Election Vs. Consensus in Asynchronous Systems*

Laura S. Sabel†
Cornell University
Department of Computer Science

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

Abstract

It was shown in 1985 that the *Consensus problem* cannot be solved in an asynchronous system if even a single crash failure can occur. In this paper, 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 these problems that are harder than Consensus.

1 Introduction

It was shown in 1985 that the *Consensus problem* cannot be solved in an asynchronous system if even a single crash failure can occur ([6]). 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, the Consensus problem has frequently been used as a yardstick of computability in asynchronous fault-tolerant distributed systems.

In this paper, 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 these problems that are harder than Consensus. After presenting our model for distributed computation, 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 other problems that, like Election, are harder than Consensus.

*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.

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. A *correct* process is one that does not crash. 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 channels. Processes communicate by sending and receiving messages over these channels: there is no shared memory. The channels are nonfaulty: they do not lose, generate, or garble messages. Message delivery time is unbounded. The channels need not be FIFO. The state of a channel is the set of messages that have been sent along the channel but not yet received.

A process has 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. 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 set.

An *event* e is an action that maps 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.

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

We specify properties of systems using predicate logic over global states and linear-time temporal logic over (infinite) suffixes of runs ([11]).

We use the following as the meaning of the two temporal logic modal operators $\Diamond$ and $\Box$.

Definition 2 *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. Then,*

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

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

We refer to the pair (s, k) as the *prefix* of the infinite sequence of states s.

We use the following definition of a protocol:

Definition 3 *A* protocol *is a many-to-many mapping from a prefix of a run to a global state.*

Thus, repeatedly applying a protocol to an initial global state will generate a run. This represents the execution of a (possibly nondeterministic) program over time. We apply this procedure of executing a protocol A to extend a prefix (s, k) to $(s', k' > k)$, meaning that $(s, k) = (s', k)$ and that for each state $\Sigma'_\ell : k < \ell \leq k' : \Sigma'_\ell \in A((s', \ell - 1))$.

Two protocols A and B can be combined to generate a single run as follows: given a prefix of a run, one of the protocols is chosen nondeterministically and fairly to generate the next global state. We use the notation $A + B$ to indicate the combination of protocols A and B.

We use the following characterization of knowledge introduced in [5]:

Definition 4 *Two prefixes* (r, k) *and* (r', k') *are* indistinguishable *to a process* i *if the sequences of states attained by process* i *are the same in both prefixes.*

Definition 5 *A process* i knows *a predicate* Φ *(written* $K_i\Phi$*) in* (r, k) *if, for all prefixes* (r', k') *indistinguishable by* i *from* (r, k), $(r', k') \models \Phi$.

Definition 6 *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* $e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_m$ *where* $\rightarrow$ *denotes the* happens before *relation of [7].*

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'.[1]

3 Failures and Failure Detectors

We define the following events associated with crashing, detecting failures, and ceasing to detect failures:

- $crash_i$ denotes the event whereby process i crashes.
- $failed_i(j)$ denotes the event whereby process i detects the failure of process j (or "suspects" j).
- $repent_i(j)$ denotes the event whereby process i stops suspecting process j ("repents" the suspicion of j). This event is executed when i receives a message from j while i suspects j.

We define the boolean predicates CRASH_i and $\text{FAILED}_i(j)$ as follows:

- $\forall i, j$: CRASH_i and $\text{FAILED}_i(j)$ are false in an initial global state.
- CRASH_i is true in the global state resulting from $crash_i$ and in every global state thereafter.
- $\text{FAILED}_i(j)$ is true in the global state resulting from $failed_i(j)$ and in every global state until $repent_i(j)$ is executed.

CRASH_i is stable by definition but $\text{FAILED}_i(j)$ is not.

We assume that crashes can occur spontaneously, and that there is no restriction on the set of processes that may crash at any time in any run. Hence, if $(r, k) \models \neg\text{CRASH}_i$ then the prefix (r, k) can be extended to $(r', k+1)$ such that $(r', k+1) \models \text{CRASH}_i$.

We define a *failure detector* as a set of runs satisfying properties that relate *crash* events to *failed* events. A *failure detection protocol* is a protocol that generates the runs in a failure detector. We allow failure detectors to be inaccurate, in that $failed_i(j)$ can be executed in a state in which CRASH_j is false. This is called *false suspicion*. If the failure detection protocol allows false suspicions to occur, then a false suspicion can occur at any point in a run. Obviously, a suspicion that is not false can only be executed after the failure has occurred.

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

[1] 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.

in which the failure of process i is never detected. In this paper, we consider such failure detectors unrealistic and so consider only failure detectors that satisfy the following conditions. The first condition says that if any process can detect failures erroneously, then all processes can detect failures erroneously.

Symmetry Condition 1: If there exists a run r and processes i, j such that i falsely suspects j in r, then for any prefix of a run (r', k) and processes i', j' such that $(r', k) \models \neg\text{CRASH}_{i'} \wedge (\neg\text{FAILED}_{j'}(i'))$, (r', k) can be extended into a run in which j' executes $\mathit{failed}'_j(i')$.

