the protocol or crashes.
- When process i has received messages of the form "j failed" from more than $n(\frac{t-1}{t})$ processes (including itself), i executes $\mathit{failed}_i(j)$.
- When process x receives a message of the form "x failed", x executes crash_x.
- When process x receives a message of the form "y failed", x suspects the failure of y.

We will argue informally that this protocol implements the simulated fail-stop properties.

sFS2a: Process i cannot execute $\mathit{failed}_i(j)$ without sending a message of the form "j failed" to all other processes, including j. Since channels are nonfaulty, j will eventually receive such a message, upon which j will crash.

sFS2b: The full proof is given in Appendix A.4. We give an outline of the proof for cycles of length 2. Suppose that the protocol generates a run r such that $r \models \Diamond(\text{FAILED}_i(j) \wedge \text{FAILED}_j(i))$. By Theorem 7, $r \models \Box \mathcal{W}$ holds. Therefore, there is some witness w such that i received "j failed" from w and j received "i failed" from w. Process w sends these messages to all processes. If w sends "j failed" before it sends "i failed", then process j will receive "j failed" and crash before it can execute $\mathit{failed}_j(i)$. Similarly, if w sends "i failed" before it sends "j failed", then process i will receive "i failed" and crash before it can execute $\mathit{failed}_i(j)$. Therefore, it is not possible for both $\mathit{failed}_i(j)$ and $\mathit{failed}_j(i)$ to be executed in a run.

sFS2c: Process i cannot execute $\mathit{failed}_i(i)$ without receiving at least one message of the form "i failed". Upon receiving such a message, i crashes. Therefore, $\mathit{failed}_i(i)$ is never executed.

sFS2d: Since channels are FIFO, any message m sent by i to k after $\mathit{failed}_i(j)$ is executed must be received after the message "j failed". Upon receiving "j failed" 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 $\mathit{failed}_k(j)$ is executed. Therefore, message m is not received by k unless $\mathit{failed}_k(j)$ has been executed.

6 Discussion

In Section 3.2, we showed that Conditions 1, 2, and 3 are necessary for any failure model to be indistinguishable from the fail-stop model. In Section 4, we showed that the Witness Property is necessary for any one-round protocol implementing Condition 2. We then showed that the Witness Property imposes lower bounds on the number of messages that must be received before a failure can be detected and on the number of failures that can be tolerated in a system.

We gave a protocol in Section 5 to demonstrate that these bounds are tight. This protocol, however, was derived from conditions that are not necessary for indistinguishability. There may be a failure model weaker than **sFS** that is indistinguishable from **FS**. However, such a failure model is subject to the same bounds on t as **sFS**, and so we do not expect such a failure model to be substantially more interesting than **sFS**.

The bounds on t arise from **sFS2b**. A failure model satisfying only the other **sFS** assumptions would not require a process to wait for any messages before detecting a failure: the other **sFS** properties can be implemented simply by having process i broadcast a message "j failed" after suspecting j's failure and before unilaterally executing $failed_i(j)$. Such a failure model would, of course, be distinguishable from **FS**, but if a collection of processes are insensitive to cyclic failures, then they could be run in this cheaper simulated failure model. We do not know of any protocols in the literature that are insensitive to cyclic failure detection, however.

As an example of sensitivity to **sFS2b**, consider the problem of determining the last process to fail ([Ske85]). Solving this problem requires that processes record information about the failures that they detect (that is, their view of the failed-before relation). Then, when processes are recovering after a total failure, the recovering processes can determine when the last processes to fail have recovered. 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 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.

There are other protocols that require failure models even stronger than **FS**. For example, if the failed-before relation is transitive as well as acyclic, then detecting the last process to fail can be implemented so that as soon as the last processes to fail have recovered, then the processes can determine this. If the failed-before relation is not transitive, then it is necessary to wait for more processes to recover. The failed-before relation of **sFS** is not transitive. We are currently looking into several stronger versions of fail-stop, whether they are implementable given fail-stop, and into how they too can be simulated.

The protocols described in this paper are very simple and are easily implementable. Failure detection such as described here is typically done as part of a group membership service (e.g., [RB91,MPS91,ADKM92]). We believe that the protocols here could be used as the basis of a failure detector that could be used outside of a system built using a group-membership protocol. This would allow for consistent failure detection on top of any kind of lower-level communication, including point-to-point communication.

