Unifying Self-Stabilization And Fault-Tolerance (Preliminary Version)

Ajei S. Gopal   Kenneth J. Perry
I.B.M. T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598
ajei@watson.ibm.com, kjp@watson.ibm.com

Abstract

In this paper we combine two previously disparate aspects of reliable distributed computing – self-stabilization, *i.e.*, tolerance of systemic failures, and fault-tolerance, *i.e.*, tolerance of process failures. We define what it means for a protocol to solve a problem while tolerating *both* types of failures and demonstrate a "compiler" that transforms a process failure-tolerant protocol for a synchronous system into a *process and systemic* failure-tolerant protocol. For asynchronous systems, we present a protocol that solves a crucial problem (Consensus) while tolerating both process and systemic failures.

1 Introduction

In this paper we combine two previously disparate aspects of reliable distributed computing: self-stabilization and fault-tolerance. A program is said to be *self-stabilizing* if it eventually achieves its intended behavior regardless of the state in which execution commences. This models *systemic failures* that corrupt the memory of a process but leave its program unchanged. A program is said to be *fault tolerant* if it always achieves its intended behavior regardless of the possibility of some processes deviating from their protocols. This models *process failures*, e.g., crashing, omission faults, arbitrary behavior. For the first time, we present protocols that tolerate both systemic and process failures in a non-trivial way. Specifically, following a systemic failure, we require that the program eventually resumes its intended behavior even if process failures continue to occur.

The difficulty in overcoming the combined failure types stems from their interaction. In our new model, the states of all the processes can be spontaneously corrupted during execution, and furthermore, some processes may not correctly follow their protocol. This contrasts with the situation that arises in the design of protocols tolerant of only process failures (in the new model, program invariants can be violated in the initial state) or only systemic failures (in the new model, once execution begins a faulty process may not follow its protocol even if the initial state were perfect). To illustrate, even though crashing may be the only type of process failure admitted, a process with a corrupted initial state that faithfully obeys its protocol can still send an illegal message, thereby manifesting behavior akin to a malicious process failure. Similarly, a self-stabilizing protocol for re-establishing a program invariant may be thwarted if omission process failures are admitted and, as a result, some process continually sends improper messages. Thus someone familiar with designing protocols that tolerate only one of the failure types finds a situation in the new model in which previously acquired experiences and intuitions no longer apply.

1.1 Overview of results

Our first contribution is to define in a non-trivial manner what it means for a protocol to solve a problem while tolerating systemic and process failures. The trivial definition – a protocol eventually resumes its intended behavior following the *final* systemic *and* process failure – was avoided in favor of the more stringent requirement that the protocol overcome systemic and process failures that may *continually* occur. Loosely speaking our definition is that, within a bounded number of steps (called the *stabi-*

12th ACM Symposium on Principles on Distributed Computing, Ithaca NY
© 1993 ACM 0-89791-613-1/93/0008/0195...$1.50

lization time) following the occurrence of a systemic failure and despite the presence of continual process failures, the protocol's behavior converges to that of a process failure-tolerant protocol that solves the same problem but begins in the "good" initial state.

Our next contribution is three-fold: we show that, in general, it is not possible to solve problems with a *finite* stabilization time; we precisely identify the *de-stabilizing events* that are the source of the impossibility; and, as a result, we introduce a weaker definition called *piece-wise stability*. This definition requires that, in any sufficiently long interval that contains no de-stabilizing events, after a finite number of steps, the behavior of the protocol becomes equivalent to that of a protocol that always begins in a "good" state and overcomes only process failures. Thus, our definition requires that useful work be accomplished in the presence of continuing process failures.

We next demonstrate a "compiler" that transforms a process failure-tolerant protocol for a synchronous system into a *process and systemic* failure-tolerant protocol. The significance of this result is that much of the large body of existing process failure-tolerant protocols automatically can be made self-stabilizing, and that a programmer familiar with overcoming only process failures also can overcome systemic failures without further effort. In presenting the transformation, we identify and solve a problem, called *round agreement*, that seems to be fundamental to the ability to tolerate both types of failures. We present a protocol that solves this problem while overcoming both process and systemic failures, and show how this protocol forms the basis for the compiler.

For asynchronous systems, we present a protocol that solves a crucial problem (Consensus) while tolerating both process and systemic failures. This protocol draws on the exciting new result of Chandra and Toueg[CT91], which shows that an unreliable failure detector circumvents the impossibility of achieving consensus. Our contributions are *process and systemic* failure-tolerant protocols that: implement an Eventually Strong Failure Detector, given an Eventually Weak Failure Detector; and achieve consensus, relative to an Eventually Strong Failure Detector.

1.2 Comparison with Related work

Most of the previous results related to ours have considered either systemic or process failures in isolation, but not together. In contrast, our results are among the first to address both types of failures simultaneously [Lam86].