The second condition says that if some process can fail to suspect a process that has crashed, then any process can fail to suspect any process that has crashed.

Symmetry Condition 2: If there exists a run r and processes i, j such that i crashes yet j never executes $\mathit{failed}_j(i)$ in r, then for any prefix of a run (r', k) and processes i', j' and point k such that $(r', k) \models \text{CRASH}_{i'} \wedge (\neg\text{FAILED}_{j'}(i'))$, (r', k) can be extended into a run in which j' never executes $\mathit{failed}'_j(i')$.

In [3], Chandra and Toueg define the following four properties that relate *crash* and *failed* events.

Strong Completeness: $\forall r: \ r \models \forall i: \ \Box(\text{CRASH}_i \Rightarrow \forall j: \ \Diamond(\text{CRASH}_j \vee \Box\text{FAILED}_j(i)))$ (Eventually every process that crashes is permanently suspected by every correct process).

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

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

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

Using these four properties, among others, Chandra and Toueg define a hierarchy of failure detectors, including the ones defined above, that are sufficient to solve Consensus (see Section 4). For instance, a *Perfect Failure Detector* is the set of runs that satisfy both Strong Completeness and Strong Accuracy and a *Strong Failure Detector* satisfies Strong Completeness and Weak Accuracy.

4 The Consensus Problem

The Consensus problem requires that all correct processes propose a value, and then reach an agreement on some value. In [6], it was shown that the Consensus problem 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 in [6] 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.

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.

The proof of the impossibility of Consensus in [6] assumes that failures cannot be detected accurately. In [3], Chandra and Toueg show that even stronger versions of Consensus can be solved with a failure detector that satisfies Weak Accuracy and eventually satisfies Weak Completeness.

5 The Election Problem

The proof of the impossibility of Consensus in [6] assumes that it is impossible for a process to determine whether another process has crashed, or is just very slow. This assumption is widely cited as the "reason" for the impossibility result. There are other problems that cannot be solved in asynchronous systems with crash failures for the same intuitive reason that Consensus cannot be solved. Some of these problems can be solved with a weak failure detector; however, some cannot. In particular, the Election problem cannot be solved if a crashed process cannot be distinguished from a slow process.

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

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

An *election protocol* is a protocol that generates runs that satisfy the Election specification.

6 Election Is Harder Than Consensus

Though a Strong Failure Detector ([3]; see Section 3) is sufficient to solve Consensus, it is not sufficient to solve Election. Therefore, the Election problem is strictly harder than Consensus. In this section, we define a property called *Very Weak Completeness* that is weaker than Weak Completeness. We show that both Very Weak Completeness and Strong Accuracy are necessary for solving Election, and their conjunction is sufficient for solving Election.

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 every process that crashes is suspected (at least once) by some correct process).

Theorem 2 *Both Strong Accuracy and Very Weak Completeness are necessary to solve Election.*

Proof:

Necessity of Very Weak Completeness: 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 deterministic election protocol E with F such that $E + F$ is an election protocol.

By assumption, there is a run r generated by F such that $r \models (\exists i:\ \ \Box\neg\text{CRASH}_i) \wedge (\exists i: [\Diamond\text{CRASH}_i \wedge \forall j:\ \ (\Diamond\text{CRASH}_j \vee \Box\neg\text{FAILED}_j(i))])$ (there is a correct process, and some process i crashes and is never suspected by any correct process).

Assume for contradiction that there exists a deterministic election protocol E that can be combined with F such that $E + F$ is also an election protocol. Let r be a run of this protocol such that (r, k) is the first prefix in which some process (say, i) is the leader, and another process j has not crashed.

By the assumption that failures are random, we can extend the prefix (r, k) to $(r, k + 1)$ by crashing all processes except j, so that j is the only correct process.

By Symmetry Condition 2 and the assumption that some process is never suspected by a correct process in some run, it is possible to extend the prefix $(r, k+1)$ into a run in which j never detects the failure of i. By the requirements of Election, there must be a $k' > k+1$ such that j is the leader in (r, k').

Let Φ be the predicate "j is not the leader". Since $(r, k) \models K_i K_j \Phi$ and $(r, k') \models \neg K_j \Phi$, there must be a process chain $\langle i, j \rangle$ between k and k' in r.[2] This is impossible because i sent no messages between k and k'. Therefore, there is no k' in which j is the leader, which violates the assumption that $E + F$ is an election protocol.

Necessity of Strong Accuracy: 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 deterministic election protocol E with F such that $E + F$ is an election protocol.

By assumption, there is a run generated by F such that $\exists i, j : \ \Diamond(\text{FAILED}_i(j) \wedge \neg\text{CRASH}_j)$ (some process is suspected before it crashes).