References

[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.

[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.

[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.

[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.

[CT91] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. Technical Report TR91-1225, Department of Computer Science, Cornell University, August 1991.

[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.

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

[MPS91] 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.

[Pne77] A. Pneuli. 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.

[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.

A Appendices

A.1 Formal Definition of Events, Runs, and Special Predicates

Recall that an *event* e is a function that maps global states to global states. An event e applied to a global state Σ either

- yields Σ, in which case we say that e is a *null* event; or
- yields a new global state Σ' that 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.

A non-null event e is uniquely defined by the process i whose state it changes, the state s of i immediately before e is applied, the state s' of i resulting from e, the states of the channels incident on i before e is applied, and the states of the channels incident on i after e is applied. Let $\mathcal{X}_{i,j}$ be the state of channel $C_{i,j}$. 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 $\langle \mathcal{X}_{1,i}, \mathcal{X}_{2,i}, \ldots, \mathcal{X}_{n,i} \rangle$.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.

If e is a null event, then e is not of any process i and therefore is not represented by a 7-tuple.

Definition 6 *We say that (non-null)* $e = \langle i, s, s', \mathcal{X}_{i,*}, \mathcal{X}'_{i,*}, \mathcal{X}_{*,i}, \mathcal{X}'_{*,i} \rangle$ 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 $\mathcal{X}_{*,i}$.*

A null event can occur in any state.

Let $e = \langle i, s, s', \mathcal{X}_{i,*}, \mathcal{X}'_{i,*}, \mathcal{X}_{*,i}, \mathcal{X}'_{*,i} \rangle$. We abbreviate send and receive events as follows.

- If e is a send event of i and $C'_{i,j} = (C_{i,j} :: m)$ for some j, then e is denoted $send_i(j, m)$.
- If e is a receive event of i and $(m :: C'_{j,i}) = C_{j,i}$ for some j, then e is denoted $recv_i(j, m)$.

We define "crash" events and "failure detection" events as follows:

- If $\mathbf{crash}_i$ is false in s and true in s', then e is denoted $crash_i$. By assumption, $crash_i$ changes only the local variable $\mathbf{crash}_i$.

- If $\exists j$: $\mathbf{failed}_i(j)$ is false in s and true in s', then e is denoted $failed_i(j)$.

The events $send_i(j, m)$, $recv_i(j, m)$, $crash_i$, and $failed_i(j)$ are atomic; each event only changes the relevant state variables of the process on which it occurs. 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 7 *Let $r = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$ be an infinite sequence of global states of the system. We say that r is a run of the system if and only if 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 can occur in Σ_i and $\Sigma_{i+1} = e_i(\Sigma_i)$.*

- $\forall i, j, m$: $\mathrm{SEND}_i(j, m)$ and $\mathrm{RECV}_i(j, m)$ 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 $send_i(j, m)(\Sigma) \models \mathrm{SEND}_i(j, m)$. That is, $\mathrm{SEND}_i(j, m)$ becomes true when $send_i(j, m)$ has occurred.

- Let $e = recv_i(j, m)$ and let Σ be a global state such that e can occur in Σ. Then $recv_i(j, m)(\Sigma) \models \mathrm{RECV}_i(j, m)$. That is, $\mathrm{RECV}_i(j, m)$ becomes true when $recv_i(j, m)$ has occurred.

A.2 Proof of Theorem 5

Theorem 5 *The simulated fail-stop model* (**sFS**) *is indistinguishable from the fail-stop model* (**FS**).

In order to prove that for any run r that satisfies **FS1** and **sFS2a-d**, there is an isomorphic run r' that satisfies **FS1** and **FS2**, we will need to 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 $r \equiv_P r'$.

Consider $\mathcal{H}_r = (\ldots e_i, e_{i+1}, e_{i+2} \ldots)$ corresponding to run $r = (\ldots, \Sigma_i, \Sigma_{i+1}, \Sigma_{i+2}, \ldots)$. By definition, e_i can occur in Σ_i and e_{i+1} can occur in $e_i(\Sigma_i) = \Sigma_{i+1}$. Assume that e_i and e_{i+1} are non-null events.