Among the work only dealing with process failures, the emphasis has been on protocols for specific problems, e.g., the Reliable Broadcast, Consensus and Byzantine Agreement problems [LSP82, Had84, ST87b, BDDS87, DLS88, HH90], Clock Synchronization[HSSD84, ST87a], etc. The novel work on asynchronous Consensus [CT91, CHT92] is particularly relevant in that it serves as the basis of our asynchronous results.

Similarly, since the concept of self-stabilization was first introduced by Dijkstra[Dij74], most results on tolerating systemic failures have focussed on solutions to particular problems. There is little among the work prior to ours in which any failure type other than systemic failures was considered. The exceptions deal with restricted, intermittent failures in the communication system (e.g., lost messages, links crashing and recovering) [AB89, ASV91] and emphasize self-stabilizing protocols for reliable communication. Such failures are less general than the process failures that we deal with. Although Arora and Gouda[AG92] have formulated a definition of fault-tolerance that includes both systemic and process failures, they do not accommodate the occurrence of process failures following the final systemic failure. This is the "trivial" definition that we avoid. Instead, we require that the system eventually reach a legal state even if process failures continually occur, so long as no further systemic failures occur.

There has been previous work on automatically transforming an ordinary (i.e., fault-intolerant) protocol into one that is either tolerant of process failures [Sch90] or systemic failures[KP90, AKY90, ASV91, AV91], but not both. Related work focuses on methods for translating a program tolerant of one type of process failure (such as crashes) into one tolerant of a less restrictive type of process failure (e.g., Byzantine)[NT90], and on programming abstractions that facilitate the construction of process failure-tolerant programs (e.g., ISIS project[BJ90]). The "compiler" presented in this paper is the only automatic method for producing programs that are simultaneously tolerant of both systemic and systemic failure.

Although there may appear to be a superficial relationship between tolerating systemic failures and tolerating malicious (*i.e.*, Byzantine) process failures, there are significant differences. Although both types of failure may result in corrupted process states, much previous work has established that tolerating malicious process failures requires that at most a fraction (typically, one-third) of the processes in the system be able to fail. In contrast, the object of tolerating systemic failures is to be resilient to the corruption of the state of *every* process in the system.

2 Synchronous systems

To facilitate a brief presentation in this preliminary version of the paper, we make some simplifying assumptions. We assume a perfectly synchronous, completely-connected network of processes able to communicate only by message-passing (i.e., all processes take steps at the same time and message delivery time is constant). In such a system, a computation can be described as proceeding in *rounds*, beginning with the round numbered 1. Much of the previous work on protocols tolerant of process failures deals with terminating round-based protocols but it has been previously shown that terminating protocols cannot tolerate systemic failures[KP90]. To accommodate both failure types, in this abstract we restrict our attention to problems whose solutions are non-terminating protocols composed of repeated executions of a terminating sub-protocol, e.g., a non-terminating protocol for Repeated Consensus constructed by iterating a terminating protocol for a single Consensus.

The only process failures considered in this preliminary version of the paper are of the general omission type (*i.e.*, send and/or receive omission, and/or crashing). Furthermore, in keeping with the tradition in the literature on self-stabilization, we concentrate on the behavior of the processes following the *final* systemic failure. No generality is lost because we ensure a bounded stabilization time. Thus, our results necessarily hold in *each systemic failure*-free interval (not just a final suffix) of the execution whose length exceeds this time.

2.1 Model

Protocols and failures

A round-based protocol for a system is specified by a collection of initial states and transition functions, one per process in the system. One variable, c_p, is distinguished at each process p and is intended to contain a value that p considers to be the current round number. Because of systemic failures (to be defined) the value in this variable may differ from the *actual round number* of an execution, which is the true duration of the execution in rounds according to an external observer. When numbering rounds in our presentation, we use the actual round number. After completing the actions to be taken in some round, each correct process p increments c_p by one. Let c_p^r denote the value of c_p at the *start* of round r, and let s_p^r denote the rest of p's state at the start of round r. Should process p crash in some round, c_p^r and s_p^r become undefined for all subsequent rounds.

We consider two types of failures. A ***process failure*** occurs when a process fails to follow its protocol; for example when a process crashes, fails to send or receive a message, or fails to follow its protocol. Such a process is called *faulty* and we bound the number of such processes by the quantity f. A ***systemic failure*** (also called a self-stabilization failure) occurs when a process commences execution in a state other than the initial state specified in the protocol. Note that a process that correctly follows its protocol relative to the state in which execution commences is *not* faulty even if this initial state is other than the one specified by the protocol; a process becomes faulty only if it deviates from its protocol.

Histories

A *round history of round* r of a system is a vector that, for each process in the system, describes the state of the process at the start of round r and the actions taken by the process during round r. An *execution history* (or *history*) H of a system is a possibly infinite sequence of round histories. A history H of a system is *consistent* with a protocol Π for the system if the actions (including process failures) taken by the processes in the history could have been the result of executing protocol Π in the system when the initial state of each process is as given in the first round history of H. Given both a protocol Π and a history H for a system, it is possible to identify the set $\mathcal{F}(H,\Pi)$ of faulty processes (i.e., those performing an action different than what is required by Π) and the set $\mathcal{C}(H,\Pi)$ of correct processes.