Assume for contradiction that there exists a deterministic election protocol E that can be combined with F such that $E + F$ is also an election protocol. Let r be a run of this protocol such that all processes except i and j have crashed in $(r, 1)$, (r, k) is the first prefix in which some process (say, i) is the leader, and the other process (j) has not crashed in (r, k).

Extend (r, k) to $(r_1, k+1)$ by crashing i. Extend $(r_1, k+1)$ to $(r_1, k+2)$ by having process j execute $\mathit{failed}_j(i)$. By the requirements of Election, there must be a point $(r_1, k_1) : k_1 > k+2$ at which j is the leader.

Extend (r, k) to $(r_2, k+1)$ by crashing j. Let k_2 be the first point greater than $k+1$ such that i is the leader at (r_2, k_2).

In run r_1, process j does not receive any messages sent after point k, and in run r_2, process i does not receive any messages sent after point k. By Symmetry Condition 1 and the assumption that some process falsely suspects another in some run, it is possible to extend the prefix (r, k) into a run in which j falsely suspects i. Thus, we can extend (r, k) to (r_3, k_3) by first having process i execute the same events that it executes in r_2, then by having process j execute the same events that it executes in r_1 (including the now-false suspicion $\mathit{failed}_j(i)$), and by delaying the receipt of all messages sent by i and j from point k until after k_3. Thus, in (r_3, k_3) both i and j consider themselves the leader, violating the assumption that $E + F$ is an election protocol.

□

Theorem 3 *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.

□

[2] There are no failure detections in this run, so Theorem 1 can be applied.

Theorems 1 and 2 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 4 *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 ([3]) 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 having processes never repent suspicions: suspicions are never false. Therefore, a failure detector that satisfies Strong Accuracy and Very Weak Completeness can be used to implement a Perfect Failure Detector. This theorem follows from Theorem 2. □

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.

7 Other Problems that Require a Perfect Failure Detector

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 ([2]) 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. In fact, 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 are also problems that do not resemble Election but require a Perfect Failure Detector. The Terminating Reliable Broadcast problem ([10,4]) is 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 [4], it is claimed (without proof) that the weakest failure detector for solving Terminating Reliable Broadcast is a Perfect Failure Detector. We do not know whether Election is equivalent to Terminating Reliable Broadcast.

8 Conclusion

In this paper, 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 crash system. We are not the first to show that there are problems harder than Consensus. The first such result that we are aware of is [15] in which the authors show that Primary Partition Virtually-Synchronous Communication (PP-VSC) cannot be implemented with the weakest failure detector that can implement Consensus. This problem arises in systems that use Group Membership protocols and resembles the Terminating Atomic Broadcast problem mentioned above. [15] does not establish the weakest failure detector that solves 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.

We believe that there are problems harder than Election as well. One can define failure detectors that are stronger than a Perfect Failure Detector. For example, we define in [13] a failure detector that is not only perfect, but also guarantees that a 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 ([2]).

From the above, there appears to be a hierarchy of problems ranked according to the type of failure detector that they require. However, we do not know how one would specify a "hardest" problem. Furthermore, such a hierarchy would require a parallel hierarchy of failure detectors ranked by their strength. Such hierarchies exist (e.g., [3]), but it is not clear whether there is any one hierarchy that can be used to express all sensible weakenings and strengthenings of a Perfect Failure Detector. For example, one could imagine weakening a Perfect Failure Detector by having it return a set of processes of which at least one has failed. Although we do not believe so, it may be possible to solve Election using a failure detector weakened in this manner.

This paper was motivated by our study of the Group Membership problem [12,1,9]. Systems that use Group Membership protocols seemingly solve Consensus, which is impossible in an asynchronous system. Because of this, there has been recent research into what a correct yet implementable specification for Group Membership should be [8]. Yet, systems that support Group Membership protocols also seemingly allow Election to be solved. This is because with a Group Membership protocol, the failure and recovery of group members are events whose occurrence and order are agreed upon by all surviving group members. Election is therefore trivial: for example, the oldest surviving process can be the leader. The results of this paper have led us to believe that it is equally important to understand the impact on Group Membership of needing an even stronger failure detector than that needed for Consensus [14].

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Proceedings of the IEEE 13th International Conference for Distributed Computing Systems*, pages 551–560, May 1993.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. 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.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing.* ACM, August 1991.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. Technical Report TR94-1458, Department of Computer Science, Cornell University, October 1994.

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

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

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

[8] D. Malki, K. Birman, A. Ricciardi, and A. Schiper. Uniform actions in asynchronous distributed systems. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 274–283. ACM, Aug. 1994.

[9] 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.

[10] S. Mullender, editor. *Distributed Systems*, chapter ? ACM Press frontier series. Addison-Wesley, second edition, 1993.

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

[12] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 341–352. ACM, Aug. 1991.

[13] L. Sabel. *Approximating Fail-Stop in Asynchronous Distributed Systems.* PhD thesis, Cornell University, 1995.

[14] L. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, pages 138–147. IEEE, Oct. 1994.

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