Suppose that e_i and e_{i+1} are of the same process k. Since e_i changes the state of k, the state of k is not the same in Σ_i as in Σ_{i+1}. Therefore, e_{i+1} cannot occur in Σ_i.

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

In summary, e_{i+1} cannot occur in Σ_i if and only if

- e_i and e_{i+1} are of the same process, or

- $e_i = send_k(\ell, m)$ and $e_{i+1} = recv_\ell(k, m)$.

In other words, e_{i+1} cannot occur in Σ_i if and only if $(e_i \rightarrow e_{i+1})$.

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

Consider $r_{\{k,\ell\}}$ and $r'_{\{k,\ell\}}$. (Recall that repeated states are removed in these sequences.) From the construction of r', $r_k = r'_k$ and $r_\ell = r'_\ell$. Since e_i and e_{i+1} do not change the states of processes other than k and ℓ, $r_t = r'_t$ for all process $t \notin \{k, \ell\}$. Therefore, $r \equiv_P r'$.

In summary, we have shown that if $\neg(e_i \rightarrow e_{i+1})$ in $\mathcal{H}_r$, then e_{i+1} can be moved before e_i to yield $\mathcal{H}_{r'}$ such that $r' \equiv_P r$. It can also be shown that for any two events e_i and e_j in $\mathcal{H}_r$ such that $i < j$ and $\neg(e_i \rightarrow e_j)$, e_j can occur in Σ_i, e_i can occur in $e_j(\Sigma_i)$, and $e_j(e_i(\Sigma_i)) = e_i(e_j(\Sigma_i))$. Therefore, e_j can be moved to directly before e_i to yield $\mathcal{H}_{r'}$ such that $r \equiv_P r'$.

We can now prove the theorem.

Proof: If run 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 **sFS2a**, $r \models \Diamond\text{CRASH}_i$. Therefore, $\mathcal{H}_r$ is of the form $(\cdots \textit{failed}_j(i) \cdots \textit{crash}_i \cdots)$.

Definition 8 *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 $\mathcal{H}_r = (x; \textit{failed}_j(i); y; \textit{crash}_i; z)$ 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 $\mathcal{H}_{r'} = (x'; \textit{crash}_i; \textit{failed}_j(i); y'; z)$ where x' and y' are sequences of events.

> **Base Case (Inner Induction)** Assume $k = 0$. $\mathcal{H}_r = (x; \textit{failed}_j(i); \textit{crash}_i; z)$. Since $\textit{crash}_i$ and $\textit{failed}_i(j)$ are of different processes, they can be swapped to yield $\mathcal{H}_{r'} = (x; \textit{crash}_i; \textit{failed}_j(i); z)$ 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 $k = \ell$. $\mathcal{H}_r = (x; \textit{failed}_j(i); e_1; e_2; \cdots; e_l; \textit{crash}_i; z)$. By Lemma 4 we know that $\neg(\textit{failed}_j(i) \rightarrow \textit{crash}_i)$. Let e_u be the first event of $(e_1; \cdots; \textit{crash}_i)$ such that $\neg(\textit{failed}_j(i) \rightarrow e_u)$. 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 $\textit{failed}_j(i), e_1, \ldots, e_{u-1}$ are events of processes in Q. Then e_u is an event of a process in $\overline{Q}$. Therefore, there is a history $\mathcal{H}_{r''} = (x; e_u; \textit{failed}_j(i); e_1; e_2; \cdots; e_{u-1}; e_{u+1}; \cdots; e_l; \textit{crash}_i)$ 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 $r' \equiv_P r$. $\square_{\textit{Inner Induction}}$

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 $\mathcal{H}_{r'}$, such that $r' \equiv_P r$, with strictly fewer bad pairs, 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 to (x, y).