When H, Π and the system are clear from the context, we simply say, e.g., "a process is correct" rather than the more precise "a process in a system is correct for a protocol Π and history H of the system that is consistent with Π."

To model systemic failures, the global state at the start of the first round in H may be completely arbitrary: the state of each process p is not necessarily that specified by Π and the value stored in c_p is not necessarily 1. Suppose $H = H' \cdot H''$ and $|H'| = r$. The histories H' and H'' are called the ***r-prefix*** and ***r-suffix*** of H respectively. We note that both H' and H'' are consistent with Π.

Problems

A problem specifies the required behavior of processes in an execution and is formally defined as a predicate on a history and a set of faulty processes. For simplicity we assume that any problem solved by a round-based protocol requires that the correct processes agree on the round number in each round, and increment the round number by one at the end of each round:

Assumption 1 *Suppose Σ is a problem, H is a history of a system and F is a set of processes that are faulty in H. If $\Sigma(H, F)$ is satisfied, then for all rounds $r \in H$:*

1. Agreement*: For all processes p and q not in F, $c_p^r = c_q^r$ in H.*
2. Rate*: For all processes p not in F, $c_p^{r+1} = c_p^r + 1$ in H.*

Recall that, because of systemic failures, c_p^r need not equal r even if p is correct and $\Sigma(H, F)$ is satisfied.

Solving a problem

Consider a system which permits process failures but does not permit systemic failures. A protocol solves a problem Σ in such a system if all histories H consistent with the protocol satisfy the problem's specification.

Definition 2.1 *A protocol Π for a system subject to process failures and not subject to systemic failures ft-solves a problem Σ in the system, if for all histories H of the system that are consistent with Π, $\Sigma(H, \mathcal{F}(H,\Pi))$ is satisfied.*

Consider a system which permits systemic failures but does not permit process failures. A protocol solves a problem in such a system if all histories eventually "converge" to a history that satisfies the specification when no processes are assumed to be faulty.

Definition 2.2 *A protocol Π for a system subject to systemic failures and not subject to process failures ss-solves a problem Σ in the system with stabilization time r, if for all histories H of the system that are consistent with Π, $\Sigma(H', \phi)$ is satisfied, where H' is the r-suffix of H.*

A "natural" (but weak) way of combining the above definitions of ft-solves and ss-solves so as to accommodate both process failures and systemic failures is:

Tentative Definition 1 *A protocol Π solves a problem Σ with stabilization time r if and only if for all histories H consistent with Π, $\Sigma(H', \mathcal{F}(H,\Pi))$ is satisfied, where H' is the r-suffix of H.*

However, this tentative definition is too weak to be useful. Consider, for example a history H for some problem Σ in which process p, due to omission failures, does not communicate with any other process until round $r + 1$ for some large r. Furthermore, suppose Σ is satisfied until p reveals itself; that is, Σ is satisfied on some suffix of the r-prefix of H, but not on any suffix of the $(r + 1)$-prefix of H. Since p could delay revealing itself for an arbitrary number (perhaps infinite) of rounds, the tentative definition becomes meaningful only following the final process failure in H. Yet it is clear that some useful work might be accomplished prior to this (since Σ was satisfied). This argument is formalized in the following theorem, which says that, with the tentative definition, there are problems for which there exists no protocols that solves the problems with finite stabilization time.

Theorem 1 *For all problems Σ, for all finite times r, a round-based Π cannot solve Σ with stabilization time r, when solve is as defined in Tentative Definition 1.*

Intuitively, the proof of Theorem 1 hinges on the inability of a process to determine how it arrived at its present state: it may have attained the state through a positive number of transitions from the initial state, or by virtue of the present state being the initial state. In particular, when a correct process p receives a message when its round variable equals c_p^r but the message was sent by a process q whose round variable equaled c_q^r when it was sent, p cannot determine which of the following cases is true.

In the first case, process q is correct and a systemic failure caused p and q to have different round numbers. In this case, p and q have to cooperate to reach agreement on the current round number (as required by Assumption 1). In the second case, process q is faulty and failed to "join" the computation earlier. In this case, p is free to ignore q's incorrect view of the rounds.

Proof Outline: For the sake of contradiction, assume that there is some problem Σ that is solved by some protocol Π with finite stabilization time r. The proof is by scenario.

Consider an infinite history H consistent with protocol Π in a system with two processes, p and q, with an initial state in which p and q store different values (due to a systemic failure) in the local variables indicating the round number. Moreover, let H have an r-prefix H' in which p and q do not communicate (due to omission type process failures) and an r-suffix H'' in which no process failures occur. It is straightforward to show that, in general, the lack of communication between p and q in H' ensures that p and q do not agree on the current round number or state at the end of H'. Thus, at the start of round $r+1$ in H (which is the initial state of H'') $c_p^{r+1} \neq c_q^{r+1}$ in H.

There are two scenarios that explain the above behavior.
Scenario 1: Process p is correct, process q is faulty, and the lack of communication in H' is caused only by the process failures of q. Since we have assumed that Π solves Σ with stabilization time r, $\Sigma(H'', \{q\})$ must be satisfied. By the rate condition of Assumption 1, p's current round number increases by 1 in each round in H''; *i.e.*, $\forall r' \geq r$, $c_p^{r'+1} = c_p^{r'} + 1$ in H.
Scenario 2: Same as above, but with the roles of p and q reversed. We conclude that $\forall r' \geq r$, $c_q^{r'+1} = c_q^{r'} + 1$ in H.

But $c_p^{r+1} \neq c_q^{r+1}$ so the conclusion that both p and q increase their round numbers by 1 in each round after r leads to the conclusion that c_p^r and c_q^r are never equal in H''.

Now consider a third scenario.
Scenario 3: There is another execution G of Π which is identical to H'' and in which p and q are correct. By the assumption that Π solves Σ with stabilization time r, it must be the case that Σ is satisfied on an r-suffix of G. But we have just shown that the round variables of p and q never agree in H'', contradicting the agreement condition in Assumption 1. Thus Σ is not satisfied on an r-suffix of G, yielding a contradiction. □

Since the tentative definition is not adequate, we propose an alternative: Informally, during any sufficiently long interval of a history in which the system has been "stable for long enough", the predicate Σ must be satisfied on a suffix of the interval. When a faulty process suddenly reveals itself, the predicate may temporarily become falsified but this incorrectness is transitory and Σ once again must become satisfied when the interval following the revelation becomes "stable for long enough". This is called *piece-wise* stability. The informal notion of "stable for long enough" can be made formal through the concept of a *coterie*, which is defined to be the set of processes that have communicated (perhaps transitively through another process) with all correct processes, as defined below. We will also show that a change in the coterie is the *de-stabilizing event* that "ruins" a predicate Σ.

Consider a history H. Let $p \rightarrow_H q$ denote that, in history H, there is some event executed by p that *happened-before* some event executed by q, where happened-before uses the standard definition of causality introduced by Lamport [Lam78].

Definition 2.3 *Suppose a history H of a system is consistent with a protocol Π. The coterie of H with Π, denoted $coterie_{\Pi}(H)$, is the set of processes p such that, for all processes $q \in \mathcal{C}(H,\Pi)$, $p \rightarrow_H q$.*

We can now state a useful definition for a protocol solving a problem in a manner tolerant of both systemic and process failures. A protocol solves a problem Σ if the following holds for all histories H consistent with the protocol: once the coterie has been unchanged for long enough, then as long as the coterie remains unchanged, the problem's specification is satisfied. If the coterie changes, the problem specification need not be satisfied until the coterie re-stabilizes.

Definition 2.4 *A protocol Π ftss-solves a problem Σ with stabilization time r if and only if for all histories H consistent with Π, for all histories H_1, H_2, H_3 and H_4 such that $H = (H_1 \cdot H_2 \cdot H_3 \cdot H_4)$, $coterie_{\Pi}(H_1) = coterie_{\Pi}(H_1 \cdot H_2) = coterie_{\Pi}(H_1 \cdot H_2 \cdot H_3)$ and $|H_2| \geq r$, then $\Sigma(H_3, \mathcal{F}(H_1 \cdot H_2 \cdot H_3, \Pi))$ is satisfied.*

2.2 Restricting the behavior of faulty processes

The specification of a problem usually restricts the behavior only of correct processes. However, there is a class of problems in which the behavior of faulty processes is also restricted [Nei88, Gop92]. In this section, we show that problems in this class are not amenable to protocols tolerant of both systemic and process failures. To restrict our attention to non-trivial problems in this class, we assume that any protocol that restricts the behavior of a faulty process enforces the following stronger form of the agreement condition of Assumption 1:

Assumption 2 *If $\Sigma(H, F)$ is satisfied, then for all rounds $r \in H$:*

- Uniformity: *For all processes $p \in F$, either p has halted by round r or $c_p^r = c_q^r$, where $q \notin F$.*

This assumption formalizes the widely-used technique[NT90, GT89, GT91, NT93] of "self-checking and halting before doing any harm" that has proven to be valuable in the design of protocols tolerant of process failures. Theorem 2 implies that this technique cannot be used in our model.

Theorem 2 *For all problems Σ, for all finite times r, a uniform round-based Π cannot ftss-solve Σ with stabilization time r.*

```
// $r$ is the round number as seen by an external observer and is unavailable to $p$

At the start of round $r$:
  $p$ sends $(ROUND: p, c_p^r)$ to all        // $p$ broadcasts its current round number

At the end of round $r$:
  $R := \{c \mid p$ received $(ROUND: q, c)$ in this round $\}$
  $c_p^{r+1} := max(R) + 1$                   // $p$ updates its round number
```

Figure 1: Round agreement protocol: process p's actions in each round r

Proof Outline: For the sake of contradiction, assume that there is some problem Σ that is solved by some protocol Π with finite stabilization time r. The proof is by scenario.