1. $\cdots\ \mathit{crash}_a\ \cdots\ \mathit{failed}_b(a)\ \cdots\ \underline{\mathit{failed}_y(x)}\ \cdots\ \underline{\mathit{crash}_x}\ \cdots$
2. $\cdots\ \mathit{failed}_b(a)\ \cdots\ \mathit{crash}_a\ \cdots\ \underline{\mathit{failed}_y(x)}\ \cdots\ \underline{\mathit{crash}_x}\ \cdots$
3. $\cdots\ \underline{\mathit{failed}_y(x)}\ \cdots\ \underline{\mathit{crash}_x}\ \cdots\ \mathit{crash}_a\ \cdots\ \mathit{failed}_b(a)\ \cdots$
4. $\cdots\ \underline{\mathit{failed}_y(x)}\ \cdots\ \underline{\mathit{crash}_x}\ \cdots\ \mathit{failed}_b(a)\ \cdots\ \mathit{crash}_a\ \cdots$
5. $\cdots\ \mathit{failed}_b(a)\ \cdots\ \underline{\mathit{failed}_y(x)}\ \cdots\ \underline{\mathit{crash}_x}\ \cdots\ \mathit{crash}_a\ \cdots$
6. $\cdots\ \mathit{crash}_a\ \cdots\ \underline{\mathit{failed}_y(x)}\ \cdots\ \underline{\mathit{crash}_x}\ \cdots\ \mathit{failed}_b(a)\ \cdots$

   Since only events that occur between $\mathit{failed}_y(x)$ and crash_x are moved, (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. $\cdots\ \mathit{failed}_b(a)\ \cdots\ \underline{\mathit{failed}_y(x)}\ \cdots\ \mathit{crash}_a\ \cdots\ \underline{\mathit{crash}_x}\ \cdots$

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

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

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

   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.

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

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

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

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 $\neg(\mathit{failed}_y(x) \rightarrow e)$. 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 $\mathcal{H}_{r'}$

$$\neg(\mathit{failed}_y(x) \rightarrow \mathit{failed}_b(a)) \wedge (\mathit{failed}_y(x) \rightarrow \mathit{crash}_a) \tag{1}$$

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: 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 $\mathcal{H}_{r''}$

$$\neg(\mathit{failed}_b(a) \rightarrow \mathit{failed}_y(x)) \wedge (failed_b(a) \rightarrow \mathit{crash}_x) \tag{2}$$

$(\mathit{failed}_y(x) \rightarrow \mathit{crash}_a)$ in $\mathcal{H}_{r'}$ implies that $\mathit{failed}_a(x)$ occurs in $\mathcal{H}_{r'}$ by **sFS2d** and the definition of happens-before. Similarly, $(\mathit{failed}_b(a) \rightarrow \mathit{crash}_x)$ implies that $\mathit{failed}_x(a)$ occurs in $\mathcal{H}_{r'}$. Thus, Equations 1 and 2 imply that in $\mathcal{H}_{r'}$ both $\mathit{failed}_a(x)$ and $\mathit{failed}_x(a)$ occur in $\mathcal{H}_{r'}$, which contradicts **sFS2b**. Therefore, $\mathcal{H}_{r''}$ cannot have the assumed form, so both (a, b) and (x, y) must be good in $\mathcal{H}_{r''}$.

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, $r \equiv_P r''$.

□

A.3 Proof of Theorem 6

Theorem 6 $(\forall r:\ r \models \Box\mathbf{sFS2b}) \Rightarrow (\forall r:\ r \models \Box\mathcal{W})$.

We will show that $(\exists r:\ r \models \Diamond\neg\mathcal{W}) \Rightarrow (\exists r:\ r \models \neg\Diamond\mathbf{sFS2b})$. 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 $0 \le i \le k-1$, $i \in S_i$; that is, processes 0 through $k-1$ are in sets S_0 through S_{k-1}, and the rest of the processes are distributed among S_0 through S_{k-1}.

Consider the following scenario. For all $i : 0 \le i \le (k-1)$:

1. Process i suspects the failure of process $i \oplus 1$, and sends the message $\text{SUSP}_{i,i\oplus 1}$ 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 $\text{SUSP}_{j\ominus 1,j}$ from process $j \ominus 1$ for all $j \ne i, 0 \le j \le k-1$, where $\ominus$ is subtraction modulo k. Thus, process i does not learn that another process has suspected it of having crashed.

3. Before receiving $\text{SUSP}_{j\ominus 1,j}$, process i suspects the failure of process j, and sends $\text{SUSP}_{i,j}$ 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). Process i also acknowledges any SUSP messages with ACK.SUSP messages.