Consider an infinite history H consistent with protocol Π in a system with two processes, p and q, with an initial state in which p and q store different values (due to a systemic failure) in the local variables indicating the round number. Moreover, let H be such that p and q *never* communicate (due to omission type process failures) and let H' and H'' be an r-prefix and r-suffix of H. By reasoning identical to that used in the proof of Theorem 1, we may conclude that, in general, $c_p^{r+1} \neq c_q^{r+1}$ in H.

Scenario 1: Process p is faulty and process q is correct. Since p never communicates with q, the coterie in H' remains constant.

Since we have assumed both that Π ftss-solves Σ with stabilization time r, and since the coterie of H' is constant, $\Sigma(H'', \{p\})$ must be satisfied. By the uniformity condition of Assumption 2, since $c_p^{r+1} \neq c_q^{r+1}$ in H, we conclude that p must halt itself by the end of the first round of H''.

Scenario 2: There is another execution G of Π which is identical to H'' and in which p and q are correct. Since this is indistinguishable from the previous scenario, p must halt itself in G and therefore the value stored in its round variable cannot forever remain equal to the value stored in the round variable of q. Since both p and q are correct in this scenario, the agreement condition in Assumption 1 is violated, contradicting the assumption that Π ftss-solves Σ. □

2.3 Round agreement

In Figure 1, we present a protocol tolerant of systemic and process failures that ensures that eventually, all correct processes continually agree on a common number for the current round [1]. This serves two purposes. First, the proof of correctness for the protocol indicates the general flavor of proofs of correctness in our model. Second, the round agreement protocol is the basis of the "compiler" developed in the next section that transforms a protocol that *ft*-solves a problem into one that *ftss*-solves the problem.

Theorem 3 *The protocol in Figure 1 is a ftss-protocol with stabilization time of 1 round that ensures that all correct processes agree on the current round-number.*

Proof Outline: Let H be any history consistent with the protocol in Figure 1. Let F^i denote the set of processes that are faulty by the end of round i in H. To prove the theorem, suppose that the coterie remains constant from rounds x to y. We must show that $c_p^r = c_q^r$ for all correct processes p and q (*i.e.*, $p, q \notin F^r$) and for all rounds r such that $x < r \leq y$. Note that since the stabilization time for the protocol is 1 round, we are not required to show that $c_p^x = c_q^x$.

Suppose for the sake of contradiction that there is some round i, $x < i \leq y$, and that there are two processes $p, q \notin F^i$ for which $c_p^i > c_q^i$. Since both p and q are correct, q received c_p^{i-1} from p in round $i-1$ (and updated its own clock accordingly), and hence the difference between p's and q's clocks in round i is because p received a $(*, c-1)$ message[2] at the end of round $i-1$ from some process $u \neq p$, where $c = c_i^p$. Thus, process u had a round number of $c - i$ in the initial state of H (*i.e.*, $c_u^1 = c - i$) and $u \rightarrow p$ in the first $i-1$ rounds of the history H.

Since $c_p^i > c_q^i$, q did not receive a message $(*, c-1)$ by the end of round $i-1$, and hence $u \not\rightarrow q$ in the first $i-1$ rounds of history H. Since $u \rightarrow p$ and $u \not\rightarrow q$

[1] All of our protocols assume that any process, correct or faulty, correctly receives its own broadcast.

[2] We use a $*$ in the tuple $(*, c-1)$ to indicate that the value in the first field is not relevant.

```
// $r$ is the round number as seen by an external observer and is unavailable to $p$

Initialization:
  $s_p^1 := s_{p,init}$
  $c_p^1 := 1$

At the start of round $r$:
  $p$ sends $(STATE: p, s_p^r)$ to all          // $p$ broadcasts its current state

At the end of round $r$:
  $M := \{m \mid p$ received $m$ in this round $\}$
  $s_p^{r+1} :=$ function$(p, s_p^r, M, c_p^r)$     // $p$ updates its state
  $c_p^{r+1} := c_p^r + 1$                      // $p$ updates its clock
  if $c_p^r = final\_round$ then halt           // $p$ halts in the final round
```

Figure 2: Fault-tolerant full-information protocol Π: process p's actions in each round r

in the first $i-1$ rounds of H, and p and q are both correct ($p, q \notin F^{i-1}$), u is not in the coterie in the first $i-1$ rounds.

Since p and q are not in F^i, p sends a message in round i that is received by the end of round i by all correct processes. Since $u \rightarrow p$ in the first $i-1$ rounds of H, we conclude that for all processes $v \notin F_i$: $u \rightarrow v$ in the first i rounds of H; thus u is in the coterie in round i.

The above two paragraphs show that process u enters the coterie in round i. This contradicts the original choice of round i and completes the proof. □

2.4 Protocol translation

In this section we present an automatic "compiler" that transforms a large class of process failure-tolerant protocols into process and systemic failure-tolerant protocols. The significance of this result is that much of the large body of existing process failure-tolerant protocols can automatically be made self-stabilizing, and that a programmer familiar with overcoming only process failures also can overcome systemic failures without further effort.

We only transform process failure-tolerant protocols that have the following three properties. First, the protocols are round-based *full-information* protocols. This is a minor restriction, because any protocol that is not full-information easily can be transformed into such a protocol. Second, the protocols do not restrict the behavior of faulty processes. Recall Theorem 2 showed that it is impossible for any protocol to restrict the behavior of faulty processes with a finite stabilization time; hence no such protocol can be constructed by our compiler. Third, the current round number is counted by an unbounded variable. In the full paper, we show an impossibility for a bounded counter analogous to the impossibility shown in Theorem 2.

Protocol Π (Figure 2) is assumed to be a single iteration of a protocol that is to be repeated forever (e.g., a protocol for a Single Consensus, which is used as the basis of a protocol for Repeated Consensus, as discussed in the Model section). Our compiler produces protocol Π^+ which infinitely repeats a modified version of Π and is tolerant of systemic and process failures. Therefore, the problem Σ^+ that is solved by Π^+ is defined as follows:

> $\Sigma^+(H, F)$ is satisfied if and only if there are histories $H_1 \dots H_i \dots$ such that $H = H_1 \cdot \dots H_i \cdot \dots$ and $\forall i : \Sigma(H_i, F)$ is satisfied.

In other words, Π^+ repeatedly solves problem Σ.

Assume that protocol Π is given in the canonical form shown in Figure 2 and terminates in *final_round* rounds. The protocol Π^+ (Figure 3) is obtained by superimposing[Kat87, BF88] the round agreement protocol (Figure 1) onto Π. Protocol Π becomes "controlled" in the superimposition in the following sense: when the value of process p's round variable equals c_p^r, Π^+ causes p to execute round $k = c_p^r \ mod \ (final\_round) + 1$ (c_p^r converted to the range of rounds $1 \dots final\_round$ used by Π) of Π. Moreover, following the last round of the current iteration of Π, protocol Π^+ re-establishes an initial state of Π so that another iteration can begin anew.

```
// $r$ is the round number as seen by an external observer and is unavailable to $p$
// The function $sender(m)$ is the sender of a message $m$
// The function $round(m)$ is current round of $sender(m)$ when it sent $m$
// The function $normalize(c)$ converts a round number $c$ into the range $1 \ldots final\_round$
//                 $normalize(c) := c \bmod (final\_round) + 1$
// A "*" in a message's contents indicates a field of the message that is not currently required

At the start of round $r$:
  // $p$ broadcasts its current round and its current state
  $p$ sends $((STATE: p, s_p^r), (ROUND: p, c_p^r))$ to all processes

At the end of round $r$:
  // $p$ omits from $M$ any message sent by a process that it suspects
  $S := suspect_p^r \bigcup \{q \mid p$ did not receive $m,\ sender(m) = q,\ round(m) = c_p^r$ in this round $\}$
  $M := \{m \mid p$ receives $x = (m, *),\ sender(x) \notin S$ in this round $\}$

  $k := normalize(c_p^r)$                    // $p$ converts $c_p^r$ into the range $1 \ldots final\_round$
  $s_p^{r+1} :=$ function$(p, s_p^r, M, k)$          // $p$ updates state according to $\Pi$
  $suspect_p^{r+1} := S$                    // $p$ updates its suspect set

  $R := \{c \mid p$ received $((*), (ROUND: q, c))$ in this round $\}$
  $c_p^{r+1} := max(R) + 1$                 // $p$ updates it current round number

  // $p$ resets its state and suspect state on the start of a new iteration
  if $normalize(c_p^{r+1}) = 1$ then
    $s_p^{r+1} := s_{p,init}$                   // $p$ resets its state
    $suspect_p^{r+1} := \phi$                  // $p$ resets its suspect set
```

Figure 3: Protocol Π^+ obtained from protocol Π in Figure 2: process p's actions in each round r

Actually, the superimposition modifies Π slightly so as to protect it from the effects of systemic failures for which it was not designed. To illustrate, consider an execution of Π^+ in which the coterie has been unchanged for a long time, a faulty process q is in the coterie, and the value of q's round variable is smaller than that of any correct process. Any message sent by q is "out-of-date" and, as such, could lead to the falsification of Σ if it were not ignored by the correct processes. An even more insidious problem arises if q's round variable is "correct" but its state is otherwise corrupted. In order to insulate the correct processes from the effects of these potentially harmful messages, the superimposition causes Π to "tag" each message with the value of the sender's round variable. These tags enable the superimposition to maintain a set *suspect* at each process p, which used to filter messages: when changing state, p ignores messages from any member of the set.

The set *suspect* is intended to contain only those processes that p suspects of being faulty. A process q is added to p's *suspect* set if p fails to receive an expected message from q, or if q's message indicates that p and q disagree on the current round number. The *suspect* set is reset to empty at the end of each iteration. Because of systemic failures, the set may also initially contain some correct processes. This can extend the stabilization time of Π^+ by as much as *final_round*, the duration of Π.