4. Process i has now received $\text{ACK.SUSP}_{k,i\oplus 1}$ messages from all processes k in $\bigcup_{j \ne i\ominus 1} S_j$.

Let $Q_{i,i\oplus 1} = \bigcup_{j \ne i\ominus 1} S_j$ for all $i : 0 \le i \le k-1$. No process in S_j is in $Q_{j,j\oplus 1}$; 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} being a quorum, every process i has received enough ACK.SUSP messages to execute $failed_i(i \oplus 1)$. We have $failed_0(1), \ldots, failed_{(k-2)}(k-1)$, and $failed_{(k-1)}(0)$, which causes a k-cycle in the failed-before relation. □

A.4 Proof that the Protocol of Section 5 Implements sFS2b

Lemma 9 *Given the protocol of Section 5, then* $[r \models \exists S = \{1, 2, \ldots, k\}\colon (\text{FAILED}_1(2) \wedge \text{FAILED}_2(3) \wedge \cdots \wedge \text{FAILED}_{k-1}(k))] \Rightarrow [\exists q\colon (\text{send}_q(S, \textit{"k failed"}) \rightarrow \text{send}_q(S, \textit{"k − 1 failed"}) \rightarrow \cdots \rightarrow \text{send}_q(S, \textit{"2 failed"}))$ *in* $\mathcal{H}_r]$.

Proof: We use the notation $\text{SEND}_i(S, m)$ as shorthand for $(\forall p \in S\colon \text{SEND}_i(p, m))$.

The size of the quora are sufficient to ensure $\mathcal{W}$, by Theorem 7. By $\mathcal{W}$, $r \models \exists q: \forall i, j \in S: \text{FAILED}_i(j) \Rightarrow \text{RECV}_i(q, \text{"}j\text{ failed"}) \Rightarrow \text{SEND}_q(S, \text{"}j\text{ failed"})$. 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 $r \models \text{SEND}_q(S, \text{"2 failed"}) \wedge \text{SEND}_q(S, \text{"3 failed"})$. Assume for contradiction that $send_q(S, \text{"2 failed"}) \rightarrow send_q(S, \text{"3 failed"})$ in $\mathcal{H}_r$. Then, because channels are FIFO, $recv_2(q, \text{"2 failed"}) \rightarrow recv_2(q, \text{"3 failed"})$ 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 $send_q(S, \text{"3 failed"}) \rightarrow send_q(S, \text{"2 failed"})$.

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, $send_q(S, \text{"}l-1\text{ failed"}) \rightarrow \cdots \rightarrow send_q(S, \text{"2 failed"})$ in $\mathcal{H}_r$. Assume for contradiction that $send_q(S, \text{"}l-1\text{ failed"}) \rightarrow send_q(S, \text{"}l\text{ failed"})$ in $\mathcal{H}_r$. Then, as in the base case, $recv_{l-1}(q, \text{"}l-1\text{ failed"}) \rightarrow recv_{l-1}(q, \text{"}l\text{ failed"})$, so $crash_{l-1} \rightarrow failed_{l-1}(l)$ in $\mathcal{H}_r$ and $r \models \neg\text{FAILED}_{l-1}(l)$. Therefore, $send_q(S, \text{"}l\text{ failed"}) \rightarrow send_q(S, \text{"}l-1\text{ failed"})$ in $\mathcal{H}_r$. □

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 $r \models \exists S = \{1, \ldots, k\}\colon \text{FAILED}_1(2) \wedge \cdots \wedge \text{FAILED}_{k-1}(k) \wedge \text{FAILED}_k(1)$. By Lemma 9, $\exists q\colon send_q(S, \text{"1 failed"}) \rightarrow send_q(S, \text{"}k\text{ failed"}) \rightarrow \cdots \rightarrow send_q(S, \text{"2 failed"})$ in $\mathcal{H}_r$. Thus, $recv_1(q, \text{"1 failed"}) \rightarrow recv_1(q, \text{"2 failed"})$ in $\mathcal{H}_r$, $crash_1 \rightarrow failed_1(2)$ in $\mathcal{H}_r$, and $r \models \neg\text{FAILED}_1(2)$. □