Theorem 4 *If the protocol Π shown in Figure 2 ft-solves a problem Σ, then the protocol Π^+ shown in Figure 3 ftss-solves the problem Σ^+ with stabilization time of final_round.*

The outline of the proof of Theorem 4 is similar to that of Theorem 3. The crux is that whenever a faulty process "destabilizes" the computation, it enters the coterie by the next round. In the worst case, this causes each correct process to reset the variable indicating its current round number and (assuming the coterie remains stable) to take no more than another *final_round* rounds to reset its state and *suspect* set. Thereafter, for as long as coterie remains stable, the correct processes proceed in synchrony with one another.

A full proof of Theorem 4 is left for the full paper.

3 Asynchronous Systems

Both the protocol for round agreement and the "compiler" for perfectly synchronous systems readily adapt to synchronous, but not perfectly synchronized systems, We now turn our attention to *asynchronous* systems, e.g., ones in which unbounded differences in process speeds, message delivery times, etc, are admitted. In doing so, we are confronted with previous results[FLP85, BMZ88] demonstrating the impossibility of protocols that solve non-trivial problems and overcome process failures. The resulting dearth of protocols for asynchronous systems caused us to focus on implementing a process and systemic failure-tolerant protocol for a particular problem that has a solution: Consensus (relative to a Failure Detector) in the presence of *crash* type process failures.

Solving Consensus using an imperfect Failure Detector to circumvent the impossibility result was pioneered by Chandra and Toueg[CT91], whose results we briefly summarize. Chandra and Toueg show how Consensus can be solved given a very weak detector called an Eventually Weak Failure Detector. Their method is to transform an Eventually Weak Failure Detector into a more powerful Eventually Strong Failure Detector, and then use the more powerful detector in a protocol that solve Consensus. The difference between the two types of detectors is that an Eventually Weak Failure Detector satisfies the following property:

Weak Completeness: Eventually every faulty process is suspected by at least one correct process.

while an Eventually Strong Failure Detector satisfies the stronger:

Strong Completeness: Eventually every faulty process is suspected by at all correct processes.

Both types of detectors have the following property:

Eventually Weak Accuracy: Eventually there is at least one correct process that is not suspected of being faulty by any correct process.

Note that both types of detectors can erroneously suspect correct processes.

The development of our asynchronous protocol for Consensus that tolerates both crash type process failures and systemic failures mirrors that of [CT91]: given some Eventually Weak Failure Detector, we implement an Eventually Strong Failure detector that tolerates both types of faults and subsequently use this detector in a version of the Consensus protocol of [CT91] that we have made tolerant of both fault types. We briefly describe both steps in this abstract, deferring a more complete description to the full paper.

The protocol shown in Figure 4 is followed by each process p in determining the state ("dead" or "alive") of another process s. The union of such protocols over each process s gives the Eventually Strong Failure Detector for process p that is tolerant of both process and systemic failures. We assume that the Eventually Weak failure detector (upon which our protocol is based) repeatedly sets the predicate *detect*(s) as long as s is suspected of being faulty.

Note that unlike Chandra and Toueg's Eventually Strong Failure Detector, the protocol in Figure 4 does not require any initialization; *i.e.*, it is guaranteed to work correctly irrespective of the initial state of the processes. This makes the protocol tolerant of systemic failures.

Theorem 5 *Assuming that detect(s) is managed by an Eventually Weak Failure detector, the protocol if Figure 4 is an Eventually Strong Failure detector that overcomes both process and systemic failures.*

Proof Outline: (*Strong Completeness*): Suppose s is faulty. Since *detect*(s) is controlled by an Eventually Weak failure detector, this predicate will become repeatedly true at some correct process p. Process p spontaneously increases *num*[s], and each time it does so, it broadcasts an "s is dead" message. Suppose q is a correct process that has *state*[s] equal to "dead." Process p eventually broadcasts an "s is dead" message tagged with a value of *num*[s] that is larger than q's value of *num*[s], thereby causing q to set *state*[s] to "dead."
(*Eventual Weak Accuracy*): Suppose s is the correct process that is not be suspected by any correct process in the Eventually Weak failure detector. Process s is the only process enabled to spontaneously increase its copy of *num*[s], and each time it does so, it broadcasts an "s is alive" message. As in the previous case, every correct process q eventually sets *state*[s] to "alive." □

We next very briefly describe how to derive our Consensus protocol from the one in [CT91]). The original Chandra and Toueg protocol is structured as a repetition of rounds which are subdivided into phases. There are two main ideas to our derivation:

```
when detect(s):           $num[s] := num[s] + 1; state[s] :=$ "dead"

when ($p = s$):              $num[s] := num[s] + 1; state[s] :=$ "alive"

when true:                send ($s, num[s], state[s]$) to all

when deliver ($s, n, st$):   if ($n > num[s]$) then $num[s] := n; state[s] := st$
```

Figure 4: Eventually Strong Failure Detector: process p's actions to detect process s's status

- In our protocol, until a process completes a phases, it periodically re-sends every message required by the [CT91] protocol for that phase.

  This prevents a deadlock situation in which the initial state falsely indicates that every process has sent a message and that, as a result, causes all processes to wait for messages that have never been sent. This technique was previously used in [KP90].

- In our protocol, we superimpose a round agreement protocol on top of the [CT91] Consensus protocol.

  This round agreement causes the processes to eventually agree on the pair (round number, phase number) and controls the [CT91] protocol as follows. When the round agreement causes the round number to change, it causes all work of the currently executing phase to be abandoned and forces the process to begin the first phase of the newly changed round number.

  Omitting most details in this abstract, we note that the superimposition causes each message of the [CT91] protocol to be tagged with a round number so as to enable the controlling protocol to ignore messages from abandoned rounds.

As we prove in the full paper, the superimposition eventually causes a sufficient number of processes to participate in a common round and phase, and as a result, the behavior of our protocol subsequently mimics the behavior of the [CT91] protocol. Thus, the correctness of our protocol, which tolerates both process and systemic failures, is based on the controlling nature of our superimposition and the correctness of the original [CT91] protocol.

References

[AB89] Yehuda Afek and Geoff Brown. Self-stabilization of the alternating-bit protocol. In *Eighth Symposium on Reliable Distributed Systems*, 1989.

[AG92] Anish Arora and Mohamed Gouda. Closure and convergence: A formulation of fault-tolerant computing. In *Twenty-second Fault Tolerant Computing Symposium*, 1992.

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory efficient self stabilizing protocols for general networks. In J. van Leeuwen and N. Santoro, editors, *Proceedings of the Fourth International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science, pages 15–28. Springer-Verlag, September 1990. In press.

[ASV91] B. Awerbuch, B. Patt Shamir, and G. Varghese. Self stabilization by local checking and correction. In *Proceedings of the Thirty-second Symposium on Foundations of Computer Science*, pages 268–277. IEEE Computer Society Press, October 1991.

[AV91] B. Awerbuch and G. Varghese. Distributed program checking: A paradigm for building self-stabilizing distributed protocols. In *Proceedings of the Thirty-second Symposium on Foundations of Computer Science*, pages 258–267. IEEE Computer Society Press, October 1991.

[BDDS87] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement (preliminary report). In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 42–51, August 1987. Revised version received November 1988.

[BF88] Luc Bouge and Nissim Francez. A compositional approach to superimposition. In *The Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988. San Diego, California.

[BJ90] K. Birman and T. Joseph. Communication support for reliable distributed computing. In *Fault tolerant distributed computing*, pages 124–137. Springer-Verlag, 1990.

[BMZ88] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed tasks that are solvable in the presence of one faulty processor. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 263–275, August 1988.

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

[CT91] Tushar Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the Association for Computing Machinery*, 35(2):288–323, April 1988.

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

[Gop92] Ajei S. Gopal. *Fault-tolerant Broadcasts and Multicasts: The Problem of Inconsistency and Contamination.* PhD thesis, Cornell University, Department of Computer Science, January 1992.

[GT89] Ajei Gopal and Sam Toueg. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In J.-C. Bermond and M. Raynal, editors, *Proceedings of the Third International Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes on Computer Science*, pages 110–123. Springer-Verlag, September 1989.

[GT91] Ajei Gopal and Sam Toueg. Inconsistency and contamination (preliminary version). In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 257–272, August 1991.

[Had84] Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations.* PhD thesis, Harvard University, June 1984. Department of Computer Science Technical Report 11-84.

[HH90] Vassos Hadzilacos and Joseph Y. Halpern. Message and bit-optimal protocol for byzantine agreement. 1990. To appear.

[HSSD84] Joseph Y. Halpern, Barbara Simons, Ray Strong, and Danny Dolev. Fault-tolerant clock synchronization. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 89–102, August 1984.

[Kat87] Shmuel Katz. A superimposition control construct for distributed systems. Technical report, MCC-STP, MCC, Austin, Texas, 1987.

[KP90] S. Katz and K.J. Perry. Self-stabilizing extensions for message-passing systems. In *Ninth Symposium on Principles of Distributed Computing*, pages 91–101. ACM, 1990.

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

[Lam86] Leslie Lamport. The mutual exclusion problem: Part II – statement and solutions. *Journal of the Association for Computing Machinery*, 33(2):327–348, 1986.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Nei88] Gil Neiger. *Techniques for Simplifying the Design of Distributed Systems.* PhD thesis, Cornell University, August 1988. Department of Computer Science Technical Report 88-933.

[NT90] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[NT93] Gil Neiger and Mark R. Tuttle. Common knowledge and consistent simultaneous coordination. *Distributed Computing*, 6(3), 1993. To appear.

[Sch90] Fred B. Schneider. The state machine approach: a tutorial. *Computing Surveys*, 22(4):299–319, 1990.

[ST87a] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the Association for Computing Machinery*, 34(3):626–645, July 1987.

[ST87b] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.